

Code to Models

A Microcontroller from the Bottom Up

G. Andrew Mangogna

Table of Contents

Introduction	3
Target Readers	5
Overview	5
What is the bottom?	5
What is the top?	6
There Will Be Design	7
There Will Be Code	7
How to Read this Document	7
Move Along — Nothing New Here	8
Tap Root	9
The Path to main	10
Getting in on the Ground Floor	10
Executing from power up	11
Core Initialization	28
Starting main	43
Scripting the Linker	46
main at last	57
Summary	58
Crossing the Divide	60
Handler Mode / Thread Mode Split	60
Foreground / Background Interfaces	64
SVC Interface	71
System Realm Request Interface	80
Device Realm Request Interface	83
Background Notification Interface	87
Watchdog Timer	94
Foreground to Background Control Flow	107
Code Layout	118
Example	120
Batten Down the Hatches	122
Partitioning Memory Usage	122
Enforcing Memory Usage	124
Summary	135
The Tyranny of the Pins	137
Pin Function definitions	140
Sharing Pins	142
GPIO Logical Device	143
GPIO Services	148
GPIO IRQ Handler	160
Examples	161

Speak To Me	167
Design Overview	167
Representing a UART	169
UART Services	195
Servicing the UART	206
Console I/O	212
Logging to the Console	221
Privileged Transmission	221
Console I/O Example	224
Echo Example	225
Console Transmit Stress Test	226
Just a Matter of Time	228
Driving Software with Time	228
Epoch Time	271
Human Time	277
Computer Time	294
Shoring Up the Foundation	298
Software detected faults	298
Missing exception handler	307
Summary	316
Chip Whisperer	318
Apollo 3 I/O Master Peripheral	318
Representing the I2C Logical Device	319
IOM Device Control	321
I2C Logical Device Instantiation	331
I2C Services	333
Conclusion	345
Systems Programming of the Core	345
System Environment Concurrency	346
Peripheral Device Operations	346
Baton	348
The View from the Mesa	349
Core Execution Model Concepts	349
Reactive System Execution Environment (ReSEE)	350
Additional Background for Readers	351
The Execution Model	353
Executing with the Cerebral Computer	353
Computational Models of Languages	355
Higher Order Computational Models	356
Encoding Logic in Data	357
Sequencing Execution	363
Socratic Questions	370

Data Management Fundamentals	372
Relation Tuples	373
Relation Values	410
Relation Variables	503
Binary Relations Between Relation Variables	560
Adjacency Matrices for Associations	569
Adjacency Matrices for Generalizations	587
Execution Fundamentals	611
State Model Definitions	611
Representing State Models	613
Representing States	615
Representing Events	620
Representing the Transition Function	623
Representing Activity Functions	629
Transitioning State Machines	634
State Model Code Layout	637
Managing Application Data	639
Buckets of Moonbeams in my Hand	639
I Want to Hold Your Hand	677
Just My Type	740
Summary	756
Sequencing Execution	758
Relational Class Behavior	758
Association Behavior	769
Generalization Behavior	795
Bringing in the Sheaves	797
Domain Requirements	798
Defining a Domain	798
Execution Sequencing	813
Tracing Domain Execution	817
Domain Services for Data Model Transactions	844
Domain Services for Client Domains	844
Domain Code Layout	847
Button Blinky	849
Blinky LED State Model	849
Blinky LED State Machine	851
Bouncy Button State Model	851
Bouncy Button State Machine	854
An Example Domain	854
Background Notification Handling	860
Tracing execution	860
Execution sequence diagram	861

Evaluation	863
Islands in the Stream	865
Introduction	866
Supporting code	867
Memory Mapped I/O	868
Foundational Definitions	868
Naming Register Fields	869
Register Meta-programming	870
Indexed Registers	876
Single Bit Fields	878
Code Layout	880
Cortex v7e-M Core Access	883
Cortex specific instructions	883
Core registers	887
Cortex Exceptions	898
Cortex core peripherals	900
Code layout	918
Bipartite Buffer	919
Requirements	919
Bipartite Buffer Operations	919
Testing	923
Bipartite Buffer Code Layout	924
Formatted Output of Relation Tuples and Values	925
Design Concepts	925
Representing the Formatted Table	928
Code layout	959
Tagged Catalogs	960
Catalog Representation	960
Zig Traits Module	966
Supplemental materials	968
Bibliography	969
Glossary of Terms	970
Colophon	972
Appendix A: Literate Programming	973
Appendix B: Target Platform	974
Microcontroller architecture	974
SparkFun Micromod Board	974
Tool chain	974
Appendix C: Copyright Information	975
Appendix D: Edit Warning	977
Index	978

Revision History

Revision	Date	Author	Comment
0.1	April 3, 2021	GAM	Start of document.
0.2	April 27, 2021	GAM	Part I first draft completed.
0.3	June 2, 2021	GAM	Further refinements to Part I. Part II draft completed.
0.4	July 15, 2021	GAM	Draft of background / foreground execution synchronization if place.
0.5	August 11, 2021	GAM	Crossing the Divide part is finished. Some document reorganization.
0.6	September 9, 2021	GAM	Added Batten Down the Hatches to configure the MPU.
0.7	September 18, 2021	GAM	Added Time Mancy to implement a timer queue which will be needed later.
0.8	October 17, 2021	GAM	Completed coding work on Time Mancy. Improved explanations in other parts.
0.9	October 30, 2021	GAM	Completed coding work on Tyranny of the Pins.
0.10	November 15, 2021	GAM	Reorganized book as three parts. Finished GPIO pin examples. Continuing refinement of text language.
0.11	November 30, 2021	GAM	Added chapter for UART device.
0.12	December 26, 2021	GAM	Completed work on UART device chapter. Writing for the first part of the book is complete. Additional copy editing is now required.
1.0	March 31, 2022	GAM	First release of the book which contains only Part I.
2.0.0	June 21, 2024	GAM	Second major release. In this release, the Part II place holder has been removed and replaced by all new content. Part II introduces ReSEE (Reactive System Execution Environment) but is only half completed. It contains a complete description of the data management facilities of ReSEE, but no work has been done on the execution management aspects.
3.0.0	January 29, 2025	GAM	Third major release. In this release, Part I has been recoded in the Zig language. Part II is substantially unchanged and still contains only the data management facilities of ReSEE. The book is now built using the build facilities of Zig.

Revision	Date	Author	Comment
3.1.0	February 1, 2025	GAM	This release clarifies and corrects some of the language from 3.0.0 and adds the Cortex v7e-M core access module to the book.
4.0.0	April 15, 2025	GAM	Fourth major release. In this release, the chapter on Sequencing Execution is partially complete. State models for Relational Classes are described and implemented.
4.1.0	May 10, 2025	GAM	In this release, the chapter on Sequencing Execution is extended to add a section on the behavior of associations. The added behavior attaches a state model to an association so that it may be used in as a competitive association for resource allocation.
4.2.0	September 4, 2025	GAM	The primary purpose of this version is to rework code to conform to the Zig version 0.15.1 compiler release. There has been significant changes in the code that adapts the UART peripheral code to the I/O interface of the Zig standard library. In addition, there has been significant changes in the Relational Class, Association, and Generalization chapters. The chapter on Domains has been started and is currently under active development.
4.3.0	January 16, 2026	GAM	This revision converts the document formatting from asciidoc to asciidoctor . This version also builds on Zig version 0.16.0-dev.254+6dd0270a1.

Introduction

i

This book is a work in progress. This introductory chapter is incomplete. As of this version, Part I is substantially complete. Part II is under construction. It contains the design and code for the data management portions of the model execution environment and the state model code for relational classes. It lacks the state model implementation for assigner associations and for polymorphic events. Part III is empty. Part IV is a compendium of supporting software and included code is complete. However, more support code will be added.

This book contains a set of design concepts and their corresponding implementation for engineering software of reactive-oriented systems using semi-formal techniques^[1]. Starting from well-founded principles, the book describes a path for implementing complex software systems that interact with their environment while abiding by sound software engineering principles. Along that path, the following broad areas are explored:

1. Basic systems programming required to run any software and have that software interact with its environment.
2. A model of software execution which both orchestrates control flow and enforces data consistency.
3. A composition technique that supports building software components and assembling those components into larger software systems.

In this book, there are several recurring software engineering themes:

Separation of concerns

Understandability is a critical component of a sustainable software system. Our ability to conceive and understand complex systems is largely based on dividing the system into coherent subject parts. Those parts allow us to consider the aspects of the system in isolation. Implied in the separation is a technique to assemble the parts into a functioning system. This separation (and the subsequent reassembly) are necessary to overcome our human limitations of short term memory.

Constrained execution

There are many more things in a software system which can go wrong than can go right. The state space of even the smallest computing system is so overwhelming large that we must make efforts on several fronts to confine a program's execution and verify those constraints during execution in an attempt to minimize its failure modes.

Allocation of complexity

The idea to consider system complexity as consisting of *essential* complexity and *accidental* complexity was first proposed by [Brooks](#). Semi-formal models of the logic of the system are the means by which the two forms of complexity are separated. Models capture system requirements and represent the system in an implementation independent form which is free of accidental complexity. Those models are translated into system software yielding any accidental complexity associated with the implementation. The solution to a problem must precede the implementation of that solution. The fundamental difference between “what” a system does and “how” technology is brought to bear to implement the system is the basic division between the essential and the accidental. The view of complexity here is similar to that of [Moseley and Marks](#).

These goals present a large scope of effort which must be narrowed substantially to accomplish. We start by defining some fundamental terms.

We adopt the categorization of systems by [Halbwachs](#):

- a. Reactive — ``systems that continuously react to their environment at a speed determined by this environment.''
- b. Interactive — ``systems which continuously interact with their environment, but at their own rate.'
- c. Transformational — ``classical systems whose inputs are available at the beginning of the execution and which deliver their outputs when terminating.'

The focus of this book is *reactive* systems. Again, using Halbwachs definitions, these types of systems feature:

Concurrency

At the least, the concurrency between the system and its environment must be taken into account.

Time requirements

These requirements concern both their input rate and their input/output response time.

Determinism

The outputs of such a system are entirely determined by their input values and by the occurrence times of these inputs.

Reliability

It is commonplace to say that errors in reactive systems can have dramatic consequences, involving human lives and huge amount[sic] of money.

— Nicolas Halbwachs, Synchronous programming of reactive systems

Two terms, often applied to reactive systems, are avoided:

Embedded

The notion of an embedded system lies with how the system is perceived by its user. The promise of software is to be able to use multi-purpose hardware to create special-purpose machines. A system which can transform its function by executing a different program is perceived as computer-based. An embedded system performs a single specialized function despite using a computer as part of its implementation. A user may know that the operations of an embedded system are performed by a computer, but the expectation when using the system is that it is specialized for a single purpose. Common parlance associates embedded systems with being resource constrained. However, modern computing hardware advances blur the distinction of resource constraints and computers exists which span a wide range of resource availability to meet an equally wide range of system requirements. Because the term *embedded* does not shed much light on the manner in which the system performs, this book does not use the term.

Real-time

A system either does or does not have requirements which specify its response timing. Failure to meet specified timing constraints constitutes a system defect regardless of whether there is any external manifestation or consequence of the failure. The existence of time response requirements does not describe effectively the manner in which a system is designed to meet those requirements. There is no single design or programming approach that is necessitated by response time requirements. The terms *hard real time* or *soft real time* provide no additional clarity. It seems, colloquially, that *hard* time requirements are meant to be taken seriously, having possible grievous consequences if not met. There is sometimes an implication that the

timing requirements may pose difficulties for the implementation to achieve, conflating the ideas of strict and difficult. *Soft* timing requirements usually equate to, ``not too long as to annoy the human operating the system.'' If you are in the position to argue that a timing violation is of no substantial consequence, then you are also in the position to argue that the requirement is otiose. For timing violations that are detected as uncorrelated to specific controls of the system, conjectures regarding the frequency of occurrence of the violation (usually as an attempt to rationalize taking no corrective action) are a probabilistic argument that would only be valid if we dwelt in a quantum-sized world. We strive for determinism as a mechanism for meeting time response requirements.

Target Readers

This book is primarily intended for:

- Experienced developers looking for new system building techniques.
- Engineers seeking a method of building software which results in systems that are easier to understand and reason about.
- Software developers interested in software engineering concepts which are validated by an accompanying implementation of the concepts.

In the end, we must conclude this is *not* a beginners book. Albeit there is a large amount of *explanation* in the book, there is no deliberate attempt at being a tutorial. There are, for example, no exercises or other comprehension tests. The explanation is intended to describe the design concepts and the accompanying implementation demonstrates the concept in practice. Basic knowledge of programming is assumed. Basic knowledge of microcontrollers is helpful. There are many good sources of tutorial material on the Internet that are not replicated here. This is not to say that a beginner would accrue no benefit from reading the book. Rather beginners are advised that it may be necessary to search other sources for some background information to which they may not have been exposed previously.

Overview

This book is divided into five parts. The first three parts contain the main exposition of the subject matter. The remaining two parts cover common code used elsewhere and supplemental information for the benefit of readers.

- Part I, *Tap Root*, describes the core systems programming that is required to execute a program and have that program interact with its environment.
- Part II, *Baton*, describes the model of execution for system components that are decomposed by subject matter.
- Part III, *Islands in the Stream*, describes the connections between subject matter components from Part II and how those components are assembled into a complete system.
- Part IV, *Supporting Code*, contains chapters describing common code sequences used elsewhere in the book.
- Part V, *Supplemental Materials*, is a collection of terms, appendices and other information which is referenced elsewhere in the book.

What is the bottom?

The subtitle of this book is, *A Microcontroller from the Bottom Up*, so what constitutes “bottom”

must be defined. There are many deep wells of knowledge and expertise in computing electronics. It ranges from the physics of semiconductors, to the design of instruction sets, and to the amazingly complex machines used to manufacture semiconductors. Since our primary interest in this book is software and, standing firmly on the shoulders of others, the starting point is taken as a complete microcontroller hardware system, but without any software. This starting point is quite elevated in the computing food chain. There are many important aspects of how microcontroller hardware is made into working computer systems that are not considered. It is assumed that there is some technique to install code on the microcontroller and the software execution can be monitored and controlled.

The criteria for choosing a platform for demonstrating the implementation are:

- A relative modern computing architecture whose processor core supports some form of execution constraints.
- The availability of inexpensive hardware development boards.
- The availability of language tools, debuggers, and general development ecosystem.

There are many choices which meet these criteria. Components from [SparkFun®](#) have been chosen. The target implementation is the Artemis MicroMod processor board installed on a suitable carrier board. The processor is an [ARM® Cortex-M4](#) based SOC manufactured by [Ambiq®](#).

Some readers will unfortunately conclude that since the target for the implementation is a microcontroller and their interests are in larger, more capable machines, this book has little to offer them. Since the intent is to have a functioning body of software, there is substantial, unavoidable SOC specific implementation code in the book. The choice of using a microcontroller avoids many of the complexities of larger computing environments so that the focus can remain on the essentials of the engineering. However, most of the microcontroller specific details are contained in Part I and later parts of the book make little or no reference to target specific details. That is one of the first separations of concern made, namely, an adamant distinction between the application logic and the specific computing technology used to implement that logic. To that end, Part II gives an implementation of the execution scheme both for the Artemis processor board and the POSIX environment. As the title indicates, the book progresses from specific code to more abstract models of application logic in an effort to both scale development efforts to build complex, yet understandable, systems and to demonstrate clearly how those techniques are composed from fundamental constructs.

What is the top?

The goal is to be able to build complex software systems which are understandable and sustainable. That will be accomplished by decomposing the problem in to cohesive domains based on subject-matter. The domains are individually analyzed. The analysis is captured in a model of the subject matter of the domain. That model is then *translated* into the system implementation. The modeling artifacts provide two essential functions:

- a. A means to specify precisely the problem being solved. This implies that understanding about the system is *not* derived from the code which implements it, but rather by “executing” the model.
- b. A direct path from the model to an implementation. Model artifacts must contribute directly to deriving the system implementation.

A system is then composed of a set of highly cohesive domains interacting with each other and the environment by means of *bridges*. Bridging is the activity required to provide the mapping between the semantics of the various subject matter domains.

In the book [Models to Code](#), the approach started at modeling and demonstrated how code could be directly derived from the models. In this book, the approach is in the opposite direction to show how models help conquer problems in understanding and building complex systems.

There Will Be Design

Most of the content of this book is a series of design concepts for how to control aspects of a microcontroller and to create an execution environment for applications. In these cases, design concepts are presented along with their implementation.

In several places, we use the phrase, *element of the design*. This phrase is intended to convey that the subject being discussed requires creative human activity to construct the exact details of how the circumstances should be treated. For example, it is an *element of the design* to determine the exact split of processing that occurs between an IRQ handler and background processing to fulfill the requirements of the system. A software engineer must create a design to accomplish the task which gives the operational details of the split. These cases are such that there is no “checklist” or rote manner in which the activity may be accomplished and we rely upon the experience, intelligence, and creativity of the engineer.

There Will Be Code

And a lot of it. This book is also a [literate program](#). Code and description are combined in the document to facilitate understanding of the program logic. The source code is *tangled* (i.e. separated) from the document to present to the compiler.

There is a simple syntax for denoting program parts within the document and the details are presented in an [appendix](#). Literate programming allows a program to be presented to the reader in an order which is different than that demanded by the compiler. It allows the presentation of design concepts from top down, bottom up, or inside out as it appropriate to the subject matter.

The programming language used in the book is [Zig](#). Zig is a relatively new systems programming language which presents an execution model that has similar characteristics as “C” and can easily interoperate with “C”. For example, Zig programs are responsible for managing their own memory and can perform direct access to memory. But Zig is also a much bigger language than “C” and has several facilities, which are described later, that make it a suitable tool for the particulars of this type of work. For “C” programmers, the syntax is vaguely familiar, but it does *not* follow the “C” conventions where declarations of a variable look much the same as the usage of the variable in an expression. For readers unfamiliar with Zig, the source code can be skipped on the first reading, but better understanding of the programming intent is only available by learning some Zig language constructs.

How to Read this Document

Your author can assure you that this book was not written sequentially starting at page one. There was much skipping around to develop an organization for the presentation. There is no expectation that readers will start at page one and read the book sequentially. Skipping around is encouraged.

Part I of the book necessarily contains a substantial amount of processor specific systems programming and discussion of how system memory is handled. The *Crossing the Divide* chapter is central to Part I. This chapter defines the mechanisms by which the system interacts with the environment and how that interaction is brought into the system for further actions. Reading this chapter is central to understanding subsequent parts of the book.

Part II is devoted to the model of execution and the run time code used to implement it. The approach to the execution model is distinctive in that all decisions about how control is sequenced and synchronized and all policies regarding how data is managed are factored into the run time execution code. Application code has limited ways in which it can affect execution sequencing and manage its data and, consequently, makes no decisions as to which body of code is next executed. The execution model yields deterministic behavior that is only dependent upon the timing of changes detected in the system environment. Data is dealt with transactionally and its consistency is checked to insure the semantic rules of the application are maintained throughout a program's execution.

Part III shows how the independent components from Part II are formed into a complete system by bridging modeled components together. The ability to compose software pieces together is essential to building larger systems and Part III defines how those pieces are wired together to achieve the purpose of the application.

Move Along — Nothing New Here

It is important to emphasize that this book does not present any novel computer science concepts. The ideas in this book are an application of the fundamentals of Computer Science which have been around for decades. The more abstract concepts owe much to the work of Tom DeMarco, Stephen Mellor and Paul Ward, and, in particular, Sally Shlaer and Stephen Mellor. The only unique content in this book is the particular implementations and organizations that apply the concepts, which themselves were worked out long ago, to the engineering effort of building robust and sustainable systems of known functionality.

[1] We use the term *semi-formal* to avoid any implication that we intend to prove program correctness in a mathematical sense. We leave mathematical proofs of correctness to Computer Scientists. We are satisfied here to be better Software Engineers.

Tap Root

Abstract

This part of the book shows both the core systems programming necessary to get any program to run on the target and the fundamental mechanism whereby changes in the environment can be made known to the application logic. These ideas are related in the sense that substantial systems programming is required to set up our intended execution environment. That environment separates privileged from unprivileged execution and uses the Memory Protection Unit to enforce the separation of the two modes of execution. Separating execution privilege requires mechanisms to allow the two execution types to interact in a highly controlled fashion. That interaction is governed by the processor architecture and requires yet more systems programming to implement. The scheme for interacting with the environment includes both mechanisms to control peripheral devices and an implementation pattern for device interactions that provides for interrupt level processing to inform unprivileged execution of significant happenings in the environment. This mechanism is fundamental to the design scheme for handling the concurrency implicit in the environment and is accomplished using processor interrupts to preempt unprivileged execution. The manner in which the preemption is scheduled is entirely constructed from the processor's priority scheme used for exception handling. A critical part of the scheme allows interrupt handling functions to post continuation requests for additional processing to be performed by unprivileged code.

The Path to main

This chapter shows the code necessary to bring the processor to a state where it can execute the `main()` function. There are many ways to accomplish start up. Most vendors or toolchain providers supply sufficient start up code to build a program. Usually, vendor supplied code is the bare minimum required to compile and link a program. You will want to do substantially better.

Getting in on the Ground Floor

For the types of operations we consider in this chapter and in the rest of this part of the book, we make extensive use various hardware controls provided either by the processor core or the peripherals connected to the core. That control is realized via memory-mapped I/O. Certain selected locations in the memory map provide an encoding to the capabilities of the underlying hardware. We need a consistent manner to name and access them. In the ARM ecosystem, [CMSIS](#) (Common Microcontroller Software Interface Standard) is standard way in which the large amount of interfacing information, in the form of symbolic names, is provided to software developers. CMSIS has many parts, but we are most interested in that part which describes the processor core and the Apollo 3 peripherals. This is termed "CMSIS-core" and is the only part of CMSIS we consider. The processor part is standard across the various ARM architectures. The peripheral information is provided by vendors, in this case Ambiq. However, CMSIS, is coded in "C", usually in the form of pre-processor macros, and in that form is not particularly useful in a Zig environment.

For our purposes, we follow closely the CMSIS way of doing things with several notable differences:

1. Access to the ARM core, namely the v7e-M architectural components is kept separate from any Apollo 3 specific peripherals. This matters most when dealing with system exceptions verses peripheral interrupt exceptions.
2. The Apollo 3 peripherals are kept in a separate module and only those peripherals that are used in the book have been recoded into Zig.
3. Both of the above pieces are coded in terms of a generalized memory-mapped I/O module that gives more type specific access to the underlying hardware control

One significant difference in this approach is to favor Zig naming conventions rather than the "C" pre-processor conventions used by CMSIS. We strive to keep the letters used in the names the same, but capitalization and the occasional inserting of underscores as word separators is more in keeping with Zig conventions. The attempt is to make CMSIS names easily recognizable as Zig names, even if there is a difference in capitalization. This approach effectively eliminates the *wall of capital letters* and large number of bit-wise operations found in "C" implementations that use CMSIS.

There are three modules provided that form the basis for the core and peripheral control.

1. A `mmio` module provides the Zig basis for mapping core and peripheral facilities directly onto memory access.
2. A `v7m` module provides access to the controls for the core itself.
3. An `apollo3` module provides access to the controls for the peripherals of the Apollo 3 SOC. Each peripheral is given its own namespace for its registers.

The description of these modules are provided in Part IV of this book.

Executing from power up

Getting a computer to do useful work from power up is a surprising difficult undertaking. Desktop computers running an operating system go through many phases of initialization just to get to the point where you can run any program at all. When writing programs for an established operating system, the tedious aspects of initializing the system to run a program have already been done for you. The language compilers and other tools have already been configured to produce an executable that the operating system can [load and run](#). After all, easily running a set of application programs is an essential aspect of an operating system.

A microcontroller usually only runs one program at a time. Coming from the bottom up, you are faced with the task of determining out how to produce an executable which is acceptable to the processor. Starting up a microcontroller is simpler, but requires detailed knowledge in four particular areas. This is an unfortunately common situation of needing to know a lot in order to do anything at all.

1. The processor architecture insists upon a certain arrangement of data values in memory at start up time and has a precise mechanism it uses to begin execution.
2. The compiler must be directed to generate the required code and data, organized in the manner the processor requires. The start up code could be written in assembly language where complete control is possible. It is more convenient to have the source code in one language, even if that means resorting to *inline assembly* language on occasions.
3. The linker must be directed to locate the startup code and data at the place in memory where the processor expects to find it. The linker must also be directed where to place other code and data that has not yet been written.
4. The system specific initialization must be done. This usually involves at least setting up clock sources and frequencies.

For ARM Cortex-M4 processors, power up is one of the conditions where the Reset exception is executed and the mechanism for executing any exception starts with the exception vector table^[1]. When the processor powers up, it initializes the value of the stack pointer to the first element of the exception vector table. The processor then loads the program counter with the value of the second element. The second element in the vector table is the pointer to the Reset exception handler. The internal registers and state of the processor are set to known values. For an SOC, there can be several forms of reset. Some peripheral device registers are initialized to reset values while others may preserve their values when the reset does not originate from a power on reset. Once the reset state is established, the microcontroller does what all computers do — fetch the next instruction, as given by the Program Counter, and execute it.

Graphically, a generic Cortex-M4 vector table appears as follows. ^[2]

Exception number	IRQ number	Offset	Vector
16+n	n	0x0040+4n	IRQn
.	.	.	.
.	.	.	.
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			
9			Reserved
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

Figure 1. Cortex-M Vector Table

The vector table is specialized by the Apollo 3 SOC and appears as follows. [3]

Exception Number	IRQ Number	Offset	Vector	Peripheral/Description
255	239	0x03FC	IRQ239	
.	.		.	
.	.	0x00C0	IRQ31	Clock Control
.	.	0x00BC	IRQ23-30	Stimer Compare[0:7]
.	.	0x009C	IRQ22	Stimer Capture/Overflow
.	.	0x0098	IRQ21	SW INT
.	.	0x0094	IRQ20	MSPI
.	.	0x0090	IRQ19	PDM
.	.	0x008C	IRQ18	ADC
.	.	0x0088	IRQ17	SCARD
.	.	0x0084	IRQ16	UART1
.	.	0x0080	IRQ15	UART0
.	.	0x007C	IRQ14	Counter/Timers
.	.	0x0078	IRQ13	GPIO
.	.	0x0074	IRQ12	BLE
.	.	0x0070	IRQ11	I ² C/SPI Master 5
.	.	0x006C	IRQ10	I ² C/SPI Master 4
.	.	0x0068	IRQ9	I ² C/SPI Master 3
.	.	0x0064	IRQ8	I ² C/SPI Master 2
.	.	0x0060	IRQ7	I ² C/SPI Master 1
.	.	0x005C	IRQ6	I ² C/SPI Master 0
.	.	0x0058	IRQ5	I ² C/SPI Slave Register Access
.	.	0x0054	IRQ4	I ² C/SPI Slave
.	.	0x0050	IRQ3	Voltage Comparator
18	2	0x004C	IRQ2	RTC
17	1	0x0048	IRQ1	Watchdog Timer
16	0	0x0044	IRQ0	Brownout Detection
15	-1	0x0040	Systick	
14	-2	0x003C	PendSV	
13		0x0038	Reserved	
12			Reserved for Debug	
11	-5	0x002C	SVCall	
10			Reserved	
9				
8				
7				
6	-10	0x0018	Usage Fault	
5	-11	0x0014	Bus Fault	
4	-12	0x0010	Memory management Fault	
3	-13	0x000C	Hard fault	
2	-14	0x0008	NMI	
1		0x0004	Reset	Unused
		0x0000	Initial SP value	

Figure 2. Ambiq Apollo 3 Vector Table

Note that the first 16 exception vectors are the same as required by the Cortex-M4 architecture. The remaining 32 exception vectors are dedicated to IRQ handlers that are specific to the peripherals of the Apollo 3.



As shown in the previous figure, ARM makes a distinction between the *Exception number* and the *IRQ number*. Exception numbers have a direct association to elements of the vector table. An IRQ number can also be used as an index into the vector table in some contexts. We will keep the two numbering schemes separate. This is different from the CMSIS approach which defines a platform specific enumeration for all the IRQ numbers and deals exclusively with IRQ numbers. For now, just remember the distinction between system exceptions and interrupt exceptions, endeavoring to keep the terms straight. System exceptions are defined by the core architecture. Interrupt exceptions are defined by the microcontroller SOC.

Another goal is to create a means of start up that can be used repeatedly when building programs for the device. Even if usually only running one program at a time is running, it is necessary to build conveniently multiple programs.

I consider building an application as building and running a long series of test applications — the last one of which is pronounced, “finished”^[4].

Vector table

Except for the first element being a pointer to memory suitable for holding the stack (e.g. RAM memory), the exception vector table is just an array of pointers to functions, i.e. a jump table. Computer memory doesn't have any associated type — data types are a programming language construct.

There is one further complication which is sometimes overlooked and can cause confusion. It has to do with how the processor encodes instruction set selection in a function pointer. ARM processors have had a 32-bit ARM instruction set and sometimes a more densely encoded 16-bit instruction set. The 16-bit instruction set is called, *Thumb*, and comes in two variations, Thumb-1 and Thumb-2^[5]. Cortex-M4 processors only execute the Thumb-2 instruction set. ARM processors that execute the 32-bit instruction set can switch instruction encodings between the 32-bit instructions and the 16-bit instructions. The difference is encoded in the least significant bit (LSB) of the function pointer value. This bit is not used for actually addressing an instruction since instructions must be aligned to even addresses. To detect if 32-bit instructions got mixed into a Thumb-2 processor, the function pointers in the exception vector table all have the LSB set. A raw dump of memory would show the pointer addresses as all odd numbered, even though the instructions for a functions definitely do not start on an odd address. In certain situations such as this one, bits are precious and are used to overload the meaning of a memory address.

The goal is to arrange a sequence of values where each value is 32-bits and must be specified as the appropriate function prototype. The type of the vector table function pointers is defined in the v7m module.

```
<<startup_apollo3: imports>>=
const v7m = @import("v7m");
const apollo3 = @import("apollo3");
```

The v7m module is a Zig module that follows the pattern of CMSIS, the standard ARM supplied "C"

header files that encodes the specifics of the core symbolically. This design supplements the CMSIS core definitions with functions that manipulate the controls bits of the system registers. Named functions are used to manipulate hardware registers.

The Zig declaration (taken from the v7m module) for an exception handler is:

```
pub const ExceptionHandler = *const fn () callconv(.c) void;
```

In words, an exception handler type is a pointer to a function which takes no arguments and returns no value. That corresponds to the nature of exception level execution. Exceptions arise from either the internal operations of the processor or by signals from outside the processor, temporarily suspend computation to run the exception handler, and returning to the interrupted location to resume the suspended computation. There really is no way to pass an argument and no fixed call site to which a returned value^[6] can be delivered. That doesn't mean that exception handlers can have no effect on the system. Since exception handlers are just running code, they can read and write memory and that is all the mechanism needed to have an exception handler affect subsequent processing. Note, we request the calling convention to abide by the Arm standard.

We tell the compiler that an exception handler function uses the same calling conventions as a "C" function (using `callconv(.c)`). Many processor architectures have specific requirements as to how exception handling code must interface to the system and compilers often support annotation of exception handlers so they may generate the correct code. On the Cortex-M4 architecture, the hardware arranges the entry to an exception handler according to the [Procedure Call Standard \(AAPCS\)](#), which is defined for the processor architecture. This ABI is the same as used for ordinary "C" functions, which the Zig compiler can produce. The net effect is that exception handlers look just like ordinary functions at the binary level and there is no need for special annotations that indicate the function is used specifically for handling an exception. Our annotation of the exception handler is simply to request the Zig compiler to generate function prologue and epilogue code in accordance with the "C" language.

The processor architecture requires data values in the layout shown in the previous figure. That layout of memory is an array which holds function pointers to our exception handlers. There are two broad types of exception handlers:

1. Those defined by the processor architecture. These are defined by ARM and are the same for all Cortex-M4 cores.
2. Those associated with peripheral interrupts. These are defined by the chip manufacturer and differ from chip to chip.

In the first case, there are 16 such system exceptions.

```
<<startup_apollo3: declarations>>=
const system_exception_count = v7m.system_exception_count;
```

In the second case, the Apollo 3 SOC defines 32 sources of interrupts.

```
<<startup_apollo3: declarations>>=
const interrupt_count = apollo3.irq_count;
```

The exception vector table is defined as:

```
<<startup_apollo3: declarations>>=
```

```

/// The exception vector table. Note the array is constant and intended
/// to be placed in ROM. Also note the array is placed in the ".vectors"
/// linker section so that it may be easily located in the correct place
/// in memory.
export const __vectors linksection(".vectors") =
    [system_exception_count + interrupt_count]v7m.ExceptionHandler{
    <<startup_apollo3: processor exceptions>>
    <<startup_apollo3: peripheral interrupts>>
}; ①

```

① There's a lot of information in this definition:

- The exception vector table is in reality an initialized array variable named `__vectors`. Since the variable will be a global symbol, a name is chosen that is unlikely to collide with other symbols.
- The vector table size is defined to fit only the exceptions defined by the processor (16) and those defined by the Apollo 3 as interrupts (32). This limit needs to be in place since the Apollo 3 defines a patch area for the Bluetooth controller which is expected to follow immediately after the IRQ handlers.
- The variable has a `const` qualifier indicating writes to the variable are not allowed. The vector table is placed in read only memory.
- Finally, the compiler is directed to place the vector table in a specific linker section. This linker section name is used later to place the exception vector table in memory at a location expected by the processor.

Before getting to the exception handling functions, you need to decide what to do about a stack. Having a stack is essential to calling functions since it holds parameter and return information. The stack must be placed in read/write memory. That is almost always supplied by RAM which is part of the SOC. It is possible and sometimes desirable to have several stacks and the Cortex-M architecture defines multiple stack pointer registers. Before considering those more complex scenarios, start with the simple case of providing a Main Stack Pointer (MSP). A MSP must be provided since it is the first element of the exception vector table.

So the first entry in the vector table is the value of the MSP that is loaded by the processor core upon reset.

```

<<startup_apollo3: processor exceptions>>=
@ptrCast(main_stack_top), ①

```

① The cast is necessary to inform the compiler that the address of the stack is to be typed the same as the other elements of the array. Alternatively, you could have defined the vector table to be a structure whose first element is a data pointer (rather than a function pointer), but that is an additional complication avoided with the cast. The start up code doesn't access the vector table so being that strictly typed doesn't grant us much advantage.

So where did the symbol `main_stack_top` come from and why is its value a pointer? The system stack is discussed [below](#) when memory initialization is discussed. The following code imports the linker generated symbol, `main_stack_top` to be a local constant named, `main_stack_top`. For now, the `main_stack_top` symbol is created by the linker and has been given a distinctive name that is unlikely to collide with any other names in the program (the symbol name is globally known). Later, we see the linker script that creates the symbol. The symbol value is the address of the top of the stack. Since the stack grows downward in memory addresses (known as a *full descending stack*), the initial value is set to the largest memory address allocated to the stack^[7].

```
<<startup_apollo3: declarations>>=
const main_stack_top =
    @extern(*u64, {.name = "__main_stack_top__", .linkage = .strong}); ●
```

① Note in keeping with the requirements of the core architecture, stack space is 64-bit aligned.

The exception handler names for the system exceptions defined by the processor architecture are standardized by CMSIS. The following system exception handlers use CMSIS names but are stated using Zig naming conventions.

```
<<startup_apollo3: processor exceptions>>=
                                // Exception number
resetHandler,                  // 1
nmiHandler,                    // 2  Unused on the Apollo 3.
hardFaultHandler,              // 3
memManageHandler,              // 4
busFaultHandler,               // 5
usageFaultHandler,             // 6
defaultHandler,                // 7  Exceptions number 7 - 10 are reserved.
defaultHandler,                // 8
defaultHandler,                // 9
defaultHandler,                // 10
svcHandler,                   // 11
debugMonHandler,              // 12
defaultHandler,                // 13  Reserved
pendSvHandler,                // 14
sysTickHandler,               // 15
```

The usual naming convention for SOC specific interrupts is to use the interrupt name given in the SVD file and append `_IrqHandler`. Note that these functions follow Zig naming conventions rather than what you might see in a "C" based implementation.

```
<<startup_apollo3: peripheral interrupts>>=
                                // IRQ number
brownOutIrqHandler,           // 0
wdtIrqHandler,                // 1
rtcIrqHandler,                // 2
vCompIrqHandler,              // 3
ioSlaveIrqHandler,            // 4
ioSlaveAccIrqHandler,         // 5
ioMstr0IrqHandler,            // 6
ioMstr1IrqHandler,            // 7
ioMstr2IrqHandler,            // 8
ioMstr3IrqHandler,            // 9
ioMstr4IrqHandler,            // 10
ioMstr5IrqHandler,            // 11
bleIrqHandler,                // 12
gpioIrqHandler,               // 13
ctimerIrqHandler,              // 14
uart0IrqHandler,               // 15
uart1IrqHandler,               // 16
```

```

scardIrqHandler,           // 17
adcIrqHandler,             // 18
pdmIrqHandler,             // 19
mspi0IrqHandler,           // 20
software0IrqHandler,       // 21
stimerIrqHandler,          // 22
stimerCmpr0IrqHandler,     // 23
stimerCmpr1IrqHandler,     // 24
stimerCmpr2IrqHandler,     // 25
stimerCmpr3IrqHandler,     // 26
stimerCmpr4IrqHandler,     // 27
stimerCmpr5IrqHandler,     // 28
stimerCmpr6IrqHandler,     // 29
stimerCmpr7IrqHandler,     // 30
clkGenIrqHandler,           // 31

```

①

- ① The Apollo 3 chip uses a patch table that follows immediately after the vector table for 16 entries. There is little documentation of this other than its appearance in their start up files and it is apparently used in conjunction with the Bluetooth Low Energy (BLE) module.

Exception vector functions

At this point, an array variable has been defined that is to be the exception vector table and names given to the functions that form the initializer for the array. But now you need to make an important decision about how to supply the code for the exception handling functions.

The simple approach is to open up a file and start typing in the functions whose names are in the vector table array. That approach works fine, but if you think a little further ahead, you realize that it is not very flexible. At this point in time, you don't know what code to put into the function. You could stub out the functions, but then every time you did decide what an exception handler should do, you would have to edit that file. Every different program would have its own file with the exception handling function in place. You would like to use this arrangement of system level code for any program intended for the Apollo 3. For that to happen, you must be able to do two things.

1. Supply your own function without having to edit any system level files. You would like simply to define a function whose name is the same as an exception handler, e.g. `gpioIrqHandler`, and have that function used in the vector table.
2. Supply a default handler function for any exceptions not used in a program.

Needless to say, this problem has been solved and it is accomplished by giving information to the linker about how to resolve symbolic information.

In the first case, the function is declared to have *weak* linkage. Normal function definitions are considered by the linker as *strong* and attempting to link multiple object files which have strong symbols with the same name is an error. The linker would have no basis for choosing the correct one. However, a weak symbol tells the linker to use a strong symbol with the same name if one is available and otherwise use the weak symbol. Declaring a function as weak achieves the goal of being able to override easily a default implementation of an exception handler.

The second case is handled by simply creating weak functions that invoke the `defaultHandler`.

Not every exception handler needs the flexibility to be substituted. The `defaultHandler` is the default function for missing handlers and must be defined in this file. The `resetHandler` is can be

defined so that it does the correct operations irrespective of the specifics of an application. Swapping out what to do at reset is not common and if necessary requires changes to the `resetHandler` definition.

Neither the `defaultHandler` nor the `resetHandler` ever return. They represent the extremes of the processing spectrum. The `resetHandler` is entered upon power up or reset, transfers control to our application program, and there is no call site to which to return. The `defaultHandler` is entered if an exception is taken and no appropriate handler has been supplied, indicating something is terribly wrong, and it's not possible to go forward with any further program execution. Both functions are shown below.

The following is the resulting long sequence of handler function declarations. Note that the Zig compiler is requested to generate a "C" ABI function, in keeping with the requirements of the processor core.

```
<<startup_apollo3: functions>>=
fn nmiHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn hardFaultHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn memManageHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn busFaultHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn usageFaultHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn svcHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn debugMonHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn pendSvHandler() callconv(.c) void {
    asm volatile(
```

```
        "b defaultHandler"
    );
}
fn sysTickHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn brownOutIrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn wdtIrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn rtcIrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn vCompIrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn ioSlaveIrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn ioSlaveAccIrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn ioMstr0IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn ioMstr1IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
fn ioMstr2IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}
```

```
}

fn ioMstr3IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}

fn ioMstr4IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}

fn ioMstr5IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}

fn bleIrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}

fn gpioIrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}

fn ctimerIrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}

fn uart0IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}

fn uart1IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}

fn scardIrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}

fn adcIrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}

fn pdmIrqHandler() callconv(.c) void {
```

```
asm volatile(
    "b defaultHandler"
);
}

fn mspi0IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
);
}

fn software0IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
);
}

fn stimerIrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
);
}

fn stimerCmpr0IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
);
}

fn stimerCmpr1IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
);
}

fn stimerCmpr2IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
);
}

fn stimerCmpr3IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
);
}

fn stimerCmpr4IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
);
}

fn stimerCmpr5IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
);
}

fn stimerCmpr6IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
);
}
```

```

);
}

fn stimerCmpr7IrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}

fn clkGenIrqHandler() callconv(.c) void {
    asm volatile(
        "b defaultHandler"
    );
}

```

The following is an equally long sequence of declarations that request the Zig compiler to publish the exception handler function names as "`"weak"`" so they may be overridden as the handlers become available.

```

<<startup_apollo3: functions>>=
comptime {
    @export(&nmiHandler, {.name = "nmiHandler", .linkage = .weak });
    @export(&hardFaultHandler, {.name = "hardFaultHandler", .linkage = .weak });
    @export(&memManageHandler, {.name = "memManageHandler", .linkage = .weak });
    @export(&busFaultHandler, {.name = "busFaultHandler", .linkage = .weak });
    @export(&usageFaultHandler, {.name = "usageFaultHandler", .linkage = .weak });
    @export(&svcHandler, {.name = "svcHandler", .linkage = .weak });
    @export(&debugMonHandler, {.name = "debugMonHandler", .linkage = .weak });
    @export(&pendSvHandler, {.name = "pendSvHandler", .linkage = .weak });
    @export(&sysTickHandler, {.name = "sysTickHandler", .linkage = .weak });
    @export(&brownOutIrqHandler, {.name = "brownOutIrqHandler", .linkage = .weak });
    @export(&wdtIrqHandler, {.name = "wdtIrqHandler", .linkage = .weak });
    @export(&rtcIrqHandler, {.name = "rtcIrqHandler", .linkage = .weak });
    @export(&vCompIrqHandler, {.name = "vCompIrqHandler", .linkage = .weak });
    @export(&ioSlaveIrqHandler, {.name = "ioSlaveIrqHandler", .linkage = .weak });
    @export(&ioSlaveAccIrqHandler, {.name = "ioSlaveAccIrqHandler", .linkage = .weak });
    @export(&ioMstr0IrqHandler, {.name = "ioMstr0IrqHandler", .linkage = .weak });
    @export(&ioMstr1IrqHandler, {.name = "ioMstr1IrqHandler", .linkage = .weak });
    @export(&ioMstr2IrqHandler, {.name = "ioMstr2IrqHandler", .linkage = .weak });
    @export(&ioMstr3IrqHandler, {.name = "ioMstr3IrqHandler", .linkage = .weak });
    @export(&ioMstr4IrqHandler, {.name = "ioMstr4IrqHandler", .linkage = .weak });
    @export(&ioMstr5IrqHandler, {.name = "ioMstr5IrqHandler", .linkage = .weak });
    @export(&bleIrqHandler, {.name = "bleIrqHandler", .linkage = .weak });
    @export(&gpioIrqHandler, {.name = "gpioIrqHandler", .linkage = .weak });
    @export(&ctimerIrqHandler, {.name = "ctimerIrqHandler", .linkage = .weak });
    @export(&uart0IrqHandler, {.name = "uart0IrqHandler", .linkage = .weak });
    @export(&uart1IrqHandler, {.name = "uart1IrqHandler", .linkage = .weak });
    @export(&scardIrqHandler, {.name = "scardIrqHandler", .linkage = .weak });
    @export(&adcIrqHandler, {.name = "adcIrqHandler", .linkage = .weak });
    @export(&pdmIrqHandler, {.name = "pdmIrqHandler", .linkage = .weak });
    @export(&mspi0IrqHandler, {.name = "mspi0IrqHandler", .linkage = .weak });
    @export(&software0IrqHandler, {.name = "software0IrqHandler", .linkage = .weak });
    @export(&stimerIrqHandler, {.name = "stimerIrqHandler", .linkage = .weak });
    @export(&stimerCmpr0IrqHandler,

```

```

    .{ .name = "stimerCmpr0IrqHandler", .linkage = .weak });
@export(&stimerCmpr1IrqHandler,
    .{ .name = "stimerCmpr1IrqHandler", .linkage = .weak });
@export(&stimerCmpr2IrqHandler,
    .{ .name = "stimerCmpr2IrqHandler", .linkage = .weak });
@export(&stimerCmpr3IrqHandler,
    .{ .name = "stimerCmpr3IrqHandler", .linkage = .weak });
@export(&stimerCmpr4IrqHandler,
    .{ .name = "stimerCmpr4IrqHandler", .linkage = .weak });
@export(&stimerCmpr5IrqHandler,
    .{ .name = "stimerCmpr5IrqHandler", .linkage = .weak });
@export(&stimerCmpr6IrqHandler,
    .{ .name = "stimerCmpr6IrqHandler", .linkage = .weak });
@export(&stimerCmpr7IrqHandler,
    .{ .name = "stimerCmpr7IrqHandler", .linkage = .weak });
@export(&clkGenIrqHandler, .{ .name = "clkGenIrqHandler", .linkage = .weak });
}

```

Hitting Rock Bottom

It may seem strange to consider at this time what happens when the software gets to a spot where it does not know how to proceed. This is the classic *this should never happen* case, which during development, all too frequently does actually happen. It's desirable to have a single function to call when the software doesn't know what else to do. That function must be used judiciously, but is a key part of always maintaining control of the processor execution. The name chosen for the function, which is derived from a venerable, historic name, is `systemAbEnd()`.

So what is the last act of a completely confounded microcontroller? Reset. Strange as it may sound, sometimes resetting the processor and starting all over again is the only recovery available. But, if a debugger is connected, popping back to the debugger is a great help to the developer. Fortunately, the Cortex-M architecture has the capability to determine if a debugger is enabled and an instruction to break out to it.

```

<<system_apollo3: systemAbEnd>>=
export fn systemAbEnd() noreturn {
    if (v7m.dcb.dhcsr.readField(.c_debug_en) == 1 or
        v7m.dcb.demcr.readField(.mon_en) == 1) { ❶
        @breakpoint();
    }

    v7m.scb.systemReset() ; ❷
}

```

- ❶ The test of whether Halting debug or the DebugMon exception are enabled before executing a BKPT instruction may seem superfluous. The object of the test is to prevent causing a Hard Fault while trying to terminate. If debugging is enabled or the DebugMon exception is enabled then the BKPT instruction would cause a *debugging event* to be generated. However, if neither of these conditions is enabled, as might be the case in a deployed system, the debugging event would cause the generation of a Hard Fault. You want to be able to invoke `systemAbEnd()` even in the Hard Fault handler. Creating a fault in the Hard Fault handler causes lockup, a situation to avoid since it complicates the diagnosis of any problem where the running program itself causes

the lockup.

- ② This is a core function which resets the processor through the System Control Block (SCB). The implementation ensures the function never returns.

Default Exception Handler

Most vendor supplied default exception handlers just go into a tight, infinite loop. In that case, when things go terribly wrong the system hangs. If you happen to have a watchdog timer going, it would probably reset the system. There are many options for what to do in the default handler. For a more robust system which supports easier diagnosis of fault problems, you should leave “bread crumbs”, i.e. copies of important status registers that determine what went wrong. Of course, if you leave a trail, you also must provide a means to examine what was left to diagnose the problem. Handling the boundaries conditions of a system going up or down, e.g. when its battery runs out of energy, is a broad topic deferred until later.

Starting at the bottom, it is difficult to know exactly what must be done for the exception handling. A means of adding in better exception handling later is needed. That means gathering information when the exception happens so that it may be used to diagnose the exception later. Similar to what was done with the default exception handler itself, more substantial processing can be provided by overriding functions. In particular two features are needed:

1. Gather the exception information required for better handling. Namely, access to the stack frame created when the exception occurred, the exception return information, and the exception number which is running are needed.
2. Forward the handling of exceptions whose handlers are missing to a function that can be overridden for the specifics of your particular system. Missing exception handlers are usually an indication of a problem building the program where the intended handler does not get included to override the weak default handler.

Because in this case a generic solution is desired, some assembly language is required.. The majority of Cortex-M programs do not require any assembly language. The design of the processor and the AAPCS are coordinated to remove the need for *glue* assembly language just to write an exception handler. But inevitably, systems programming near the bottom requires a small amount of assembly to access system specific information which can be used by compiled code for analysis.

For readers unfamiliar with ARM Thumb-2 assembly language, the techniques used to accomplish the above goals can be safely skipped. For others, this is the time to roll up your sleeves and dive in.

The design goal is to make the `defaultHandler` a wrapper around a general function for missing exception handlers. The `defaultHandler` gathers the necessary information and invokes `missingHandler()`. The definition of `missingHandler()` is given [below](#).

Three arguments are provided to `missingHandler()`.

1. A pointer to the exception stack frame.
2. The exception return value.
3. The value of the IPSR status register which holds the exception number of the currently executing exception.

To adhere to the AAPCS when invoking the missing exception handler, `exc_frame` must be placed in **R0**, `exc_return` in **R1**, and `ipsr` in **R2**. With the arguments in place, the control flow is a normal subroutine call to `missingHandler()` followed by an exception return. The exception return prompts the processor to restore the state information stacked during exception entry to resume processing

where the exception took place. Note in this case, there is no exception return. The code for `missingHandler()` invokes `systemAbEnd()`.

There is one other consideration for the missing handler. The processor builds an exception frame on whichever stack is in use when the exception occurs. The Cortex-M4 supports two stacks and we will both of them. The exception handling code itself always uses the MSP, but if the PSP was in use when the exception occurred, then it is used for the exception stack frame. Fortunately, bit 2 of the exception return value, which is in the link register, tells which stack pointer was in use when the exception happened and, consequently, which stack pointer contains the exception stack frame.

The following assembly language implements the design. Detailed code commentary follows.

```
1 <<startup_apollo3: functions>>=
2 /// The `defaultHandler` function is a wrapper to call the `missingHandler`
3 /// function. It performs the necessary work to determine on which stack
4 /// the exception frame is located and gathers other arguments for the
5 /// `missingHandler` function.
6 fn defaultHandler() callconv(.c) void {
7     asm volatile (
8         \tst    lr,#4
9         \ite    eq
10        \mrseq r0,msp
11        \mrsne r0,psp
12        \mov    r1,lr
13        \mrs    r2,ipsr
14        \push   {lr}
15        \bl    missingHandler
16        \pop    {pc}
17    );
18 }
19
20 comptime {
21     @export(&defaultHandler, .{ .name = "defaultHandler", .linkage = .weak });
22 }
```

Line 6

The `callconv(.c)` qualifier requests the compiler to emit a function with a "C" ABI in keeping with the way we have treated other exception handlers.

Line 8

The test of bit 2 in the link register determines which stack was in use when the exception occurred. The exception frame data is pushed onto whichever stack was in use when the exception occurred. If the bit is clear, then the MSP was used to stack the exception state. If the bit is set, then the PSP was used for the exception state information.

Lines 9-11

This is an example of conditional instruction execution that is part of the ARM architecture. Only one of the two instructions on lines 6 and 7 actually execute. The other instruction does not execute because its condition code does not match the condition of the `ite` instruction. This saves explicit branching when the number of instructions around the branch location is small.

Line 12

The link register holds the value of exception return. It is copied to `R1` to become the second

argument.

Line 13

The IPSR is one view of the extended processor status register and the MRS instruction is used to access it. The ISPR value is placed in **R2** to become the third argument.

Line 14

The value of the link register must be saved in preparation for invoking a function.

Line 15

Invoking the function is accomplished by executing a BL instruction which stores the return address (*i.e.* the value of the PC after the BL instruction) into the link register and jumps to the beginning of the function.

Line 16

Exception return is accomplished by popping the value of the EXC_RETURN (which is sitting on the stack where it was saved in line 14) directly into the program counter.

The subject of handling exception is further discussed in [a subsequent chapter](#) of this book.

Reset Exception Handler

Power up and reset conditions result in a Reset exception and vector table directs the processor to execute the Reset exception handler.

The goal here is to prepare the system to execute a program beginning at `main()`. This involves the following tasks:

1. Get the hardware into the state required execute a program.
2. Initialize read/write memory to the way the Zig language requires and that supports our running program.
3. Initialize the exception priorities.
4. Initialize the memory protection.
5. Switch to the Process Stack and disable privileged execution.
6. Transfer control to `main()`.

The `resetHandler` is implemented by invoking two weakly-linked functions. Following the CMSIS pattern, the first function performs system initialization and the second one enters `main`. Both these functions are defined later and given weak linkage so they may replaced by custom solutions.

```
<<startup_apollo3: declarations>>=
extern fn systemInit() void;
extern fn startMain() noreturn;
```

```
<<startup_apollo3: functions>>=
/// The `resetHandler` function is executed in response to a reset condition.
/// There are many such reset conditions, power up being the simplest.
export fn resetHandler() noreturn {
    v7m.msp.set(@intFromPtr(main_stack_top)); ①
    v7m.primask.set(0);
```

```

    systemInit() ;
    // Before starting main, switch to the Process Stack. This avoids
    // any problem with the function prologue of `startMain`.
    v7m.control.update(.{
        .spsel = .psp,
    });
    startMain() ;
}

```

- ① If we enter the reset handler from a bootloader, the MSP may not be set properly. This in fact happens when the code is loaded by the Sparkfun Variable Loader (SVL) program. Before the bootloader transfers control to the downloaded application, it does **not** set the MSP to the value from the exception vector table. The bootloader also leaves the interrupts disabled, causing a system fault to happen upon the first SVC call.

Startup Code Layout

The code for start up is placed in the `startup_apollo3.zig` file. This section shows the order of the literate program chunks that are put together to create the source file that is given to the compiler.

```

<<startup_apollo3.zig>>=
///! The `startup_apollo3.zig` file contains the code necessary to
///! manage the process of starting an Apollo3 microcontroller
///! from a power up or other reset condition.
///! It consists of the exception vector table with necessary defaults
///! and the `resetHandler`.
<<edit warning>>
<<copyright info>>

// Imports
<<startup_apollo3: imports>>

// Declarations
<<startup_apollo3: declarations>>

// Functions
<<startup_apollo3: functions>>

```

Core Initialization

In the previous section, the focus was on laying out the exception vector table and supplying code for the `resetHandler`. In this section, the focus shifts to getting the processor into the state needed to execute a program. The intent is to initialize only those aspects of the hardware which are essential setting up the processor core for any programs we intend to run. Other hardware initialization, *i.e.* most of the peripherals, is deferred until after `main` is reached. This code is intended to be generally applicable to the class of applications expected to be deployed and at start up exactly what other hardware might be used by a program is not known.

CMSIS conventions are to supply two files that handle SOC specific code sequences.

1. `startup_<device>.c` is meant to do just what it says, start up the computer. This is the file created in the previous chapter. We supply this file as `startup_apollo3.zig`.
2. `system_<device>.c` is a file which contains implementations of functions that specific to the SOC. At a minimum, CMSIS specifies that the functions `systemInit` and `systemCoreClockUpdate` be supplied. For most vendor supplied `system_<device>.c` files, that's all you get. In this chapter we show the code for system initialization going into considerably more detail. The code is placed in the file, `system_apollo3.zig`.

`systemInit()`

The `systemInit()` function is intended to be invoked first in the reset handler to put the computer in its intended operational state.

For example, most microcontrollers have multiple ways to supply clocks to the core and peripherals. Often, an external high speed crystal oscillator is used to clock the processor. But such crystals need to be started up and there is a period of time where they must stabilize. To get the processor running at all, there is usually an internal RC oscillator which runs at a lower frequency, but is ready for use immediately. These are the types of duties allocated to `systemInit()`.

There is one other detail about `systemInit()` which is easily overlooked. At the time `systemInit` is invoked, memory has not been set up according to the expectation of a Zig program. This makes some sense as you are just trying to get things to run in a stable configuration. After `systemInit()` returns and before `main()` is invoked, the initialized data section and the zero-initialized data section are set up as shown in the previous section. The consequence is that assigning values to global variables has no effect since the read/write data areas of the program will be modified after `systemInit` returns.

The following are the areas of general interest for the class of programs intended to run:

1. Clock frequency of the processor core.
2. Flash memory cache.
3. Vector table offset.
4. Processor controls.
5. Floating point unit (FPU).
6. Exception priorities.
7. Memory protection unit (MPU).

```
<<system_apollo3: systemInit>>=
/// The `systemInit` function initializes the core processor and a limited
/// number of peripherals to enable the system to run. You cannot assume the contents
/// of RAM at the time it is called.
export fn systemInit() void {
    // Do not use global variables because this function is called before
    // memory is initialized. Read/write sections maybe overwritten afterwards.

    apollo3.pwrctrl.dev_pwr_en.writeDirect(0); ①

    <<systemInit: high frequency clock>>
    <<systemInit: enable flash cache>>
    <<systemInit: vector table>>
```

```

<<systemInit: system timer>>
systemInitRam() ;
systemControlsInit() ;
systemFpuInit() ;
systemExcPriorityInit() ;
if (!config.disable_mpu) {
    systemMemProtectInit() ;
}
systemCoreClockUpdate();

// In preparation for the switch to running on the Process
// Stack, the PSP must be initialized.
const process_top = @extern(*u64, .{
    .name = "__process_stack_top__",
    .linkage = .strong,
});
v7m.psp.set(@intFromPtr(process_top)); ②
}

```

- ① Power all the devices off. Remember, we may arrive at this code via bootloader and don't want to depend upon any system state the bootloader may have failed to clean up.
- ② Execution is now on a direct path to `main()` and so the PSP can be set to its top value. At this location, the processor is using the Main Stack.

For the rest of this section, the discussion is specific to the manner in which the ARM core and the Apollo 3 Blue SOC will be run. Some familiarity with the data sheet is essential for a complete understanding, but the general functions dealt with here are present in other microcontrollers.

```

<<system_apollo3: imports>>=
/// Access to compile time configuration constants.
const config = @import("config");
/// Access to the ARM core.
const v7m = @import("v7m");
/// Access to the Apollo3 peripherals.
const apollo3 = @import("apollo3");

```

High Frequency Clock

According to the Apollo 3 data sheet, the reset value of the core clock control sets the High Frequency clock to half speed. Here it is adjusted up to full speed. The effects on power consumption are difficult to predict. The higher speed clock consumes more power, but the processor can finish computations more quickly and return to lower power modes. Measurement on an actual running application is necessary if ultra-low power operation must be considered.

```

<<systemInit: high frequency clock>>=
apollo3.clkgen.clk_key.write(.unlock_key); ①
apollo3.clkgen.c_ctrl.writeFields({ .core_sel = .hfrc });
apollo3.clkgen.clk_key.write(.lock_key);

```

- ① It is necessary to unlock (and later relock) the clock generator registers to change them. This protects against accidental changes. The default value of the field is the magic value needed to "unlock" the clock generator peripheral.

Flash Memory Cache

To speed up the access to instructions and data obtained from flash memory, the Apollo 3 has a cache which can return data without any wait states on a cache hit. The setup code here is cribbed from the Ambiq SDK function, `am_hal_cachectrl_config()` using values from the default configuration as well as recommended values from the data sheet. The cache controller is a sophisticated peripheral and only a subset of its capabilities are used here.

```
<<systemInit: enable flash cache>>=
apollo3.cachectrl.cache_cfg.writeFields(.{
    .data_clk_gate = 1,
    .cache_clk_gate = 1,
    .dcache_enable = 1,
    .icache_enable = 1,
    .config = .wl_128b_1024e,
});

apollo3.cachectrl.flash_cfg.writeFields(.{
    .lpm_mode = .standby,
    .se_delay = 5, // recommended value from datasheet
    .rd_wait = 1, // recommended value from datasheet
});

apollo3.cachectrl.cache_cfg.setBitField(.enable);
apollo3.pwrctrl.mem_pwd_in_sleep.setBitField(.cache_pwd_slp); ①
```

- ① An Ambiq report (AMA3B1KK-xxx) suggests powering down the cache in deep sleep mode to avoid any potential of corruption of data in the cache during deep sleep.

Vector Table Offset

To ensure that the processor uses the IRQ handlers defined here, the VTOR register is set to point to the exception vector table.

```
<<systemInit: vector table>>=
const vtab = @extern(*u32, {.name = "__vectors", .linkage = .strong });
v7m.scb.vtor.writeDirect(@intFromPtr(vtab));
v7m.dsb(); ①
```

- ① This is the Data Synchronization Barrier instruction. It ensures that all the data transfers, to the SCB block in this case, are complete before the next instruction is executed. For the Cortex-M4, the SCB is *strongly ordered* and so the DSB is not strictly necessary. It is included based on ARM architectural recommendations. The best guide to these situations is the [ARM Cortex-M Programming Guide to Memory Barrier Instructions: Application Note 321](#).

System Timer

The Apollo 3 has a system timer peripheral which is used for timestamping and other purposes. Note this is different from the SysTick core peripheral. SysTick is not used in this design. Its clock rate is too high and it is only 24 bits long. The intended use for SysTick is as timer for round-robin task scheduling. This design does not use any tasks so scheduling is unnecessary.

The system timer peripheral is initialized here so that it is available as a source of timestamps,

particularly for fault handlers.

```
<<systemInit: system timer>>=
apollo3.stimer.st_cfg.setBitField(.freeze);
// The system timer is run off of the 32 kHz crystal oscillator.
// Each tick is approximately 30.5 microseconds.
apollo3.stimer.st_cfg.writeFields({ .clk_sel = .xtal_div1, });
apollo3.stimer.st_cfg.setBitField(.clear); ①
apollo3.stimer.st_cfg.clearBitField(.clear);
apollo3.stimer.stim_int_clr.writeBitField(.overflow);
apollo3.stimer.stim_int_en.writeBitField(.overflow); ②
apollo3.stimer.st_cfg.clearBitField(.freeze);
v7m.nvic.enableIRQ(apollo3.SystemTimer.irq_number);
```

- ① The "clear" bit field must be set to one then cleared to zero to cause the counter value itself to be zeroed.
- ② The timer overflow interrupt is used to track larger time counts in special purpose registers which also survive reset.

The timer overflow is handled by adding to the SNVR[0-3] registers to keep the extra time precision in system timer registers.

```
<<system_apollo3: stimerIrqHandler>>=
/// The `stimerIrqHandler` function runs in response to the overflow
/// interrupt of the `stimer` peripheral. The overflow is used to keep track
/// of larger time increments beyond 32 bits. The `stimer` peripheral
/// has 4 non-volatile registers that are dedicated to that purpose.
fn stimerIrqHandler() callconv(.c) void {
    if (apollo3.stimer.stim_int_stat.testBitField(.overflow)) {
        apollo3.stimer.stim_int_clr.writeBitField(.overflow);

        var result, var ovf = @addWithOverflow(apollo3.stimer.s_nvr[0], 1);
        apollo3.stimer.s_nvr[0] = result;
        result, ovf = @addWithOverflow(apollo3.stimer.s_nvr[2], ovf);
        apollo3.stimer.s_nvr[1] = result;
        result, ovf = @addWithOverflow(apollo3.stimer.s_nvr[2], ovf);
        apollo3.stimer.s_nvr[2] = result;
        apollo3.stimer.s_nvr[3] += ovf;
    } else {
        v7m.nvic.disableIRQ(apollo3.SystemTimer.irq_number); ①
    }
}
comptime {
    @export(&stimerIrqHandler, {.name = "stimerIrqHandler", .linkage = .strong}); ②
}
```

- ① The overflow and capture registers share the main STIMER_IRQn interrupt vector. Currently, this design does not use the capture registers. If some interrupt other than an overflow arrives, then somehow the capture register interrupts were enabled without modifying this IRQ handler. The fallback is just to shut the whole thing down. The compare registers of the system timer

peripheral have their own interrupt vectors.

- ② Publish this as a **strong** symbol to override the default handler.

Initializing RAM Memory at Start Up

It is necessary to ensure that the values of non-automatic variables which are initialized have their initial values in place when `main` runs. It is necessary to ensure that non-automatic variables which do *not* have initializers are set to zero, *i.e.* all bits in the variable are zero. With a conventional operating system, the program is set up to bring in the variable initializers from the disk file which holds the program executable image. This is usually done on demand, *i.e.* when the program actually accesses a variable which was initialized. Zero initialized pages can be allocated by the operating system and zeroed in place.

On a microcontroller there may not be a file system or even any external storage. Only one program is typically executed and there is no mechanism to page in from disk. At power up, RAM values are indeterminate, so the variable initializers must be stored somewhere and copied into RAM. Once the RAM memory is initialized, normal program execution can then read or update the values as needed. The most convenient source of non-volatile memory is the flash memory where our program is stored. So, the data initializers must be stored in flash and copied to RAM before `main` executes. Using flash memory to store initializers does reduce what is available for program instructions, but in most microcontrollers the amount of flash memory far exceeds the amount of RAM memory.

A similar situation exists for zero initialized memory. Since the value to be stored in RAM is just zero, there is no need to waste flash space storing a potentially large array of zeros. So, the memory locations occupied by uninitialized variables must be programmatically set to zero.

Where does the necessary information to copy memory or zero memory come from? The memory addresses are only known at link time and the specifics will vary from one program to another. The linker is able to supply the needed information. How that is done is covered in a [following section](#). For now, assume that there are some symbols whose values are set by the linker. These symbols give you all the information needed to write the code to perform the memory initialization. Note that the code and the linker directives are now tightly coupled by the names given to those symbols. The chosen names are conventional and follow a pattern, but the names are arbitrary as long as both the initialization code and the linker directives use the same name to mean the same thing.

Two other memory initialization situations arise:

1. Procedure linkages and automatic variables are stored on the stack. It does not need to be initialized for proper operation since the procedure linkages are always written before they are read and automatic variables have indeterminate values before they are written. However, the stack has a fixed size and it is convenient to initialize the stack to some known, improbable value in order to determine the extent of stack usage. By initializing the stack area and then running the system, execution can be halted to examine a memory dump of the stack area. The contrast between the improbable initial memory value and what is written during program execution will give a good measure of the extent of stack usage. There are other, and better ways, to protect the system from stack excursions. The MPU can be used or clever stack placement in memory can catch the stack excursions which a deeply recursive function might trigger. But simply initializing the stack memory to known values is an easy way to get started.
2. It is convenient to have a section of memory set aside which is not initialized unless there has been a power on reset (POR). A processor reset can come from many sources and may even be caused intentionally by the software. It is useful during development to leave a trail of clues about the state of the system before the reset by storing information in a section of memory which is not, unlike ordinary variables, initialized at reset. Of course, should the reason for the

reset be power on, i.e. a hard, cold start, RAM contents are indeterminate and the trail is lost. But during development, using a special section of memory can be a useful tool to uncover difficult to diagnose errors. Such a memory area is also useful for gathering operational statistics which must survive reset. If an external interface to the memory contents is provided (e.g. via some communications interface), it can help diagnose problems in a deployed system. Most modern microcontrollers have a specific peripheral to monitor and control resets and that hardware usually has registers which indicate the cause of the last reset. The Apollo 3 has such a register.

The implementation of the RAM initialization follows directly from the considerations of the four memory segments just discussed.

```
<<system_apollo3: systemInitRam>>=
/// The `systemInitRam` function is run as part of the system initialization
/// that happens at reset time. It initializes various sections of RAM memory
/// to ensure that initial data values are in place, stacks have been painted
/// with a special value and allows for memory section values to survive
/// a reset that is not the result of a power on.
fn systemInitRam() void {
    <<systemInitRam: data type declarations>>
    const SpecialMemoryArea = struct { ①
        <<systemInitRam: data declarations>>
    };
    <<systemInitRam: data sections>>
    <<systemInitRam: bss sections>>
    <<systemInitRam: stack sections>>
    <<systemInitRam: no init sections>>
}
```

- ① Note that we use namespace container variables to allocate the memory needed by various sections that require special treatment.

Copying Variable Initializers to RAM

There are couple of conventional tactics used to get the linker derived information into the memory initialization code.

1. You can define linker symbols which mark the boundary of a section and use those symbols to know the source of variable initializers and the RAM destination of the copy. This is simple and works well if the layout of initializer data is contiguous.
2. You can use the ability of the linker to place values into memory to define a *table* which contains a set of source and destination information. This approach easily handles multiple, non-contiguous initializer sections.

Here the second approach is used. It is only slightly more involved (requires a loop to iterate through the table) and does not require significantly more code to implement. The linker script directs the linker to build a *table* containing the needed descriptive information about the memory section.

That table is constructed such that you can declare an equivalent Zig structure. In Zig terms, and `extern struct` has a known layout in memory equivalent to that generated by "C". This is, for example, a common technique used to define the layout of memory-mapped peripheral device registers. In this case, the layout and values are being provided through a linker script. You must declare an equivalent Zig structure to enable the code to retrieve the linker-generated values

correctly.

The information required to copy data initializers to RAM appears in Zig source code to be an array of structures of the following type. Note the design uses 32-bit quantities which are convenient for the Cortex memory architecture.

Data copy structure declaration

```
<<systemInitRam: data type declarations>>=
const CopySegmentDesc = extern struct {
    src: [*]const u32,
    dst: [*]u32,
    nwords: u32, ①
};
```

- ① In units of 32-bit words, i.e. `nwords` is the number of 32-bit words to copy. All subsequent `nwords` fields are also in 32-bit word units.

The memory initialization code then uses the values in the linker-generated table to perform the initialization. The net effect is no different than if you had built the table by defining the array directly in our own "C" code. But since the needed values are only known at link time, the linker is enlisted to place the values into memory for us. This must be done carefully, as discussed [below](#), to ensure matching the memory layout implied by the `CopySegmentDesc` structure.

If you treat the copy table as an array, you must know where the array ends. If the number of elements was known, then the end could be calculated. However, in laying out the array using linker commands, the easiest solution is to define two linker symbols which mark the start and end of the implied array of copy segments.

```
<<systemInitRam: data sections>>=
const copy_seg = @extern([*]CopySegmentDesc, .{
    .name = "__data_copy_table_start__",
    .linkage = .strong,
});
const copy_seg_end = @extern(*CopySegmentDesc, .{
    .name = "__data_copy_table_end__",
    .linkage = .strong,
});
const copy_seg_len =
    (@intFromPtr(copy_seg_end) - @intFromPtr(copy_seg)) / @sizeOf(CopySegmentDesc);
const copy_segments = copy_seg[0..copy_seg_len]; ①

for (copy_segments) |copy_segment| {
    const dst = copy_segment.dst[0..copy_segment.nwords]; ②
    const src = copy_segment.src;
    @memcpy(dst, src);
}
```

- ① Since we have a table of segments to copy, create a slice from the raw pointers. We have to perform the pointer arithmetic ourselves since Zig does not (yet) allow pointer arithmetic within array boundaries.

- ② The strategy is to use @memcpy, so we compute a slice for each memory segment to be copied.

Zeroing Out Uninitialized Variable Values in RAM

The information required to place zero values into memory is simpler since we are writing a constant zero into the memory region.

As before with the initialized data, you create an `extern struct` declaration to match the layout of data produced by the linker commands.

```
<<systemInitRam: data type declarations>>=
const InitSegmentDesc = extern struct {
    dst: [*]u32,
    nwords: u32,
};
```

The implementation of the code to zero out the memory follows a similar pattern used to copy over the initialized data values. In this case, the strategy is to use `@memset` to perform the zero initialization.

```
<<systemInitRam: bss sections>>=
const zero_seg = @extern([*]InitSegmentDesc, .{
    .name = "__zero_table_start__",
    .linkage = .strong,
});
const zero_seg_end = @extern(*InitSegmentDesc, .{
    .name = "__zero_table_end__",
    .linkage = .strong,
});
const zero_seg_count =
    (@intFromPtr(zero_seg_end) - @intFromPtr(zero_seg)) / @sizeOf(InitSegmentDesc);
const zero_segments = zero_seg[0 .. zero_seg_count];

for (zero_segments) |zero_segment| {
    const dst = zero_segment.dst[0 .. zero_segment.nwords];
    @memset(dst, 0);
}
```

Setting the Stack Memory to a Known Value

Before discussing initialization of the stack section, two design decisions must be made:

How many stacks do we support?

The ARMv7-M architecture supports both a Main Stack and a Process Stack. You must supply a Main Stack.

Where in memory is the stack to be located?

There are several strategies which are practiced. The stack, unlike most other memory usage, grows down, i.e. smaller stack addresses hold more recent stack data.

For this use case, a two stack arrangement is supported for the following reasons.

- Exception processing always uses the Main Stack. This type of processing tends to be better bounded in terms of its stack usage. It is less likely to have any deep recursion or need large amounts of automatic variable space.
- The Process Stack is used by application code and is much less predictable in the amount of memory it requires.
- By separating the Process Stack from the Main Stack, it is unnecessary to add extra space in the Process Stack for the stack space needed by exceptions. Exception nesting is always present since high priority system faults can occur at any time, *i.e.* you must always account for Hard Fault and NMI.
- Separate Main and Process Stacks offer the opportunity to restrict access to the memory based on privilege. This could be used to prevent unprivileged code from accessing the Main Stack.

The considerations can be quite involved as this [application note](#) details.

Several stack placement strategies are commonly used. One strategy is to place the stack at the top of RAM and let it grow downward without any bound. That strategy does nothing to guard against a potential deeply recursive function working all the way down RAM and overwriting program data. Another strategy is to define the bottom limit of the stack to be the beginning of RAM and place the initial top of the stack somewhere higher up, say 4KiB, in memory. Since memory below the beginning of RAM is usually not present, this can cause a fault if the stack grows too large. This strategy is adopted here. The process stack is placed at the bottom of RAM and the main stack located immediately next to it, higher in memory.

It is difficult to estimate up front how large the stack should be. That's one of the reasons to fill the stack space with an unusual number that will allow you to get an estimate of usage. The build script for this code allows a `main_stack_size` configuration value so that different default can be obtained. Start with 4KiB as the default and make provisions to change the size with the configuration constant that can be defined as an option to the compilation.

The stack is then just an array variable of the desired size and the linker places the memory where it is directed.

```
<<systemInitRam: data declarations>>=
export var __main_stack:
    [config.main_stack_size]u8 align(@align0f(u64)) linksection(".main_stack") =
    [_]u8{0} ** config.main_stack_size; ❶
```

❶ Another information packed definition.

1. The stack is aligned on an 8 byte boundary to comply with AAPCS requirements.
2. Place the `__main_stack` variable in the `.main_stack` segment so you can tell the linker where to locate the stack in memory.
3. The `export` designation is necessary to have the memory show up since it is not otherwise referenced here. Unfortunately, the export also places the symbol in the global symbol table.

```
<<systemInitRam: data declarations>>=
export var __process_stack:
    [config.process_stack_size]u8 align(@align0f(u64)) linksection(".process_stack") =
    [_]u8{0} ** config.process_stack_size;
```

Initializing stack memory is similar to initializing the BSS section. However, there are two other

consideration that must be attended to:

1. The startup code itself is executing using the Main Stack. You must not overwrite any of the Main Stack that is in use. This means you must *not* write past the address contained in the MSP.
2. You want to write the bottom of the stack with a value which is different than the value used for filling the remaining stack memory. The different value can be used, potentially, to calculate the stack usage through a debugging interface.

Just as before, the linker script builds a table of stack sections. The only variation here is the different manner in which the stack section values are set.

```
<<systemInitRam: stack sections>>=
const stack_seg = @extern([*]InitSegmentDesc, .{
    .name = "__stack_table_start__",
    .linkage = .strong,
});
const stack_seg_end = @extern(*InitSegmentDesc, .{
    .name = "__stack_table_end__",
    .linkage = .strong,
});
const stack_seg_count =
    (@intFromPtr(stack_seg_end) - @intFromPtr(stack_seg)) / @sizeOf(InitSegmentDesc);
const stack_segments = stack_seg[0 .. stack_seg_count];

const msp = v7m.msp.get(); ①

for (stack_segments) |stack_segment| {
    const dst = stack_segment.dst;
    const segment_end = @intFromPtr(&dst[stack_segment.nwords]);
    const dst_end = if (msp > @intFromPtr(dst) and msp < segment_end)
        msp
    else
        segment_end; ②
    const dst_count = (dst_end - @intFromPtr(dst)) / @sizeOf(u32);

    const stack_words = dst[0 .. dst_count];
    stack_words[0] = 0xdeadbeef; ③
    @memset(stack_words[1 .. dst_count], 0x12345678) ;
}
```

① Since the stack pointer is in play, *i.e.* it is currently being used, you must make sure not to overwrite the portion of the stack that is in use.

② The if expression makes it possible to declare the value as `const`.

③ There are a few "cute" words which can be spelled with the alphabetic characters used for hexadecimal numbers. All that is required is something distinctive to recognize the boundary and it is useful if each byte of the 4 byte word is a different value.

Memory Which Survives Reset

Just as there are `.data` and `.bss` sections, memory intended to survive reset also needs to be split between sections which are explicitly initialized and zero initialized. So, you define the `.noinit_data` and the `.noinit_bss` sections.

The memory sections that are intended to survive reset must have their initialization conditioned upon whether an actual Power On Reset (POR) occurred. If a POR occurred, then the memory locations are set to zero. Otherwise, the memory is left in tact.

For the Apollo 3 chip, the reset generator peripheral has a register which holds indicators of the source and type of the reset. This is the information used to determine if there was a POR.

There are two other considerations:

- A *magic* number is written into the first word of the memory section to indicate that the initialization is complete. That magic value is expected always to be there and if it is not, then the memory is reinitialized regardless of the POR status.
- The Apollo 3 data sheet indicates that the reset generator status register value is *not* retained when deep sleep is entered. To make it available to the application, it is stored in the *.noinit_bss* section.

These considerations are captured in a variable. The variable serves to give the section a size and other parts of the system may declare variables in the *.noinit_data* or *.noinit_bss* sections to store things they wish to survive reset. One good use would be to store the conditions detected during a major system exception such as a Hard Fault.

To access Core definitions and other chip definitions, a module is provided that contains the specifics of the memory mapped I/O for the Apollo 3 microcontroller.

```
<<systemInitRam: data type declarations>>=
const noinit_magic: u32 = 0xe4fd777; ①
```

```
<<systemInitRam: data declarations>>=
const NoinitSegmentStatus = extern struct {
    magic: u32,
    reset_status: apollo3.ResetStatus.StatusType,
};
```

① An arbitrary, improbable number that is used as the initialization indicator.

```
<<systemInitRam: data declarations>>=
var noinit_status: NoinitSegmentStatus linksection(".noinit_bss") = .{
    .magic = 0,
    .reset_status = @bitCast(@as(u32, 0)),
};
```

The code to initialize the segments combines the tables created by the linker (as was used in the other segments) along with the *noinit_status* variable.

```
<<systemInitRam: no init sections>>=
const reset_status = apollo3.rststat.stat.read();
const pwr_on_reset = reset_status.por_stat == 1 or reset_status.poir_stat == 1; ①

if (pwr_on_reset or SpecialMemoryArea.noinit_status.magic != noinit_magic) {
    const noinit_seg = @extern([*]CopySegmentDesc, .{
        .name = "__noinit_data_copy_table_start__",
        .linkage = .strong,
```

```

});  

const noinit_seg_end = @extern(*CopySegmentDesc, .{  

    .name = "__noinit_data_copy_table_end__",  

    .linkage = .strong,  

});  

const noinit_seg_count =  

    (@intFromPtr(noinit_seg_end) - @intFromPtr(noinit_seg)) / @sizeOf(CopySegmentDesc);  

const noinit_segments = noinit_seg[0 .. noinit_seg_count];  

for (noinit_segments) |noinit_segment| {  

    const dst = noinit_segment.dst[0 .. noinit_segment.nwords];  

    const src = noinit_segment.src;  

    @memcpy(dst, src);
}  

const noinit_bss_seg = @extern([*]InitSegmentDesc, .{  

    .name = "__noinit_bss_table_start__",  

    .linkage = .strong,  

});  

const noinit_bss_seg_end = @extern(*InitSegmentDesc, .{  

    .name = "__noinit_bss_table_end__",  

    .linkage = .strong,  

});  

const noinit_bss_count =  

    (@intFromPtr(noinit_bss_seg_end) - @intFromPtr(noinit_bss_seg)) /  

@sizeOf(InitSegmentDesc);  

const noinit_bss_segments = noinit_bss_seg[0 .. noinit_bss_count];  

for (noinit_bss_segments) |noinit_bss_segment| {  

    const dst = noinit_bss_segment.dst[0 .. noinit_bss_segment.nwords];  

    @memset(dst, 0);
}  

SpecialMemoryArea.noinit_status.magic = noinit_magic ;  

}  

SpecialMemoryArea.noinit_status.reset_status = reset_status ;

```

- ① For the Apollo 3 SOC, the reset generator peripheral allows for software to generate the same power on initialization signal as would be generated in hardware. The reset status keeps separate status for the two conditions, but you want to have the same behavior for both.

systemControlsInit()

The `systemControlsInit()` function initializes system registers which control the actions taken by the processor in response to situations created by the running program.

```

<<system_apollo3: systemControlsInit>>=
fn systemControlsInit() void {
    v7m.scb.ccr.updateFields(.{
        .stk_align = 1, ①
        .div_0_trp = 1,

```

```

        .unalign_trp = 1,
    });
    v7m.scb.scr.setBitField(.sleep_deep);
}

```

① Enabled by default on a Cortex-M4.

The Cortex-M4 core has a number of controls that affect aspects of behavior is enforced for all application programs.

- Stack alignment is set to 8 bytes to conform to AAPCS requirements and conventional usage. Care must be taken that stack areas are 8 byte aligned.
- Division by zero is not allowed. Any code which divides must either know by context that the divisor is non-zero or must make a test of zero to prevent the condition.
- Unaligned memory accesses are not allowed. *N.B.* that the target feature `+strict_align` must be given to the Zig compiler to prevent it from generating code which has unaligned accesses. Unaligned access can result from *up casting* a pointer from a lesser alignment to a greater alignment. This is an indication of a program design issue. Generally, Zig is quite pedantic about alignment issues, but this setting gives a check by the processor core that is independent of the release level of the compiler.
- Enabling deep sleep mode in the processor requires a control bit to be set.

systemFpuInit()

The `systemFpuInit()` function initializes the system registers associated with the FPU to configure access and stack context behavior.

```

<<system_apollo3: systemFpuInit>>=
fn systemFpuInit() void {
    v7m.scb.enableFpu(); ①
    v7m.fpu.setFpuStackContext(.no_stack); ②
}

```

① By default after reset, floating point operations are not enabled. Here, they are enabled with full access.

② This setting is important and the setting chosen here is appropriate when there is no task context switching and none of the IRQ handlers use floating point. This describes the types of the targeted applications.

The Cortex-M4 has a single precision floating point unit. In this design, the FPU is dedicated to application use and no floating point operations are allowed from IRQ handlers. Since there is only a single processor context, it is never necessary to stack the floating point registers.

systemExcPriorityInit()

Setting the priorities for system exceptions and interrupts is an extended topic. It is discussed in a following [section](#). There the code required to initialize the exception priorities is shown.

systemMemProtectInit()

The `systemMemProtectInit()` function initializes the Memory Protection Unit. Once initialized is also enables the Memory Management fault. Since the subject of memory protection requires

considerable discussion, MPU initialization is deferred to a later [chapter](#).

systemCoreClockUpdate()

The `systemCoreClockUpdate` function updates the value of the global variable, `systemCoreClock` to be the frequency of the clock running the processor in units of Hertz. This is a standard CMSIS function. System code should invoke this function after any change to the frequency of the core clock. Applications may use the global value, `systemCoreClock`, to compute frequency information for other clocks in the system which may be derived from the core clock frequency.

The function and global variable follow CMSIS conventions.

```
<<system_apollo3: systemCoreClockUpdate>>=
/// The `systemCoreClockUpdate` function calculates the system clock frequency
/// based upon current register settings.
/// This function can be used to update the system core clock frequency
/// when clock control registers are changed. The value of the
/// core clock is global and can be used by peripheral control code
/// when the peripheral clocking scheme is related to the core clock frequency.
fn systemCoreClockUpdate() void {
    systemCoreClock = apollo3.sys_osc_clk /
        (@intFromEnum(apollo3.clkgen.c_ctrl.readField(.core_sel)) + 1);
}
```

```
<<system_apollo3: systemCoreClock>>=
// System Clock Frequency (Core Clock) ①
export var systemCoreClock = apollo3.system_clock;
```

① Note that when the initialized data is copied from flash memory to RAM during RAM initialization, the core clock frequency value will match the effect of `systemCoreClockUpdate()` function.

missingHandler()

The default implementation for any exception handlers which are *not* overridden by the application program is to invoke the abnormal ending function.

```
<<system_apollo3: missingHandler>>=
fn missingHandler (
    _: *const v7m.ExceptionFrame,
    _: u32,
    _: u32,
) callconv(.c) noreturn {
    systemAbEnd(); ①
}

comptime {
    @export("missingHandler", {.name = "missingHandler", .linkage = .weak });
}
```

- ① **N.B.** no recovery is attempted here. The default implementation forces an abnormal termination the program.

System Code Layout

This section shows how the content of the file `system_apollo3.zig` is composed from literate program chunks.

```
<<system_apollo3.zig>>=
//! The `system_apollo3.zig` file contains system specific initialization code
//! for the Ambiq Apollo 3 microcontroller. This code is invoked from the system
//! reset handler.
<<edit warning>>
<<copyright info>>

// This file was substantially derived from the Apollo 3 HAL.
// The following is the copyright information for the HAL.
<<ambiq copyright info>>

// Imports
<<system_apollo3: imports>>

// Declarations
<<system_apollo3: declarations>>

<<system_apollo3: systemInit>>
<<system_apollo3: systemInitRam>>
<<system_apollo3: stimerIrqHandler>>
<<system_apollo3: systemCoreClock>>
<<system_apollo3: systemCoreClockUpdate>>
<<system_apollo3: systemControlsInit>>
<<system_apollo3: systemFpuInit>>
<<system_apollo3: systemExcPriorityInit>>
<<system_apollo3: systemMemProtectInit>>
<<system_apollo3: missingHandler>>
<<system_apollo3: systemAbEnd>>
```

Starting main

At this point in starting up the system, we have completed the minimal initialization that is required and are ready to transfer control to the `main` application. This function constructed so that it can reside in a separate object file.



It may seem strange to create a separate function for `startMain` and to place it in a separate file. When the `CONTROL` register is set for unprivileged execution, the next instruction must be fetched from memory which has unprivileged access. At this time, the Memory Protection Unit (MPU) may have been configured by the `systemInit` function and unprivileged execution must be separated from the privileged execution of the Reset Handler. This is discussed in a [subsequent section](#).

The design of this function follows the pattern for start up in the Zig standard library (`start.zig`) but is much simplified from that implementation.

```
<<start_main: functions>>=
/// The `startMain` function transfers control to the `main` function
/// to begin execution of the application program. N.B. that
/// this function resets the stack pointers and forces the
/// execution to be unprivileged and using the Process Stack.
export fn startMain() noreturn {
    // We enter this code as the last step of the Reset Handler.
    // We are executing in Thread mode, with privilege.
    // The stack pointer has already been set to use the Process Stack.
    // The Memory Protection Unit has been initialized and this
    // code was must be placed in the unprivileged text section.
    v7m.control.update(.{
        .npri = .unprivileged,
    });
    ①

    // After the update of the control register, we are executing
    // in Thread mode, unprivileged, and using the Process Stack.
    // Background execution continues under these conditions.

    const bad_main_ret =
        \\ expected return types of main are:
        \\ 'void', '!void', 'noreturn', 'u8', or '!u8'
        ;
    const ReturnType = @typeInfo(@TypeOf(root.main)).@("fn").return_type.?;
    switch (ReturnType) {
        void => {
            root.main();
            @panic("returned from main with no value");
        },
        noreturn => {
            root.main();
            @panic("unexpected return from main");
        },
        u8 => {
            _ = root.main();
            @panic("returned from main with value");
        },
        else => {
            if (@typeInfo(ReturnType) != .error_union) @compileError(bad_main_ret);
            root.main() catch @panic("returned from main with error");
        },
    }
    @panic("aborting: return from main");
}
```

① The core CONTROL register is used to select the stack used for Thread mode and to remove privilege execution from the running thread. See the [following](#) discussion for the rational of using unprivileged execution. Immediately after the CONTROL register is updated, we begin

unprivileged execution. N.B. that it is essential that the update function be inlined. Otherwise, we will switch stack pointers before the function return and execution is then uncontrolled.

Start Main Code Layout

The code that invokes the `main()` function is in a separate file as mentioned [previously](#).

```
<<start_main.zig>>
//! The `start_main.zig` module transfers control to the `main` program.
//! It is executed as the last step in the system Reset Handler.
//+
<<edit warning>>
<<copyright info>>

const root = @import("root");
const v7m = @import("v7m");

<<start_main: functions>>
```

Privileged Execution

Before transferring control to `main()`, two bits in the CONTROL register were changed that have significant design implications.

- Change the stack pointer for Thread mode operation to use the PSP.
- Set Thread mode operation to be unprivileged.

At reset, the processor executes in Thread mode (*cf.* Handler mode) using the MSP and privileged execution. This is the simplest mode of operation and allows unfettered access to the system by any code. Many systems are run in this configuration.

In this design, both stacks are used. The reasoning for a two stack approach was discussed [previously](#).

A more complex set of trade-offs is associated with executing the application code in unprivileged mode. By placing restrictions on the application code, the intent is to build a more constrained execution environment. Small software errors can easily send a microcontroller executing in an uncontrolled manner. The intent for a more controlled execution environment is to be able to catch unwanted behavior earlier in the development process. When a microcontroller system is deployed, there is usually little recovery opportunity should things go terribly wrong. You would like to be able to catch such behavior early and, if it does happen in a deployed system, to limit the potential for damage. By excluding the application code from operating with privilege, the problem is divided in half by ensuring that any computation on critical system registers or peripherals cannot happen in the application by means of a stray pointer.

The topic of constraining execution of the software is a recurrent theme in this book.

One trade-off for erecting an execution barrier between the system and the application is that it is computationally more costly to control the peripheral devices. Requiring privilege and protecting the memory mapped I/O regions means that all access to system and peripheral registers must happen via the exception handling mechanism. That mechanism involves at least pushing and

popping an exception stack frame for any system level access. Effectively, building a wall between the system and application processing implies scaling that wall whenever the two sides need to interact. It is simply the price that must be paid for a more robust system. Whether the increased cost of privileged access is a performance problem is determined both by the exception overhead and the frequency at which the overhead is incurred. This can only truly be determined by measurement. Good design practice attempts to minimize application code doing intensive system interactions in performance critical areas. There are usually effective ways to design the interactions between the unprivileged and privileged code so as not to impose an undue burden.

Note that the intent here is not to impose some level of security on the system to prevent malicious intent. Microcontroller systems typically execute a single program and, if field updates are protected from malicious intervention, our intent is to mitigate the effects of software bugs. Bugs are to software as noise is to electronics—something that does not go completely away despite your best efforts at control.

Scripting the Linker

The last steps to getting a program to run involve scripting the linker. Let's recap what has been done have so far.

- The exception vector table provides the Cortex-M4 core the necessary data to find the Reset handler function. The table was defined to include all the exception handlers and made arrangements for it to be populated either by a default function or one found elsewhere in the program code. The Reset handler function itself is provided and cannot be overridden.
- The Reset handler first initializes those aspects of the processor core required to execute any program.
- After the core is executing as intended, the RAM memory is initialized as required by the Zig language and to suit the needs of the applications.
- With the core initialized and RAM set up properly, `main` is called to start the application.

One nagging detail remains. Where in memory is the exception vector table and all the other code located? It's essential to be able to compile pieces separately. But with that flexibility comes the need to put the pieces back together, and that is what a linker does.

Linker sections have an additional complication. It has to have a name, and other pieces of code will ultimately need to refer to that name. The usual solution is to come up with a naming convention. Compilers use linker sections to keep code, initialized data, and uninitialized data separate. The names used are conventions imposed by the compiler. For other specialized sections, naming is arbitrary, but long standing conventions should be respected. Following established convention, the name, `.vectors`, has been given to the linker section holding the exception vector table.

Linker defined symbols must also be handled. As we have seen, the code simply uses the `@extern` builtin to obtain symbol values.

Linker script layout

A linker script must define:

- where to begin execution,
- the addresses of the different types of memory, and
- how that memory is filled with code and data.

```
<<apollo3.ld>>=
/*
Apollo 3 linker script for unprivileged execution.

<<copyright info>>
*/
<<linker script: entry point>>
<<linker script: memory layout>>
<<linker script: section descriptions>>
```

Entry point

In this environment, the program starts at the Reset Handler.

```
<<linker script: entry point>>=
ENTRY(resetHandler)
```

Memory layout

The following table shows the memory map for the Apollo 3 SOC. Note that RAM starts at `0x1000_0000` and that the first 64 KiB are Tightly Coupled Memory (TCM).

Address	Name	Executable	Description
0x00000000 – 0x000FFF	Flash	Y	Flash Memory
0x00100000 – 0x03FFFF	Reserved	X	No device at this address range
0x04000000 – 0x07FFF	External MSPI Flash	Y	XIP Read-Only External MSPI Flash
0x08000000 – 0x0800OFFF	Boot Loader ROM	Y	Execute Only Boot Loader and Flash Helper Functions.
0x08001000 – 0x0FFFFF	Reserved	X	No device at this address range
0x10000000 – 0x1000FFF	SRAM (TCM)	Y	Low-power / Low Latency SRAM (TCM)
0x10010000 – 0x1005FFF	SRAM (Main)	Y	Main SRAM
0x10060000 – 0x3FFFF	Reserved	X	No device at this address range
0x40000000 – 0x50FFF	Peripheral	N	Peripheral devices
0x51000000 – 0x51FFF	External Memory	X	Read/Write External Memory (MSPI) [Chip Rev B Only]
0x52000000 – 0xDFFFF	Reserved	X	No device at this address range
0xE0000000 – 0xE00FFF	PPB	N	NVIC, System timers, System Control Block
0xE0100000 – 0xFFFFF	Reserved	X	No device at this address range
0xF0000000 – 0xF0000FFF	Debug ROM	N	Debug ROM
0xF0001000 – 0xFFFFFFF	Reserved	X	No device at this address range

Table 1. Apollo 3 System Memory Map

The SparkFun board being used does not have any external memory. So, flash and RAM are the only two interesting areas of the memory map.

```
<<linker script: memory layout>>=
MEMORY
{
    FLASH (rx) : ORIGIN = 0x00010000, LENGTH = 960K
    RAM (rwx) : ORIGIN = 0x10000000, LENGTH = 384K
}
```

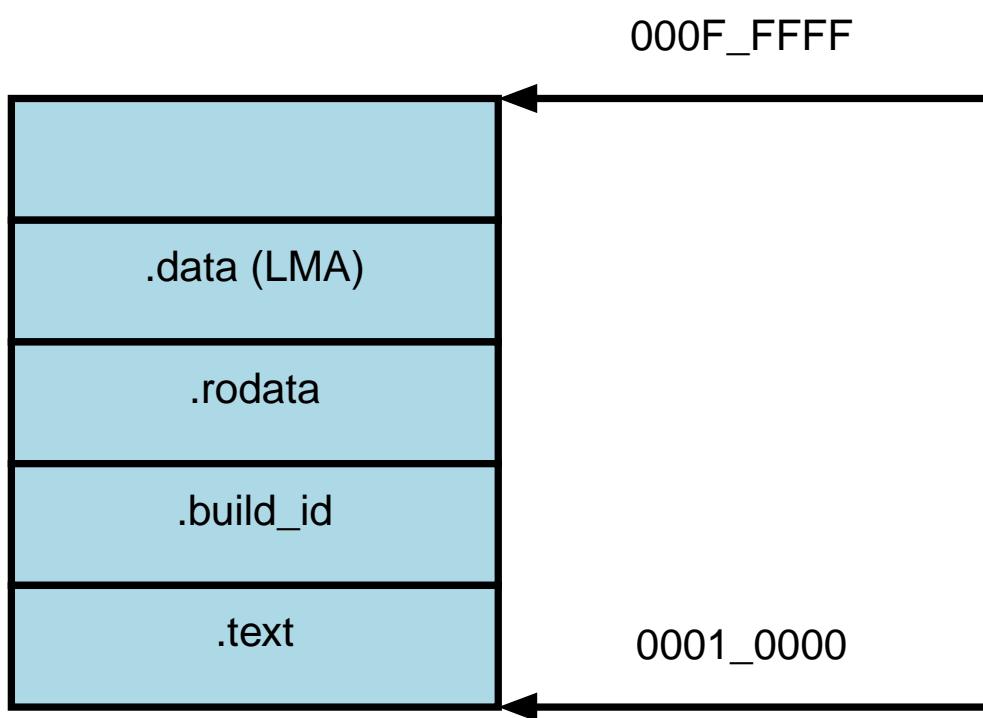
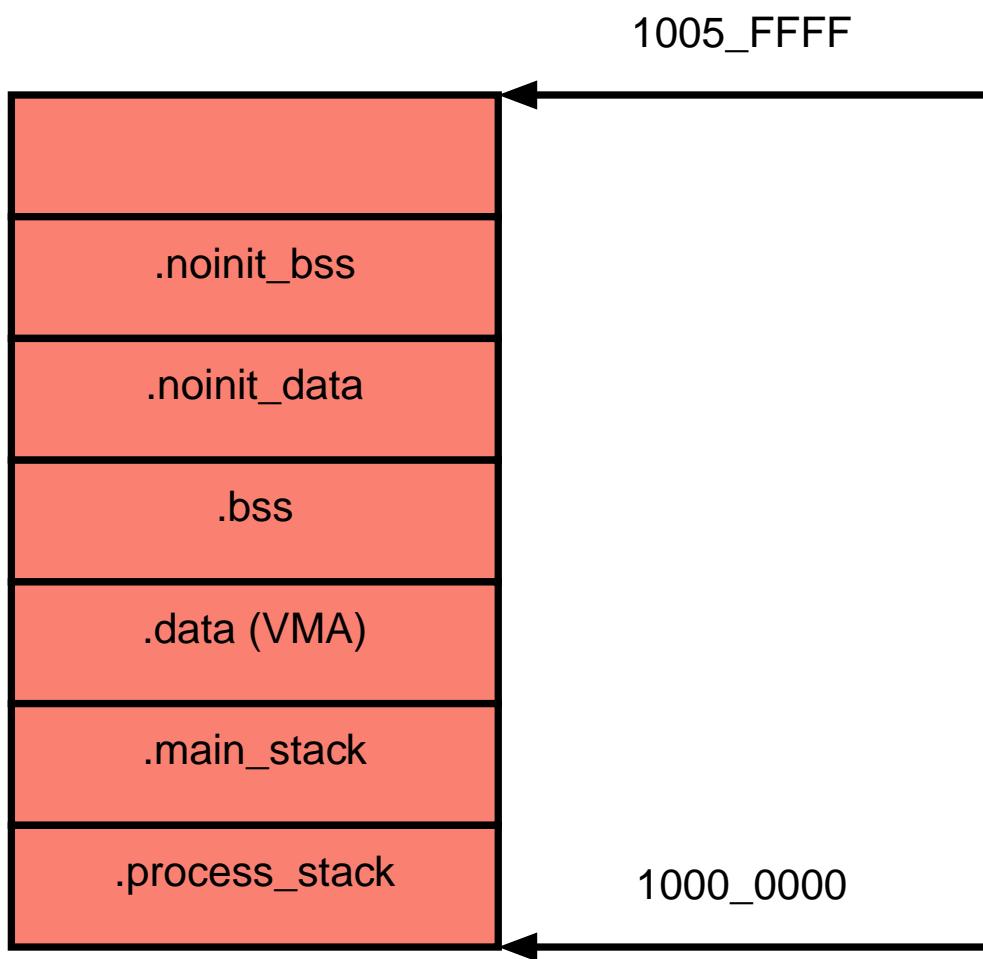


SparkFun has installed a secondary bootloader into the part. To preserve the secondary bootloader, the flash origin is set to 0x0001_0000. To overwrite the SparkFun bootloader, set the RAM origin at 0x0000_C000.

Linker sections

This section shows the linker commands which place the objects into memory. The order here is flash first and RAM second. The RAM layout order follows that of the [previous discussion](#) about the code which performs the RAM initialization.

The following diagram shows the layout in graphical form.



The linker script to layout the code contains a set of output section descriptions in the same order.

```
<<linker script: section descriptions>>
SECTIONS
{
    <<linker section: text>>
    <<linker section: build_id>>
    <<linker section: rodata>>
    <<linker section: ARM.extab>>
    <<linker section: process stack>>
    <<linker section: main stack>>
    <<linker section: data>>
    <<linker section: bss>>
    <<linker section: noinit data>>
    <<linker section: noinit bss>>

    PROVIDE(end = .) ;
}
```

Ⓐ A conventional linker generated symbol, **end**, is to show the end of occupied memory.

Text section

The **.text** section includes all the object code for the program. The exception vector table is placed first so it is located at the origin of the flash memory.

```
<<linker section: text>>
.text : ALIGN(4)
{
    __text_start__ = . ;
    KEEP(*(.vectors))
    KEEP(*(.ble_patch))

    . = ALIGN(4) ;
    *(.text*)

    . = ALIGN(4) ;
    __text_end__ = . ;
} > FLASH
```

The corresponding external declarations for the linker symbols is then given by:

Build id section

The **.build_id** section loads the contents of a *note* section which contains an identifier of the executable. To obtain a build id in a section requires an explicit request to the linker by using the **--build-id** linker command option.

```
<<linker section: build_id>>
.build_id : ALIGN(4)
```

```
{
    PROVIDE(SystemBuildIDNote = .) ;
    *(.note.gnu.build-id)
} > FLASH
```

The linker places the build ID in a *note* section. The layout of data in that section contains several elements. Again, a "C" structure declaration must be constructed that matches the layout data as the linker generates it. The following structure declaration shows the components.

```
<<startup_apollo3: declarations>>=
const ElfNoteSectionDesc = extern struct {
    namesz: u32,
    descsz: u32,
    type: u32,
    data: u8, // Beginning byte of the data.
};
```

The linker packs naming and descriptive information together into the variable sized **data** member. The offset in **data** to the build ID is given by the **namesz** member.

Read only data section

The compiler places read-only data in a separate section from executable code. Ultimately, the MPU is used to prevent executing the read-only data. This just makes that step a bit easier when it is introduced later.

In addition to the read only data sections emitted by the compiler, the initialization copy tables are placed in the same section. These tables are read only data, but are generated by the linker rather than the compiler.

```
<<linker section: rodata>>=
.rodata : ALIGN(4)
{
    __rodata_start__ = . ;
    *(.rodata*)
    . = ALIGN(4) ;
    <<linker section: memory init tables>>
    . = ALIGN(4) ;
    __rodata_end__ = . ;
} > FLASH
```

Memory Initialization Tables

In a [previous section](#), linker defined tables were used to describe the memory regions which needed initialization. Here, the linker commands required to build the copy table values are described. There are five areas of RAM which require specialized initialization.

```
<<linker section: memory init tables>>
<<linker tables: data copy>>
<<linker tables: bss zero>>
<<linker tables: stack>>
<<linker tables: noinit data copy>>
<<linker tables: noinit bss zero>>
```

To copy a section of memory to another location, the address of the source, the address of the destination, and the amount of data to copy are needed. The linker is requested to place the required information into the linked output. To be useful to the code, the sequence of 32-bit values placed into memory by the linker must match the data type declaration that is used by the code. *N.B.* assumptions about the alignment and padding of structure members used by the compiler are being made. These are safe assumptions in this case since the Cortex-M4 has a regular 32-bit word size that does not need any padding. Still, the linker and compiler do not know about this correspondence and the responsibility to ensure it is correct falls upon us.

Data copy table values

```
<<linker tables: data copy>>
__data_copy_table_start__ = . ;
LONG(LOADADDR(.data))           ①
LONG(ADDR(.data))              ②
LONG(SIZEOF(.data) / 4)
__data_copy_table_end__ = . ;
```

- ① The LOADADDR() operator returns the address where the section is loaded in memory.
- ② The ADDR() operator returns the address where the section resides during execution.

In the previous linker script segment, the LONG command places a 32-bit value into the output section. These commands also illustrate the linker concepts of *virtual memory address* (VMA)^[8] and *load memory address* (LMA). The VMA is the address where the data is assumed to reside when the program is executing. The LMA is the address where the section data is placed before execution begins, *i.e.* where it is loaded in memory by the executable file. For all other sections, these two addresses are the same. But for the .data section, the initializer values must be copied from flash, *i.e.* from the LMA, to where they are referenced by the program code, *i.e.* to the VMA.

When the linker resolves symbols between code and data, the resolution happens in VMA terms. Setting the LMA to be different from the VMA and then placing the LMA of the .data section into flash memory allows the use of flash memory as non-volatile storage for the data initializer values. At reset, the values are copied to the VMA and the program begins execution with a known state for its initialized variables. The VMA for the .data section must be placed in RAM since a program may certainly modify the values of its initialized variables. The linker conveniently provides the built-in functions, ADDR() to obtain the VMA of a section and LOADADDR() to obtain the LMA of a section. So, for the copy table, the LMA of the .data section is the source for the copy and the VMA of the .data section is the destination. How to specify the LMA of a section to be different from its VMA is shown below.

The access to the linker generated table is completed by an external declaration of the the copy table giving the compiler the data type whose structure has been constructed to match the way in which the linker placed the values in memory.

The initialized data section which survives reset still needs have its initializers present so they may

be copied into RAM when the start up code detects that there has been a power on reset. At power on reset, the memory is indeterminate and the data initialization values must be copied to the correct location.

Noinit_data initialization table

```
<<linker tables: noinit data copy>>=
__noinit_data_copy_table_start__ = . ;
LONG(LOADADDR(.noinit_data))
LONG(ADDR(.noinit_data))
LONG(SIZEOF(.noinit_data) / 4)
__noinit_data_copy_table_end__ = . ;
```

ARM Exception Information

We have no use for exception unwinding information. It is discarded to keep the executable file sizes down and clear as to exactly what is needed.

```
<<linker section: ARM.extab>>=
/DISCARD/ : { *(.ARM.exidx* .gnu.linkone.armexidx*) }
/DISCARD/ : { *(.ARM.extab.*) }
```

The remaining memory areas which require initialization do *not* require a stored set of values. The memory values are being set to a fixed number, so the address of the section and its length are all that is needed. All of these sections must be allocated to RAM since the program writes to the memory during initialization and during execution.

Start with the .bss section.

BSS initialization table

```
<<linker tables: bss zero>>=
__zero_table_start__ = . ;
LONG(ADDR(.bss))
LONG(SIZEOF(.bss) / 4)
__zero_table_end__ = . ;
```

The remaining initialization tables follow the established pattern.

Stack initialization table

```
<<linker tables: stack>>=
__stack_table_start__ = . ;
LONG(ADDR(.process_stack))
LONG(SIZEOF(.process_stack) / 4)
LONG(ADDR(.main_stack))
LONG(SIZEOF(.main_stack) / 4)
```

```
_stack_table_end_ = . ;
```

Noinit_bss initialization table

```
<<linker tables: noinit bss zero>>=
_noinit_bss_table_start_ = . ;

LONG(ADDR(.noinit_bss))
LONG(SIZEOF(.noinit_bss) / 4)

_noinit_bss_table_end_ = . ;
```

An external declaration of the `noinit_status` variable, used as part of the strategy for preserving some memory over a reset, is included. It's not really a linker generated symbol, but this is a convenient place to put it.

With all the copy table descriptors defined, placing sections into the output can be continued.

Initialized data section

Previously, the difference between the VMA and LMA for a section was discussed. The `.data` section is distinctive because the section data is loaded at one address and linked to be run at a different address. The linker AT operation sets the LMA for a section.

```
<<linker section: data>>=
.data : ALIGN(4)
{
    _data_start_ = . ;
    *(.data*)

    . = ALIGN(4) ;
    _data_end_ = . ;
} > RAM AT> FLASH
```

The previous script fragment uses the AT operation to set the LMA of the `.data` section to be at the value of the `data_lma` variable, but the VMA is placed in RAM.

Stack section

The first RAM section is for the Process Stack. A significant difference for the stack section is the need to align it on a 64-bit boundary for AAPCS reasons. The stack is just an area of memory, contains no data, and is initialized at start up, the `NOLOAD` section type is used to inform the linker there is nothing to place into the executable file.

```
<<linker section: process stack>>=
.process_stack (NOLLOAD) : ALIGN(8)
{
    _process_stack_limit_ = . ;
    KEEP(*(.process_stack*))
```

```

. = ALIGN(8) ;
__process_stack_top__ = . ;
} > RAM

```

The Main Stack is placed at the next RAM location.

```

<<linker section: main stack>>
.main_stack (NOLOAD) : ALIGN(8)
{
    __main_stack_limit__ = . ;
    KEEP(*(.main_stack*))

    . = ALIGN(8) ;
    __main_stack_top__ = . ;
} > RAM

```

Both the Main and Process stacks are placed in the first locations of RAM for the following reasons:

- Since the stack grows down, should stack activity dip below the allocated space, there is no memory there and the SOC generates a Hard Fault exception. This is preferable to overwriting some other portion of RAM. Since the Process Stack is more difficult to estimate, it is placed first with the Main Stack following.
- The first 64 KiB of RAM are TCM RAM. Since stack areas are accessed frequently, it is advantageous to place the stack in TCM RAM.

Uninitialized data section

The `.bss` section is similar to the stack. There is nothing to place in the executable so it is designated as a `NOLOAD` section. Unlike the stack, there are variables allocated by the program code in the `.bss` section and the linker must resolve those references in the code. It is also traditional to place the `COMMON` data in the `.bss` section. Common data is an historical remnant of past programming practices, but it is conventional to include it in the linker script.

```

<<linker section: bss>>
.bss (NOLOAD) : ALIGN(4)
{
    __bss_start__ = . ;
    *(.bss*)
    *(COMMON)

    . = ALIGN(4) ;
    __bss_end__ = . ;
} > RAM

```

Non-initialized data section

After placing the `.bss` section in RAM, the same type of placement is used for the `.noinit_bss` section. This section is zeroed by the start up code only upon power on reset.

```

<<linker section: noinit data>>=
.noinit_data : ALIGN(4)
{
    __noinit_data_start__ = . ;
    *(.noinit_data*)
    . = ALIGN(4) ;
    __noinit_data_end__ = . ;
} > RAM AT> FLASH

```

Linker symbols are necessary to support the copying of initializer values. Note that unlike the initializer copying for the `.data` section, only a single block of `.noinit_data` section initializers is supported.

Non-initialized bss section

The final RAM section is similar to `.bss`. The difference, [as seen previously](#), is how the area is initialized at start up allowing values to be preserved across system reset.

```

<<linker section: noinit bss>>=
.noinit_bss (NOLOAD) : ALIGN(4)
{
    __noinit_bss_start__ = . ;
    *(.noinit_bss*)
    . = ALIGN(4) ;
    __noinit_bss_end__ = . ;
} > RAM

```

main at last

Finally, a trivial program to resolve the `main` symbol can be written.

```

<<path-to-main-test.zig>>=
//! The `path-to-main-test` module contains a trivial test program that
//! runs the code from the "Path to Main" chapter in "Code to Models".
<<edit warning>>
<<copyright info>>

comptime {
    _ = @import("start_main"); ❶
}

pub fn main() !void {
    return error.MainReturned;
}

```

- ① The `start_main` module is required to transfer control properly to `main`.

Testing

As a first step in starting the processor, manual testing and observation is necessary. At this point, there is no infrastructure in place that would allow reporting the test results and, consequently, no means to make an evaluation of whether the actual results matched the expected results.

Manual inspection through the debugger shows:

- The exception vector table is as expected. The first entry is the top of the stack. Next comes the Reset Handler. The remaining exceptions all point to the Default Handler.
- The two stack areas have been initialized with an end marker and a repeating pattern.
- The value of the `systemCoreClock` is 48 MHz, as expected.

The main program has no statements. If execution continues, returning from `main`, the startup code invokes panics.

Summary

In this chapter, the executable code and scripting necessary to get an Apollo 3 SOC to initialize an execution environment so that an application can begin at `main()` has been shown. This is certainly not the only way to accomplish this goal. There are a large number of ways to get to `main()` and the implementation of a startup sequence, like anything else in software, needs to meet the requirements of the target system.

This design has several limitations which more experienced readers will have noticed.

- There is no support for C` programs. The `C language imposes additional requirements which must be accomplished at initialization time and additional start up code and linker sections are needed. Since there is no intention to include any C++ code for this project, the startup sequence was simplified.
- Finally, experienced readers will have noticed that there is no provision for a system wide heap for dynamic memory allocation. The use of system heaps in microcontroller based systems is a subject that has generated considerable discussion in the greater microcontroller community. Some hold the opinion that it should not be used at all. There are good reasons, such as fragmentation and recovery potential, not to use a system-wide heap. Others hold that heap allocation during initialization is acceptable but not afterward. Still others see no problems using a system-wide heap. For the type of applications targeted here, e.g. those which are long running and have deterministic requirements, no system wide heap is used and its negative aspects are avoided. When there are specific situations that require dynamic memory allocation, that allocation is managed with worst case, compile-time-allocated memory blocks tied to the data type of the allocated memory objects.

[1] It is common for microcontrollers to use a bootloader to download and flash programs. In those cases, it is the bootloader that gets control of the processor after power up. When control is transferred to the downloaded program, we will have set the entry point to be the reset exception handler.

[2] ARM DUI 0553A, ID121610, p2-24

[3] Apollo 3 MCU Datasheet, Doc. ID: DS-A3-1p3p0, p.74

[4] Whatever finished means in software.

[5] The whole truth is more detailed. Instruction set variations exist because of optional core components such as the FPU. Here you are concerned with the ARMv7E-M instruction set, including the optional single precision floating-point instructions.

[6] There is one exception, to this circumstance, `SVCall`, and that is discussed later.

[7] Technically, the value is one past the last byte allocated to the stack. The stack pointer is always pre-decremented before

it is used to store a value on the stack

[8] The term is slightly confusing since there is no *virtual* memory for this microcontroller as there might be for a processor with a Memory Management Unit (MMU). But a MMU serves to make some place in memory appear to the program as if it a physical address.

Crossing the Divide

The [previous chapter](#) discussed how running thread mode computations in unprivileged mode effectively erects a barrier between handler mode and thread mode. This chapter shows how foreground and background activities cooperate to transfer control and data between the two sides. This mechanism is central to the design of how the system responds to its environment.

Handler Mode / Thread Mode Split

Only handler mode is allowed^[1] to access system and peripheral registers. The only way that handler mode is entered is by an exception.

This chapter shows the design elements needed to work with the split between privileged and unprivileged code. To be clear, the split between handler mode and thread mode is intended to divide the area of system concerns between those aspects which monitor and interact with the external environment from those which are associated with the logic of an application's response to the environment. Since the core architecture provides the mechanism to handle this split, minimal additional software execution is required to provide the separation.

Separating and confining different system concerns is another recurring topic in this book.

There are four mechanisms used to enter handler mode.

Interrupt requests

Peripheral devices can cause can *pend* an interrupt. The processor core then arbitrates between pending interrupts to select the order in which they are made active.

System faults

The system can detect certain faulty behavior of both the hardware and software to cause an exception.

Thread mode request

The Cortex-M architecture provides the SVC instruction as the primary means for thread mode software to create an exception. The exception created by the SVC instruction is used to request services from handler mode software.

Handler mode request

The other architectural exception mechanism provided is the PendSV exception. The intended use of this mechanism is to have handler mode software request additional handler mode services. The exception can be used to defer computations which started in handler mode to a lower priority exception. The usual use case allows a higher priority interrupt to defer additional work to a lower execution priority, allowing an interrupt to be split into multiple parts. Continuing additional hander mode processing at a lower priority prevents blocking other, potentially high priority exceptions, any longer than is necessary. This mechanism is sometimes used for switching between virtual execution contexts (e.g. task context switching). This design does not the PendSV mechanism, but some suggestions are made as to potential uses in this context.

Since the goal is to build a *reactive system*, all actions of the system are ultimately initiated by handler mode activities. Even periodic time based actions are ultimately driven by an interrupt from a timer peripheral. This section gives the design of the how handler mode exception processing is managed. In another section, the manner in which thread mode computations are managed is shown.

Handler mode computation has the following properties.

- Handler mode is the only provided mechanism that can preempt thread mode activities. In keeping with a reactive systems focus, handler mode preemption allows the system to track and react to the environment at speeds governed by the environment itself. This design does *not* include any mechanism which allows thread mode computations to preempt other thread mode computations, *i.e.* this design does not create any execution contexts beyond those provided by the processor core.
- Handler mode computation runs to completion and there is no waiting or pausing. Computations which require synchronization between different parts of the software are run in thread mode.
- Handler mode notifies thread mode activities of the occurrence of changes in the environment and can *push* data and control to thread mode activities.
- Conversely, thread mode activities can only affect the system environment by making requests to handler mode actions which ultimately involve peripheral control with the possibility of *pushing* data and control into the environment.

Exception Priorities

It is a goal of this design to manage the order and sequencing of computation in the software strictly by manipulating the execution priority of the software components using the mechanisms provided by the core to configure execution priority. By using the mechanisms directly supported by the core, the amount of additional software execution required just for deciding which code is to run next is lessened. In this section, the priorities assigned to the various architectural mechanisms provided by the ARMv7e-M architecture are described. The Apollo 3 Blue SOC provides 3-bits of exception priority, as is quite common and is the minimum defined by the core architecture. Three bits equals 8 priority levels. Exception priorities are discussed in terms of these 8 levels but the same design considerations apply independent of the number of priority levels. Staging the discussion in terms of the specifics of an 8 level scheme makes more concrete the choice of design elements supported by the core which match the system goals.

As a further simplification, this design does *not* use *sub-priorities*. Sub-priorities direct the choice of which exception to run when there are multiple pending exceptions at the same *priority*. Without sub-priorities, the processor chooses the smallest IRQ number from among simultaneously pending requests which have the same priority. The mechanisms provided by the ARMv7e-M architecture are intended to support a large number of different system design goals. Here, there are specific requirements that can be met using a less complicated arrangement of core features.

The following describes the rationale for the assignment of exception priorities.

- All system faults which have a configurable priority run at the highest priority. If the core detects a problem, it is desirable to act upon it at once. The Reset, NMI, and Hard Fault exceptions have assigned, fixed priority that are always higher than any configurable priority.
- Provisions are made for a debug monitor priority which is less than a system fault. An *in-system* debug monitor has a number of uses. We do not pursue building an *in-system* monitor in this book, but hold out the prospect for work.
- All interrupt requests happen at the same priority. This is a *flat* interrupt priority scheme. Although ARMv7-M core supports using priority to *nest* interrupt service, that complication is avoided in this design. As shown later, the assignment of priorities in this design allows for a use case with nested of interrupts, but that capability is not used here based on complexity arguments alone.
- The SVC priority is lower than that of an interrupt request and is the lowest priority exception considered. This choice means that IRQ's can preempt the SVC handler. This requires taking more care in dealing with peripheral device access, but it implies that interacting with the

environment has higher priority over requesting system services.

The following table shows the prioritization scheme. To reiterate, a lower priority level preempts a higher priority level and the priority number accounts for the number of priority bits supported by the core and the location of the bits in the system registers.

Priority level	Priority number	Description
0	0x00	All configurable system fault handlers.
1	0x20	Unused.
2	0x40	Reserved for debug monitor.
3	0x60	Unused.
4	0x80	Unused.
5	0xA0	All peripheral device interrupts.
6	0xC0	SVC exception.
7	0xE0	Unused.

Table 2. Exception Priority Assignments

The following priority levels are not used by this design, but some possible use cases are described.

- Priority 1** can be used if an interrupt is needed to work in conjunction with the debug monitor. For example, if the debug monitor is receiving input from a UART, it may be more convenient to handle the UART with interrupts and those interrupts would need to have a higher priority than the debug monitor. Alternatively, it may be necessary for some peripheral devices to continue to be serviced even when a debug monitor is running.
- Priority 3 - 4** can be used to implement a nested interrupt scheme for those systems which have requirements where a flat scheme may not be sufficient. Note that nested interrupt schemes require careful design to get correct and make the system more difficult to reason about. Also note that there are specific code sequences given later which depend upon flat IRQ prioritization. Those situations are pointed out as they arise.
- Priority 7** can be used for the PendSV exception. This design does not use PendSV, but it might be needed by systems which have a more complicated nested interrupt structure and which might wish to defer some of the actions for a particular interrupt to a lower exception priority. One can envision a queuing scheme aimed at the PendSV exception handler that is used to sequence lower priority privileged execution.

Initializing Exception Priorities

With three bits of priority available and sub-priorities not being used, the priority group number is four. The number four defines where an implied radix point is placed, *i.e.* between bits 4 and 5, to separate the exception priorities from sub-priorities. The fact that the priority numbering scheme uses the high order bits of the priority number takes some getting used to, but works well in the face of SOC's supporting different numbers of priority bits. So, priority group 4 has three bits of priority and any remaining priority bits supported by the core (zero in our case) are devoted to sub-priorities. The remaining constants are as discussed above.

Since this is a particular configuration of core usage, we defined the constants as part of the system configuration of the design.

```
<<system_config: constants>>=
/// Priority group 4 implies 3 bits of priority and 0 bits of subpriority
/// for a core that has 3 total bits of NVIC exception priority.
pub const priority_group: v7m.PriorityGroup = 4;
pub const exception_sub_priority = 0; // Sub-priorities are not used in this design

pub const sys_handler_priority: v7m.ExceptionPriority = 0;
pub const sys_handler_encoded_priority: v7m.ExceptionPriority =
    v7m.encodePriority(priority_group, sys_handler_priority, exception_sub_priority);

pub const debug_monitor_priority: v7m.ExceptionPriority = 2;
pub const debug_monitor_encoded_priority: v7m.ExceptionPriority =
    v7m.encodePriority(priority_group, debug_monitor_priority, exception_sub_priority);

pub const irq_priority: v7m.ExceptionPriority = 5;
pub const irq_encoded_priority: v7m.ExceptionPriority =
    v7m.encodePriority(priority_group, irq_priority, exception_sub_priority);

pub const svc_priority: v7m.ExceptionPriority = 6;
pub const svc_encoded_priority: v7m.ExceptionPriority =
    v7m.encodePriority(priority_group, svc_priority, exception_sub_priority);
```

```
<<system_apollo3: imports>>=
// Specific constants relating to how the Apollo 3 microcontroller
// is configured for use in this particular case.
const system_config = @import("system_config.zig");
```

In [Chapter 2](#), we deferred the discussion of initializing exception priorities. We are now ready to show the implementation.

```
<<system_apollo3: systemExcPriorityInit>>=
// The `systemExcPriorityInit` function initializes the configuration of
// system and IRQ handlers. The chosen priority levels insure that
// system exceptions have highest priority, all IRQ exceptions have the
// same priority, and the SVC exception has the lowest.
fn systemExcPriorityInit() void {
    const scb = v7m.scb;

    scb.setPriorityGrouping(system_config.priority_group);

    scb.setExceptionPriority(.mem_manage, system_config.sys_handler_encoded_priority); ①
    scb.setExceptionPriority(.bus_fault, system_config.sys_handler_encoded_priority);
    scb.setExceptionPriority(.usage_fault, system_config.sys_handler_encoded_priority);

    scb.setExceptionPriority(.debug_monitor, system_config.debug_monitor_encoded_priority);

    scb.setExceptionPriority(.sv_call, system_config.svc_encoded_priority);
```

```

var irqn: v7m.IrqNumber = 0;
while (irqn < apollo3.irq_count) : (irqn += 1) {
    v7m.nvic.setIrqPriority(irqn, system_config.irq_encoded_priority); ②
}

scb.shcsr.updateFields({ ③
    .bus_fault_ena = 1,
    .usg_fault_ena = 1,
});
apollo3.mcuctrl.fault_capture_en.setBitField(.fault_capture_en); ④
}

```

- ① The encoded priority numbers are computed at `comptime` to produce the proper number to use as an argument to `setExceptionPriority` or `setIrqPriority`. And yes, it is all a bit confusing but is the price for considerable flexibility.
- ② By setting all the peripheral interrupts to the same level, any missing peripheral device interrupts that are inadvertently triggered cause the `systemMissingException()` function to be executed at the same priority as all the other IRQ's.
- ③ Enable the Bus Fault and Usage Fault exceptions. We are determined to track down all the faults that the core can diagnose.
- ④ Enable all the fault related exceptions and the Apollo 3 specific fault capture mechanism.

Foreground / Background Interfaces

In the last section, the exception prioritization scheme that orders preemptive execution was described. Exception processing was discussed using ARMv7e-M architectural terms such as Thread Mode and Handler Mode. From here onward, the older more generic terms of *foreground* and *background* are used. For the ARMv7e-M architecture, *foreground* execution is defined as the processing which happens in Handler Mode, and *background* execution as the processing which happens in Thread Mode. Other processor architectures may not have the same configurability as the ARMv7e-M architecture, but many of the same considerations apply, even when reduced to the simple two mode scheme of interrupt and non-interrupt execution where no priority configuration is possible.

This section describes how the division between foreground and background processing is coordinated. Design concepts are presented for how background processing can access privileged services and how foreground processing can notify background processing of happenings in the system environment.

Climbing Toward the Foreground

Since a barrier between the background and the foreground has been erected, it is necessary to design a mechanism to allow the two sides to interact. The design concept used is that of a *proxy*. When background processing needs to control devices or aspects of the system execution, it makes a request to the foreground to obtain the required service. Conceptually, the request is similar to a message. Fortunately, the mechanism operates within a single processor core, and simpler memory-based mechanisms are used to deliver the requests.

A function executing in the background makes a request to a *proxy function* which executes in the foreground to fulfill the request. The proxy function can be viewed as a continuation of the background request which is executed with privilege. The SVC instruction is the core architectural mechanism used to enable the background request to obtain service from the foreground proxy. The

following figure shows a simplified schematic of how background requests are routed to foreground functions acting a proxy for the request.

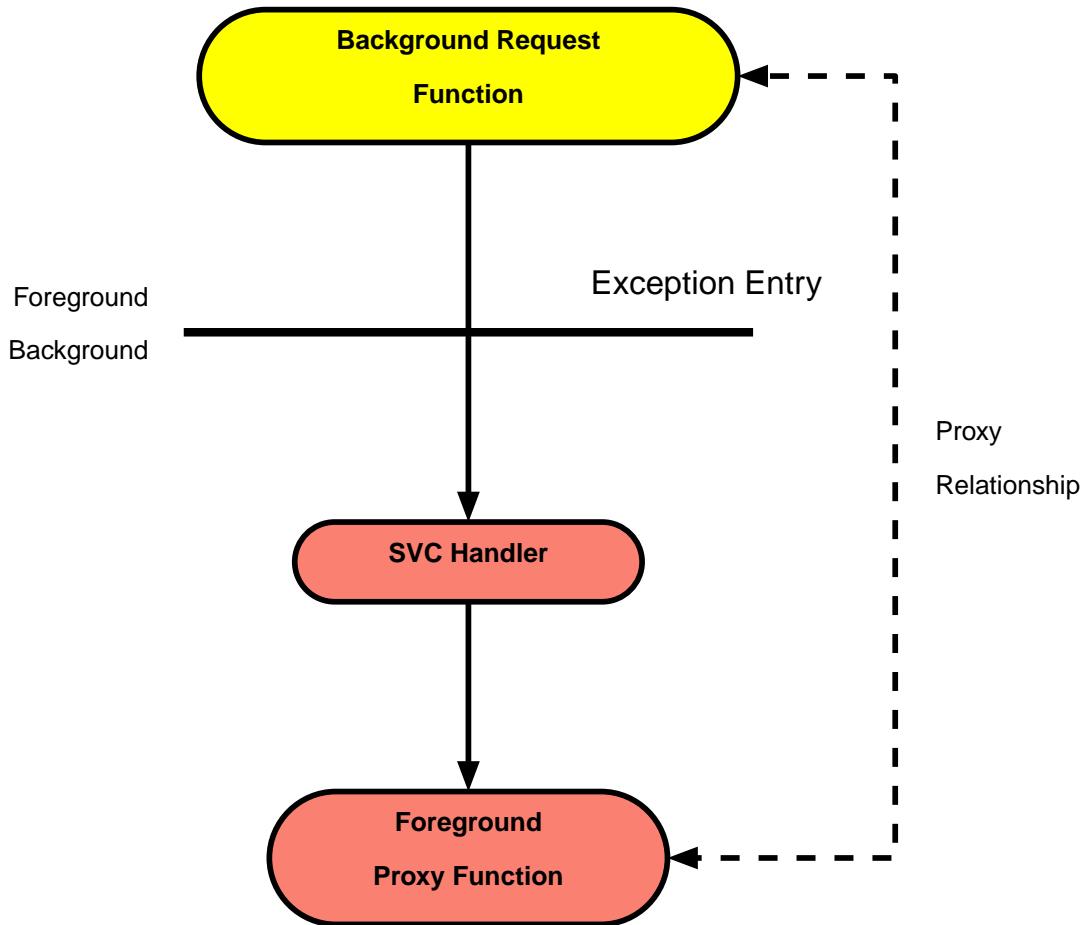


Figure 3. Overview of Proxy Relationship

In this figure, the yellow box represents a request from background processing for a service which requires privilege to execute. The red boxes represent foreground processing. Control is transferred from the **Background Request Function** to the **Foreground Proxy Function** by means of the SVC exception which is handled by the **SVC Handler**. There is a one-to-one correspondence between the **Background Request Function** and its **Foreground Proxy Function** counterpart. The request may provide arguments to control aspects of the request and the Foreground Proxy function may return data as part of fulfilling the service request.

The scheme shown in the previous figure is overly simplistic for realistic programs. The number of request functions is large considering that all peripheral device access and other “device driver” type operations must be performed in the foreground. Peripheral device control is a major part of any microcontroller system and an organization for how device control software is accessed must be designed. A long, unstructured enumeration of function names is not a sufficient organization for the foreground proxy functions. To make the groupings smaller, a hierarchy is used, as is common practice, and the background requests are divided into subcomponents related by an hierarchical fashion.

The first subdivision in the foreground organization is the concept of *service realms*. There are two service realms in this design.

1. System realm
2. Device realm

These two realms differ by their ability to issue background notifications. Functionality to manage processor core resources and control execution sequencing is deemed part of the *system* realm. The system realm does not have any background notification requirements. Asynchronous activity arising from problems detected by the processor core is delegated to system fault handlers and does not require any background execution to resolve. Since this design considers system faults as fundamentally unrecoverable, no mechanism is provided for the system realm to direct asynchronous background execution. All system realm requests execute synchronously and to completion.

Functions associated with peripheral devices are deemed part of the *device* realm. In the device realm, the requests are partitioned first by *device class* and second by *device operation*. The class of a device is a logical grouping of one or more physical peripheral devices which act in concert together. The logical device supports a set of device operations, that are, in general, specific to the class. A device class may also have multiple *device instances* which all operate the same and according to the defined device operations. Device realm services may require subsequent background notifications. Some device realm services execute synchronously, but many do not. The asynchronous services use the background notification queue to inform the background processing of any detected changes in the environment that the device supports. Some other terms used for device class and device instance are major device number and minor device number.

The following figure expands upon the previous figure by including the division into service realms and, for the device realm, the further subdivision into device classes and device operations. This gives a more accurate picture of the proxy relationship between background requests and foreground proxy functions when they involve different service realms.

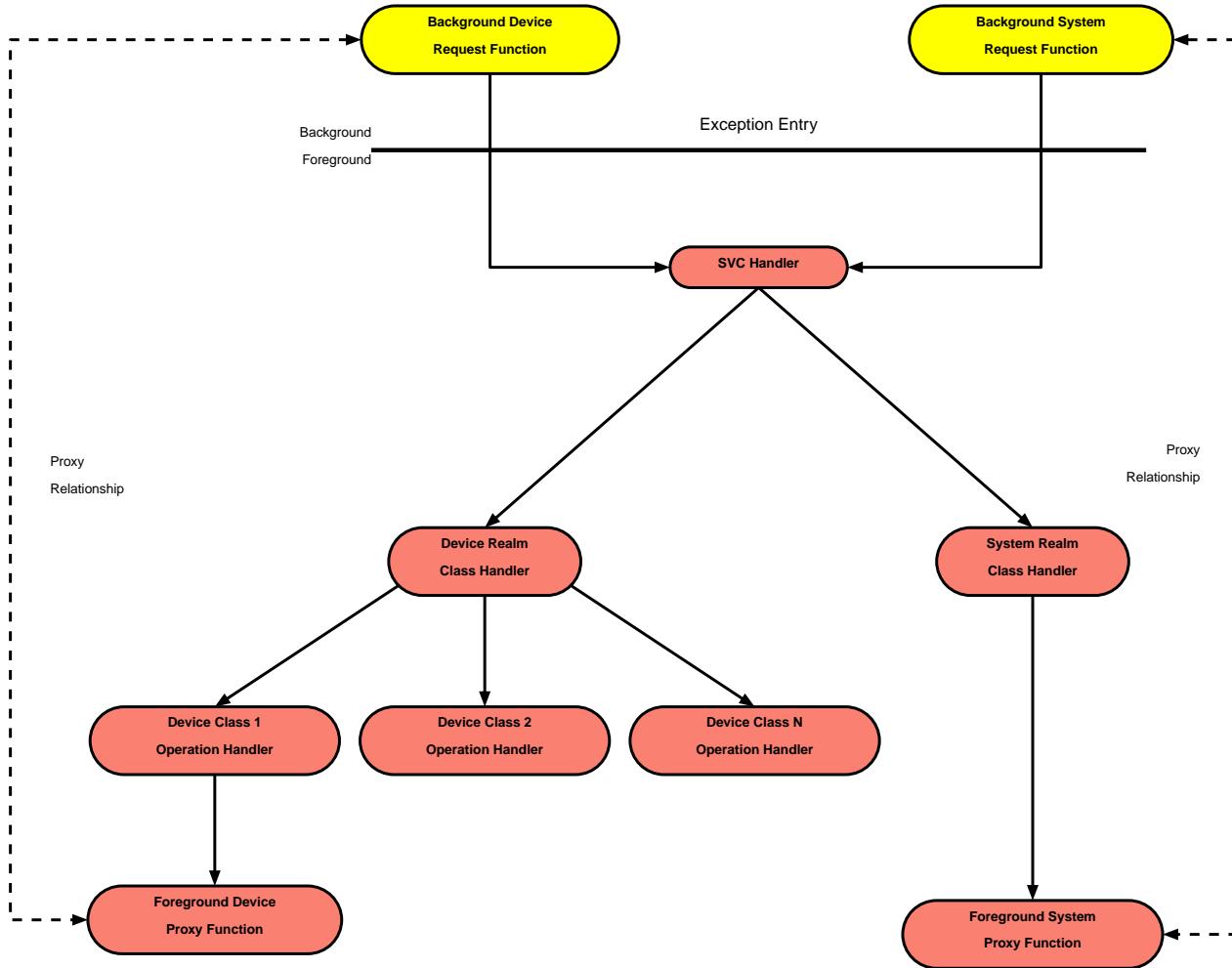


Figure 4. Proxy Relationship with Service Realms

In this figure, the two service realms have been added. The **SVC Handler** decides to which realm the request is directed. System realm requests are dispatched using only a single level of decision. Device realm requests are dispatched first to a particular device class and then, in a second level, the requested operation is dispatched to a proxy function. The device instance is passed along as a parameter to the proxy function so it may choose the particular hardware instance on which it operates.

Climbing Toward the Background

In the previous section, the design concepts for transferring control and data from background processing to foreground processing were discussed. It is necessary to provide mechanisms which traverse the barrier in the other direction. It would be convenient if the traversal from foreground to background were symmetric with the background to foreground direction just shown. However, it is not.

Recall that the initiation of all execution in the system ultimately arises from an interrupt request (IRQ). In general, an IRQ handler needs to inform the background of the occurrence of the interrupt and often must transfer data from the system environment to the background for further action. Much of the requirements for IRQ handlers interacting with the background derive from the desire to minimize the amount of time spent at IRQ level in order to provide timely response to other inputs from the environment. This is part of the currency of the environment which must be handled by a reactive system.

This design is driven by changes in the environment which are transported into the system by interrupts. The processor core invokes the IRQ handler which is then responsible for starting the computations necessary to react to the environment. The required work to fully realize the effect of the interrupt is typically split between foreground processing and background processing. It is an important design consideration to determine how the required computations are split. Platitudes such as, “Interrupt handlers should be as short as possible,” give no practical guidance for determining where the processing associated with an interrupt occurs.

At a minimum, an IRQ handler must:

- Re-arm the interrupt for the next occurrence. Interrupts are not a one-time happening. The source of the interrupt must be made quiescent so that the next time the interrupt is asserted the system will recognize another instance of the interrupt.
- Capture any state of the environment which may be present in the registers of a peripheral device. For example, a UART receive interrupt needs to read the received character out of the device. Otherwise, the UART receiver may overflow and any message composed of the received bytes would be corrupted.
- Move data or control into the environment. For a UART transmit interrupt, data from the system must be moved into the peripheral device registers for transmission.
- Recognize semantically significant conditions and initiate the processing to complete the required function. Consider the case where data is obtained from the environment in block units. When a complete block has been received, background processing must be notified that a received block is ready for the additional processing required to produce the system response.

Because the processor can only do one thing at a time and because while handling an IRQ nothing else can execute^[2], to remain responsive to the environment, it is desirable to have the execution time of an IRQ handler be small. Of course, IRQ handlers always take some time to execute. It is important to recognize that the longest running IRQ handler determines the worst case response latency of the system for detecting changes in the environment.

The detailed manner in which a logical device operates and the manner in which an IRQ handler contributes to the overall device operation is a creative output of its design. There is considerable flexibility as to where processing is allocated and practical patterns have been established for where the split in processing occurs. However, it is a goal to have only one way to notify the background processing of conditions detected by the IRQ handler. To accomplish this goal, it is necessary to define a mechanism for IRQ handlers to generate background notifications that are used by background processing to compute the system response.

Because the environment of the system operates asynchronously to the program and because the timing of happenings in the environment cannot be predicted, a queuing mechanism is used to allow IRQ Handlers to submit notifications. Background processing is not synchronous to foreground interrupts. The following figure shows an overview of the background notification scheme.

Handler Mode Only Processing

For certain types of systems, typically small systems which act as a simple bridge between components, it is possible that all processing can be done in the foreground. This type of design eliminates any background processing and all IRQ handling runs to completion to generate the system response. It is a common enough design that the ARMv7-M architecture supports a special sleep mode. Setting the SLEEPONEXIT bit in the System Control Register causes the processor to enter sleep (or deep sleep) upon returning from an IRQ exception. Thus if the longest service time of an IRQ is within the bounds of the desired response latency of the application, it is possible to build a system composed entirely of IRQ handlers.

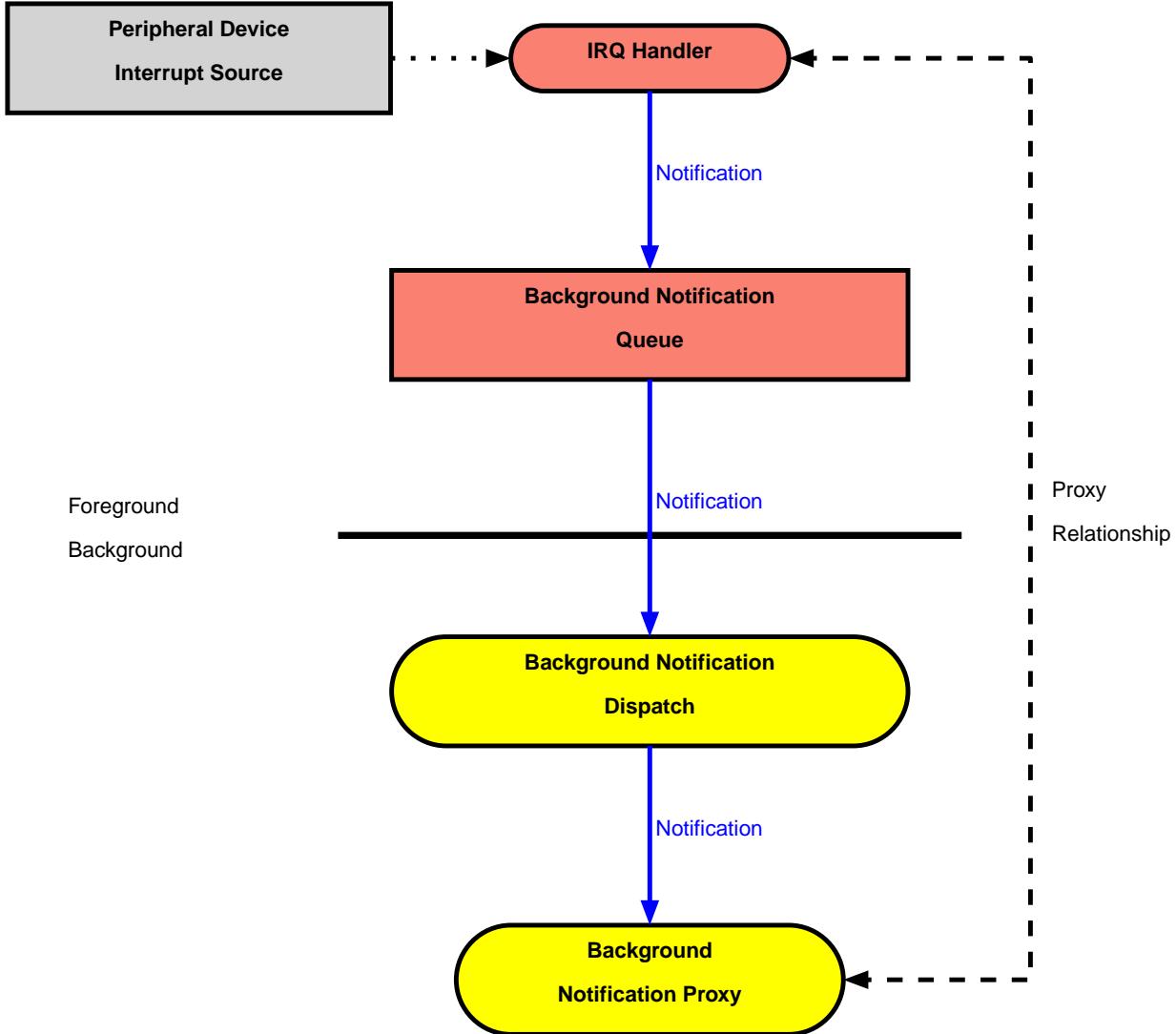


Figure 5. Background Notification Overview

This overview shows the relationship between an **IRQ Handler** and the **Background Notification Queue**. If an IRQ Handler needs to notify background processing of a happening in the environment, it inserts a notification into the **Background Notification Queue**. The **Background Notification Dispatch** code removes queued notifications and dispatches them. The dispatch results in executing a **Background Notification Proxy**. The IRQ handler and its notification information have a proxy relationship with the **Background Notification Proxy**. The **Background Notification Proxy** can be viewed as a continuation function of the IRQ handler. The binding of a background notification proxy and a logical device is made at run time, *i.e.* device realm requests associated with devices that support some form of notification accept parameters to specify which background proxy function is to handle the notifications generated by the device. Note that there may be multiple **Background Notification Proxy** functions for any given logical device, depending upon the design of the operations of the peripheral device. The flexibility given by binding a logical device to a **Background Notification Proxy** at run time makes it much easier to run different applications on an existing base of device realm code, *cf.* the binding of background requests to foreground proxies discussed in the last section where the binding is done at compile time. The compile-time binding of background requests to foreground proxy functions implies that the capabilities of a device are known when a system is built and the interfaces to those capabilities are, consequently, also fixed.

The previous figure is also a simplification of the foreground/background interaction. A mechanism

is required for background processing to obtain the notifications for dispatch. As shown in the diagram, the **Background Notification Queue** is part of the foreground processing and thus requires privilege to access. The only way for **Background Notification Dispatch** to obtain a notification is to issue a system realm request to obtain one. The following diagram shows the details of using a system realm request to obtain a background notification placed in the **Background Notification Queue**.

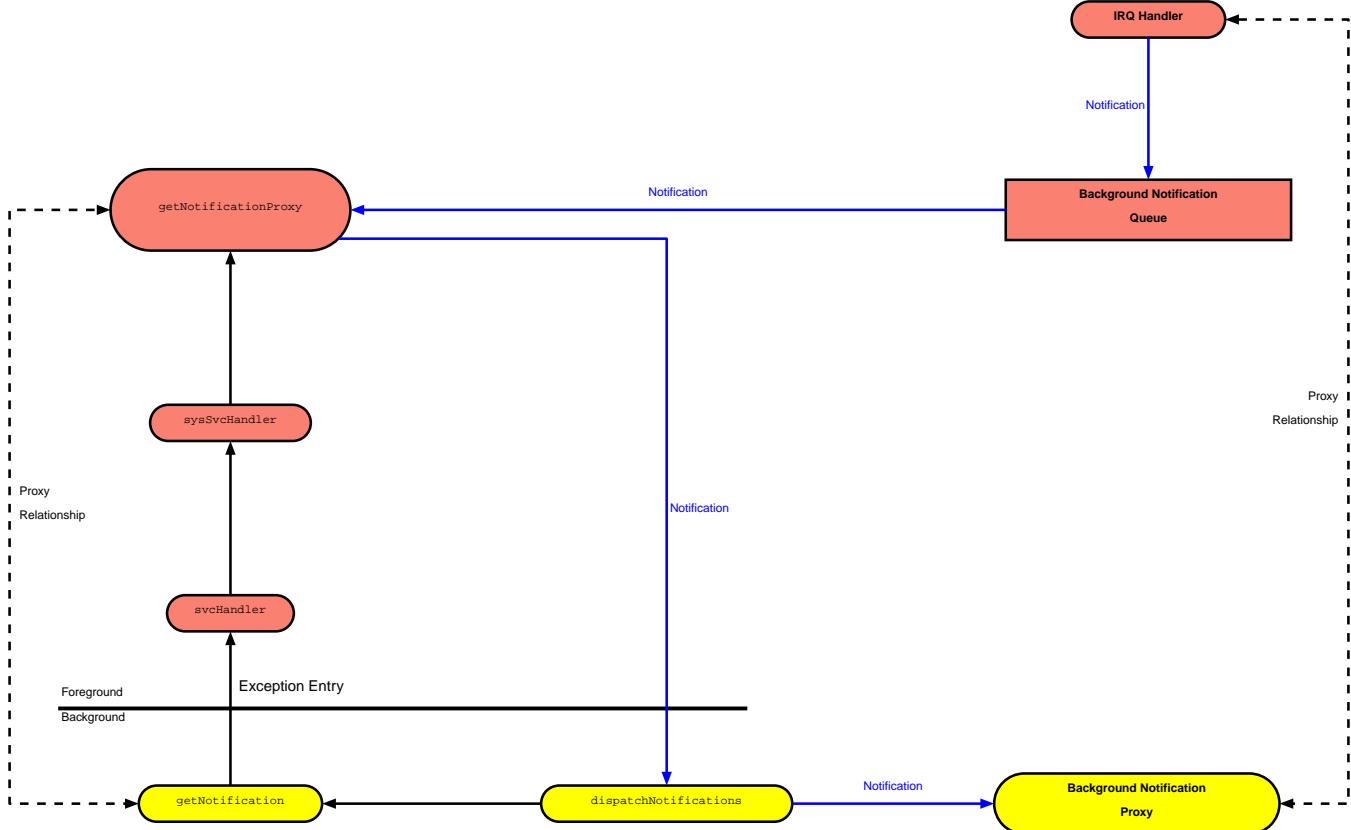


Figure 6. Background Notification Retrieval

In this figure, a system realm request is used to obtain a notification from the **Background Notification Queue**. The proxy relationship between the IRQ Handler and the **Background Notification Proxy** is established (at run time) prior to enabling any IRQ for the device (typically at device “open” time). When the **IRQ Handler** is run, it places a notification in the queue. **Background Notification Dispatch** then uses the `getNotification` function to request notifications from the queue. Notifications are copied from the privileged foreground memory to the unprivileged background memory by privileged execution using unprivileged load and store instructions. Once copied, the `dispatchNotifications` function invokes the appropriate **Background Notification Proxy**. It is important that data flow is from privileged execution to unprivileged execution.

In the next sections, the design and code for the components of the foreground/background interface are shown. First, the SVC exception handler is considered. The discussion is focused on its major role as the conduit for the flow of control and data between the foreground and background arenas. Second, the details of the mechanism by which background requests reach their corresponding foreground proxy functions are shown. Third, the design of how IRQ handlers queue notifications to the background, including how the background notification queue operates, is shown.

SVC Interface

The SVC instruction is the only means provided by the processor architecture for unprivileged code to request privileged service^[3]. This implies that the SVC handler must be able to pass input parameters and return results across the foreground/background boundary. Typically, exception handlers do not perform that function and it requires some intricate programming to make happen. Specifically, the interface must:

- Pass data into the foreground service realms as arguments. The argument passing conventions of the AAPCS must be used. The AACPS, in its most general case, is a complex set of conventions. This design adheres to the basic rules which pass the first four word sized arguments to a called function in the R0 - R3 registers.
- Return data back to the background as the results of a request.
- Have the functions which provide the privileged service in the foreground be invoked as ordinary Zig functions using the parameter passing conventions of the AAPCS. Specifically, it is a goal to avoid function interfaces which have, as an argument, a pointer back into a stack frame. Such a design would “color” all such functions to have detailed knowledge of the exception frame layout. It is undesirable to force the foreground proxy functions to be aware of and tied to the layout of processor resources such as stack frames. This is deemed an unnecessary cognitive burden for writing foreground proxy functions.
- Segment the foreground processing into the service realms previously discussed.
- Make all argument passing and returned results of the SVC handler interface agnostic to the data types of the values being passed. This makes the SVC handler strictly a control/data transportation mechanism with no knowledge of the meaning or use of the passed data values. The goal in this case is to make the SVC interface easier to extend to service realms other than the two we have discussed.

There are five data values at which can be used in the SVC interface design:

1. The value of the immediate operand of the SVC instruction itself. Part of the instruction encoding is an 8-bit value which can be chosen as necessary.
2. Four 32-bit values contained in the R0 - R3 registers.

The SVC interface design allocates the five data values as follows:

SVC immediate operand

Selects between the system realm and the device realm. The system realm is encoded as 0 and the device realm as 1. No other immediate operand values are supported.

req \Rightarrow R0

A request identifier. The interpretation of the req value is determined by the individual service realm handlers.

input \Rightarrow R1

A pointer to the input arguments, if any, of the operation. Arguments to the foreground proxy function are marshaled into a memory object and passed by pointer reference.

output \Rightarrow R2

A pointer to where the output of the operation, if any, is returned. The pointer is to a memory object where output results from the proxy function are placed.

err → R3

A pointer to where auxiliary error information, if any, is returned. The pointer is to a memory object where additional information about the cause of an error may be placed.

Only the `req` argument is required. The pointers may be passed as `null` if the proxy operation does not require additional arguments of that type. Argument values passed through the SVC exception have no type interpretation. Data typing is lost upon exception entry and must be regained by the invoked foreground proxy function.

The return value of the SVC interface is an enumeration.

```
<<system_services: interface definitions>>
/// The `SvcResult` type defines the possible result values that indicate
/// the outcome of a service call invocation.
pub const SvcResult = enum(u32) {
    /// The request was performed successfully.
    success = 0,
    /// The realm to which the request was directed is unknown, _i.e._ the request
    /// failed to be identified as either a system realm request or a device realm
    /// request.
    unknown_realm,
    /// The class of the device to which the request was made does not exist.
    unknown_device,
    /// The operation requested is not known to the device class.
    unknown_operation,
    /// The instance of the device to which the request was made does not exist.
    unknown_instance,
    /// There was an error in a passed parameter.
    invalid_param,
    /// The operation implied by the request failed. For some requests, the error
    /// return information contains more specific indications of the failure.
    operation_failed,
    /// The operation did not complete and may be retried at another time.
    retry_operation,

    /// The `mapSvcResultToError` function converts between `SvcResult` values and
    /// `SvcError` errors.
    fn mapToSvcError( ❶
        result: SvcResult,
    ) SvcError!void {
        switch (result) {
            .success => return,
            .unknown_realm => return SvcError.unknown_realm,
            .unknown_device => return SvcError.unknown_device,
            .unknown_operation => return SvcError.unknown_operation,
            .unknown_instance => return SvcError.unknown_instance,
            .invalid_param => return SvcError.invalid_param,
            .operation_failed => return SvcError.operation_failed,
            .retry_operation => return SvcError.retry_operation,
        }
    }
};
```

- ① Values of `SvcResult` type are used across the SVC interface operations to indicate the status of the request.

For unprivileged function interfaces, *i.e.* for the application program requests, we prefer Zig error values to handle any errors. It is more convenient to handle the SVC results as a Zig error on the caller side. To do so, we provide corresponding error names and a function to map between them.

```
<<system_services: interface definitions>>
/// The `SvcError` type is an error set returned to callers of `SVC` services
/// indicating the outcome of the service invocation.
pub const SvcError = error{
    unknown_realm,
    unknown_device,
    unknown_operation,
    unknown_instance,
    invalid_param,
    operation_failed,
    retry_operation,
};
```

To accomplish the design goals for the SVC interface requires assembly language programming. The processing may be broken down as follows.

1. Determine the immediate operand value in the SVC instruction.

To obtain the immediate operand in the instruction, the return address which was placed on the stack during exception entry is examined. This step is complicated by needing to determine which stack (PSP or MSP) was in use when the SVC instruction was executed.

2. Using the value from the SVC instruction, select the proper foreground service *realm* handler.

A simple jump table is used to select the realm handler corresponding to the SVC instruction operand. It is also necessary to verify the value passed as the SVC operand to ensure it doesn't exceed the bounds of the jump table.

3. Copy the four input arguments from the exception frame into registers in accordance with the AAPCS.

Because of the *late arrival* and *tail chaining* optimizations of the processor, the values of scratch registers *cannot* be used directly. Late arriving or tail chained interrupts may change the values of the scratch registers from what they were when the SVC instruction was executed by the background processing. Since the AAPCS passes the first four arguments in the R0 - R3 scratch registers, their values must be obtained from the exception frame where they were placed when the SVC instruction executed and where they have not been disturbed in the case of late arrival or tail chaining of interrupts.

4. Invoke the appropriate realm service handler.

5. Upon return, copy the return value in the **R0** register back to the exception frame.

Since the exception return mechanism of the core restores the registers values from the stack, any return value must be copied out of the R0 register onto the stack and then, from the exception frame on the stack, the values are loaded back into the registers by the processor as part of the exception return.

6. Return from the exception back to unprivileged mode.

The following code implements the required interface. Detailed commentary follows the code.

```
1 <<svc handler>>=
2 /// The `svcHandler` function receives control through the vector table
3 /// when the `svc` instruction is executed. Service requests are dispatched
4 /// to either the system service handler or the device service handler.
5 fn svcHandler() callconv(.naked) SvcResult {
6     asm volatile (
7         "tst    lr,#4
8         ite    eq
9         mrseq ip,msp
10        mrsne ip,psp
11        push   {ip}
12        ldr    r0,[ip, #24]
13        ldrb   r0,[r0, #-2]    @ SVC immed operand
14        cmp    r0,#(2f-1f)/4  @ number of realms
15        ittt   cs
16        movcs r0,#1          @ SvcResult.unknown_realm
17        strcs r0,[ip]
18        bxc   lr
19        adr    r1,1f
20        ldr    r1,[r1, r0, lsl #2]
21        push   {r1}
22        ldm    ip,{r0-r3}  @ regs from exc frame
23        pop    {ip}          @ handler pointer
24        push   {lr}
25        blx    ip          @ handler call
26        pop    {lr}
27        pop    {ip}
28        str    r0,[ip]      @ return value
29        bx    lr
30        .align 2
31        .1:
32        .word  sysSvcHandler
33        .word  devSvcHandler
34        .2:
35    );
36 }
```

Line 5

The `.naked` calling convention requests the compiler *not* to include the usual function prologue and epilogue instructions. This compiler directive was used previously by the [Default_Handler](#).

Lines 7 - 10

This is the usual idiom to determine which stack was in use when an exception was activated. In this case, the proper stack pointer value is placed into the IP (aka R12) register. This is also a scratch register, so it may be used without preserving. Normally, it is used for veneers which are generated at link time when the offset for branching is larger than can be held as an immediate operand in an instruction. Here it serves as a useful place to hold the pointer to the working stack.

Line 11

Once the correct stack pointer has been determined its value is pushed on the stack. We will need it later when returning the result.

Lines 12 - 13

These instructions place the immediate operand value from the SVC instruction into R0. The value of the stacked program counter is offset 24 bytes into the exception frame. By stepping back from the return address by 2 bytes, the SVC immediate operand is found. It is 1 byte long and conveniently located in a single byte by itself. The immediate operand value is used as an array index into a jump table of request handlers.

Line 14

Compute the number of foreground realm service handler by address arithmetic and use the result as an immediate operand. The difference between the end of the jump table and its beginning divided by the size of each entry is the count of entries in the jump table. The 2f and 1f notation refers to the labels at lines 34 and 31, respectively. By comparing the number of entries in the jump table to the SVC immediate operand, it can be determined if the bounds of the jump table array are exceeded.

Lines 15 - 13

The previous cmp instruction sets the carry flag if the SVC operand is too large. Using conditional instructions, 1 (SvcResult.unknown_realm) is returned to indicate a bad realm designation. Note how the return value must be placed back into the exception frame at line 17.

Lines 19 - 20

Compute the address of the foreground realm handler function by indexing into the jump table. The table begins as the 1f: label (line 31). The index in R0 is scaled by 4 (lsl #2) to account for the number of bytes each pointer consumes, *i.e.* transform an element count into a byte count before fetching the handler function pointer from the jump table.

Line 21

R1 holds a pointer to the realm handler function. Before invoking the realm handler, we must load the handler arguments from the exception frame. But doing that would overwrite R1 where the handler function pointer has been placed. So it must be saved. Note it is always placed on the MSP since the processor is executing an exception handler.

Line 22

It takes only a single instruction to copy the stored values from the exception frame into the registers. Recall that IP contains the exception frame pointer value.

Line 23

Restore the value of the handler function in preparation to invoke it. The function pointer is placed in IP since its value does not need to be preserved and R0 - R3 currently contain the service arguments.

Lines 24 - 26

The conventional idiom to invoke a function when the address is in a register. The LR must be preserved because it contains the exception return value and LR will be overwritten by the BLX instruction.

Line 23

Restore the exception return value from the stack and place it back in LR. It is needed both as the return mechanism and to determine where the exception frame is located. Also, net stack usage is now zero.

Line 26

Restore the previously saved stack pointer. This is in preparation for the exception return.

Lines 27 - 28

Every proxy function returns only a single integer and that value will be in R0. We restore the stack frame pointer and use it to copy the service handler result back onto the exception frame. Other registers are purposely *not* copied back to the exception frame so as not to “leak” information to background processing.

Line 29

Branching to the contents of LR causes the return from the exception. Returning from the exception causes the exception frame values from the stack to be restored to the registers. This is why we were required to put the result of the service back into the proper location on the stack.

Lines 21 - 34

The definition of the jump table for realm handlers.

```
<<svc handler>>=
comptime {
    @export(&svcHandler, .{ .name = "svcHandler", .linkage = .strong });
}
```

Service Parameters

Each of the foreground proxy functions has to handle passed-in arguments. The structure of the arguments for any given proxy function is distinct for the service it represents. There is common code that can be factored out to obtain a consistent interface for the `input`, `output`, and `err` parameters that form the arguments to SVC requests.

When invoking a service request, the background request function must, in general, perform the following operations. Some of these operations are optional, depending upon the details of the operation being requested.

1. Any parameters to the background service function that are intended as inputs parameters to the foreground proxy are marshaled into an input parameter structure. **N.B.** by convention, any pointer to memory in the background that is to be used by the foreground proxy to return data is passed as an input argument. It is possible, and preferable for “reasonable” data sizes, to pass output data back from the foreground proxy by value. In that case, the output data would appear in the `output` parameter.
2. Declare any `output` and `err` parameters with default initialization. It is important that the default initialization be done in the background. That default initialization is used as a check to insure that data copied from the foreground to the background does not overwrite any background memory.
3. Invoke appropriate SVC operation for the intended service realm.
4. If any `output` or `err` values are returned, they must be unmarshaled from the service parameter structure.

The foreground proxy functions have the responsibility to copy parameter data across the privilege / unprivileged interface. From the foreground proxy viewpoint, the following operations must happen:

1. Receive any input parameters. This involves copying the `input` parameters using the provided

pointer from the background to the foreground.

2. Perform the request indicated by the request parameter.
3. Send any output or error values via the supplied pointers. This involves copying the data from the foreground to the background.

To support these common operations, a type function is defined. The returned data type has fields for the parameter information and a set of methods to perform the operations just described.

```
<<system_services: interface definitions>>=
/// The `DefineSvcRequestParam` function is a type function that defines the
/// detailed type of an SVC service parameter. The resulting type consists
/// of the common fields required of all service parameters plus that
/// given by the `Parameters` argument. The `Parameters` argument must be a `struct`
/// that contains any parameters that are specific to an SVC service.
pub fn DefineSvcRequestParam(
    comptime Parameters: type,
) type {
    if (@typeInfo(Parameters) != .@"struct") {
        @compileError("request parameters must be specified as a struct");
    }

    return struct {
        len: u8 = @sizeOf(Self),
        params: Parameters = undefined,

        const Self = @This();

        /// The `marshal` function returns a service parameter value initialized
        /// to contain the parameters given by the `params` argument.
        /// This function is used by background request functions to supply
        /// values for foreground service requests.
        pub fn marshal(
            params: Parameters,
        ) Self {
            return .{
                .len = @sizeOf(Self),
                .params = params,
            };
        }

        /// The `unmarshal` function returns the value of the given service parameters.
        /// This function is used by background request functions to obtain
        /// values returned by foreground service requests.
        pub fn unmarshal(
            self: *const Self,
        ) Parameters {
            return self.params;
        }

        /// All `output` and `err` parameters must be initialized before invoking
        /// the SVC request so that they may properly receive the result.
        pub const initial: Self = .{};

    }
}
```

```

/// The `receive` function copies a background request parameter into a
/// local memory object. The return value is a two-tuple containing the
/// parameter value and an `SvcResult` value which indicates the success
/// or failure of the copy operation.
pub fn receive(
    src: *const anyopaque,
) struct {Parameters, SvcResult} {
    const src_params: *const Self = @ptrCast(@alignCast(src));
    var dest: Self = undefined;
    const result = dest.copyIn(src_params);
    return .{ dest.params, result };
}

/// The `send` function copies a foreground value into a
/// local memory object of a background request.
/// The return value is an `SvcResult` value which indicates the success
/// or failure of the copy operation.
pub fn send(
    src: Parameters,
    dest: *anyopaque,
) SvcResult {
    const src_params = marshal(src);
    const dest_params: *Self = @ptrCast(@alignCast(dest));
    return src_params.copyOut(dest_params);
}

/// Copy into a local variable from an unprivileged source.
fn copyIn(
    dest: *Self,
    src: *const Self,
) SvcResult {
    const src_param_len = v7m.ldrbt(&src.len);
    if (src_param_len != @sizeOf(Self)) return .invalid_param;

    memcpyFromUnpriv(mem.asBytes(dest), mem.asBytes(src));
    return .success;
}

/// Copy out from a local variable to an unprivileged destination.
fn copyOut(
    src: *const Self,
    dest: *Self,
) SvcResult {
    const dest_param_len = v7m.ldrbt(&dest.len);
    if (dest_param_len != @sizeOf(Self)) return .invalid_param;

    memcpyToUnpriv(mem.asBytes(dest), mem.asBytes(src));
    return .success;
}
};

}

```

Unprivileged Data Access

The SVC instruction provides a means of transferring *control* from unprivileged execution to privileged execution. The SVC interface discussed previously allows for passing pointers to data structures which hold input, output, and error parameter data. The service realm handlers need access to that data and the last section showed the parameter processing functions. Passing parameters between background and foreground involves copying data across the privileged / unprivileged boundary.

A naive approach for transferring the data would directly use the pointer passed to a foreground proxy function. But even casual examination shows that directly using pointers in privileged execution which were passed from unprivileged code is a recipe for undoing the attempts to isolate the two privilege realms.

Consider, for example, returning output from a background request. Let's say that the request was to read data from a device and return that data to the background at an address specified as an input to the request. If the device data is written directly to an address passed in as an input parameter, then the memory access is done in privileged execution mode. Should the background processing make a mistake, either intentional or not, the device data could be written to memory used by privileged code. Critically, the naive approach allows the privileged code to become the unwitting patsy for unprivileged code to write into memory where it would otherwise not be allowed to write.

Fortunately, this issue is well understood. The essential rule is that any memory access by privileged code which transfers data to or from unprivileged code should be performed in an *unprivileged manner*. The ARMv7e-M architecture provides specific instructions to perform register loads and stores in an unprivileged manner. The LDR??T instructions allow reading byte, half-word, and word sized values from memory as either signed or unsigned integers and the access is performed in an *unprivileged manner*. Correspondingly, the STR??T instructions store byte, half-word, and word sized values into memory in an *unprivileged manner*. So the ability to perform unprivileged access from privileged execution is provided by the processor, but this means specific instructions must be used to do so. So, there is more systems programming and some assembly language to do.

Also note, that should an unprivileged function pass a pointer through the SVC interface to memory which requires privilege for access, using the unprivileged load and store instructions results in a Memory Management Fault or a Usage Fault.

Copying from Unprivileged Memory

To copy from unprivileged memory to privileged memory uses the ARM `ldrbt` instruction to force the memory load to be done in an unprivileged manner.

```
<<system_services: interface definitions>>=
/// The `memcpyFromUnpriv` function copies the bytes pointed to by the `unpriv_src`
/// slice to the memory pointed to by the `dest` slice. Data is read from memory
/// in an unprivileged manner. This function is used to move data from background
/// service requests into the foreground processing.
pub fn memcpyFromUnpriv(
    dest: []u8,
    unpriv_src: []const u8,
) void {
    for (dest[0..unpriv_src.len], unpriv_src) |*d, *s| {
        d.* = v7m.ldrbt(s);
    }
}
```

```
}
```

Copying to Unprivileged Memory

To copy to unprivileged memory from privileged memory uses the ARM `strbt` instruction to force the memory load to be done in an unprivileged manner.

```
<<system_services: interface definitions>>=
/// The `memcpyToUnpriv` function copies the bytes pointed to by the `src` slice
/// to the memory pointed to by the `unpriv_dest` slice. Data is written to memory
/// in an unprivileged manner. This function is used to move data from foreground
/// processing to the background.
pub fn memcpyToUnpriv(
    unpriv_dest: []u8,
    src: []const u8,
) void {
    for (unpriv_dest[0..src.len], src) |*d, s| {
        v7m.strbt(s, d);
    }
}
```

System Realm Request Interface

The pieces are now in place to show how a background request function for the system realm uses the SVC interface to transfer control to its foreground proxy function. For system realm requests, the `req` values are unsigned consecutive integers starting at zero. A data type is defined for the `req` argument.

```
<<system_services: interface definitions>>=
/// The `SysSvcRequest` enumeration gives a name to all system realm services.
pub const SysSvcRequest = enum(u8) {
    <<sys svc service requests>>
};
```

Note we use the literate program chunks to accumulate all the service requests that define the request enumeration. We use that technique for a number of enumeration definitions. Using literate program chunks is convenient for this book since it allows system requests to be accumulated into one place from across the entire book.

Making System Realm Requests

The purpose of the `sysRealmSvcCall` function is to coax the compiler into loading the operands into the proper registers and then to execute the SVC instruction with the appropriate immediate operand that indicates a system realm request. This results in the processor registers containing the correct values to match the expectations of the SVC handler.

```
<<system_services: interface definitions>>=
/// The `sysRealmSvcCall` function requests a foreground proxy to fulfill the
/// request given by the `req` argument. The `input`, `output`, and `err`
/// arguments are optional parameters as required by the specifics of each
```

```

/// request. Callers of `sysRealmSvcCall` must insure that the argument
/// expectations are met by the corresponding foreground proxy that executes
/// the request. Any errors detected in processing the request return
/// a `SvcError`. Otherwise, nothing is returned from the function.
pub fn sysRealmSvcCall(
    req: SysSvcRequest,
    input: ?*const anyopaque,
    output: ?*anyopaque,
    err: ?*anyopaque,
) SvcError!void {
    const result = asm volatile (
        "svc #0"
        : [ret] "{r0}" (-> SvcResult),
        : [req] "{r0}" (req),
        : [input] "{r1}" (input),
        : [output] "{r2}" (output),
        : [err] "{r3}" (err),
        : {.memory = true}
    );
    return result.mapToSvcError(); ⓘ
}

```

① Mapping numeric result codes to Zig errors happens in the background.

Handling System Realm Requests

As was shown in the description of the SVC handler, finding the appropriate system realm foreground proxy function first goes through a system realm handler. The system realm handler uses the `req` argument to locate the foreground proxy function. Note the interface to the system realm handler and to the request's proxy function is the same as that of the `sysRealmSvcCall` function. This is to be expected since the SVC handler itself simply passes through the value of the scratch registers R0 - R3 and recapitulates the role of the SVC exception handler as a transparent control transfer mechanism.

```

<<system_services: interface definitions>>=
/// The function signature of a foreground proxy for system realm requests.
/// Note the return value is an enumeration that is mapped to an
/// error if the request was not successful.
pub const SysSvcRequestProxy = *const fn (
    req: SysSvcRequest,
    input: ?*const anyopaque,
    output: ?*anyopaque,
    err: ?*anyopaque,
) SvcResult;

```

The implementation of the system realm handler uses a switch statement to generate the mapping of the request to the proxy function that handles the request.

```

<<svc handler>>=
export fn sysSvcHandler(
    req: SysSvcRequest,
    input: ?*const anyopaque,

```

```

    output: ?*anyopaque,
    err: ?*anyopaque,
) SvcResult {
    const proxy = switch (req) {
        <<sys svc service request prongs>>
    };
    return proxy(req, input, output, err);
}

```

Again, literate program chunks are used to accumulate the prongs of the switch statement. Those chunks take on the form:

```
<req name enum literal> => <proxy function name>
```

For example:

```
.foo => foo,
```

Termination Request

To demonstrate the details of the system realm interface, a simple background request for an abnormal termination is shown. Previously, the function, `systemAbEnd`, was defined. Because `systemAbEnd` accesses privileged registers, it may not be invoked from unprivileged code without causing a system fault. For background processing to terminate the system, the `terminate` function is defined and associated to a proxy in the foreground that executes `systemAbEnd`.

The following steps establish a pattern for creating system realm requests. We use literate program chunks to accumulate the services together.

For each service, a `struct` namespace is defined that contains the request function and any other supporting declarations.

```

<<system_services: system service definitions>>=
/// The terminate service is used to stop system execution and either
/// return to an attached debugger or reset the processor core.
pub const terminate = struct {
    <<terminate service>>
};

```

Each system SVC service must add a name to the request enumeration.

```
<<sys svc service requests>>=
terminate,
```

And each service requires a `switch` prong for the system service handler function.

```
<<sys svc service request prongs>>=
.terminate => &terminate.terminate,
```

A function for the background request must be defined. In this case, the `terminate` function is particularly simple since it takes no arguments and, ultimately, never returns.

```
<<terminate service>>=
/// The `terminate` function forces an abnormal termination of the system.
/// Ultimately, it executes the `systemAbEnd()` function. It is not expected to
/// return and by default forces a system reset.
pub fn terminate() SvcError!void {
    return sysRealmSvcCall(.terminate, null, null, null);
}
```

Termination Foreground Proxy

The other half of the system realm request is implemented by the foreground proxy function for `terminate`.

Request functions and proxy function are placed in separate files since they run in memory with different privilege. Again, we use literate program chunks to accumulate the proxy functions. Like requests, the proxy function associated with the request is placed in a `struct` namespace.

```
<<system proxies>>=
/// The terminate proxy implements the foreground portion of
/// the `terminate` service.
const terminate = struct {
    <<terminate proxy>>
};
```

```
<<terminate proxy>>=
/// The `terminateProxy` function implements the foreground processing
/// for the `terminate` system service.
fn terminate(
    : SysSvcRequest, ①
    : ?*const anyopaque,
    : ?*anyopaque,
    : ?*anyopaque,
) SvcResult {
    systemAbEnd();
}
```

① We don't really care about the `req` value. It was only used to vector to this function.

```
<<system proxies>>=
extern fn systemAbEnd() noreturn;
```

Device Realm Request Interface

The last section showed the interface to system realm requests using a `terminate` function as an example. In this section, background requests targeted at the device realm are described. The peripheral devices play a central role in the flow of data and control in the system and this section

lays out the pattern of device realm request processing.

In the system realm, the `req` argument was an integer encoded to identify a particular operation. There is a one-to-one mapping between the `req` value and a proxy function to fulfill the request. In the device realm, the `req` argument is interpreted differently. For devices, the mapping of the request to an operation function requires more information, namely:

- The *class* of the device. This is a numerical encoding that identifies the general category of the device.
- The *operation* on the device. Each device class provides its own set of operations which are specific to the device. There is no attempt to use any type of polymorphic interface for all devices. Some devices may follow more traditional “file” oriented behavior where `open`, `close`, `read`, and `write` might be appropriate. But for many microcontroller devices, considering them as producers or consumers of streams of bytes stretches the analogy so much that it no longer gives a convenient interface for device interaction. Many devices are simply control oriented and do not transfer data.
- The *instance* of the device. Many devices have multiple, independent instances of hardware which can be operated upon separately. All instances of a device support the same operations.

In this context, the device class may refer to a particular peripheral device or it may be considered a *logical* device which is some combination of physical peripherals. For example, accurately timed sampling of an *Analog to Digital Converter* (ADC), can involve a timer peripheral to trigger the signal conversion. This ensures that the conversion rate is stable and contains less jitter. It may also involve a *Direct Memory Access* (DMA) controller to move the data off the ADC and into memory. From a system context, it is preferable to treat such an arrangement of cooperating peripherals as a single logical device without having detailed information about the composition of the device from physical SOC peripheral components. Sometimes, a peripheral device may even be split into multiple logical devices. For example, I/O pins are used in many contexts both for direct interaction with the environment or as the electrical wiring of a peripheral device to the environment. The composition of peripheral device elements into a logical device is an *element of the design* for how device realm processing interacts with the SOC hardware.

The device realm must also account for there being multiple *instances* of a device. It is not uncommon to have multiple UART’s or SPI buses on an SOC. It is convenient to view that situation as multiple instances of a particular device class. This view implies that all operations supported by the device class may be applied to any instance.

The background requests into the device realm must supply the information discussed above. Since there is a single 32-bit integer value as the argument, the class, operation, and instance are encoded as bit fields within the `req` argument.

1. The class of the device is held in bits 0 - 7. This is a sequential integer encoding starting at zero.
2. The requested operation is held in bits 8 - 15. Again, a simple sequential integer encoding is used to identify an operation. The set of operations on a particular class of device is fixed at compile time.
3. The instance of the device to which the operation applies is held in bits 16 - 23. Yet again, sequential integers which start at zero are used to identify the instance.
4. Bits 24 - 31 are unused and reserved for those cases where 8 bits may not be sufficient for encoding one of the three fields which make up the device `req` argument.

The choice of the encoding for device realm entities is intended to make the identifiers useful as an array index and the encoded values are used directly in the implementation as indices.

To make the encoding of device class, operation, and instance more convenient in the code, a set of

types and functions are defined to encapsulate the specifics of the encoding rules.

```
<<system_services: interface definitions>>=
/// The `DevSvcClass` enumeration type gives the names of all device realm
/// services.
pub const DevSvcClass = enum(u8) { ❶
    <<svc device classes>>
};

pub const DevSvcOperation = u8;
pub const DevSvcInstance = u8;
/// Device service requests encode information about the device class,
/// the requested operation, and the device instance to which the
/// request is directed.
pub const DevSvcRequest = packed struct(mmio.WordRegister) {
    class: DevSvcClass,
    operation: DevSvcOperation,
    instance: DevSvcInstance,
    _reserved: u8 = ❷,
};
}
```

❶ Again, we are using literate program chunks to collect all the device classes.

```
<<system_services: interface definitions>>=
/// The `DefineDeviceOperations` type function returns a `struct` that
/// defines the encoding for the operations available for the given `class`.
/// The `ops` argument is a slice of constant strings giving the names
/// of the operations offered by the device.
fn DefineDeviceOperations(
    comptime class: DevSvcClass,
    comptime ops: []const [:0]const u8,
) type {
    const EnumField = std.builtin.Type.EnumField;
    var tag_fields: [ops.len]EnumField = undefined;
    inline for (ops, ❸..) |op_name, op_index| {
        tag_fields[op_index] = .{
            .name = op_name,
            .value = op_index,
        };
    }

    const OpEnum = @Type(.{
        .@("enum") = .{
            .tag_type = u8,
            .fields = &tag_fields,
            .decls = &.{},
            .is_exhaustive = true,
        },
    });
}

return struct {
    pub const Operation = OpEnum;
```

```

pub fn makeRequest(
    operation: Operation,
    instance: DevSvcInstance,
) DevSvcRequest {
    return .{
        .class = class,
        .operation = @intFromEnum(operation), ❶
        .instance = instance,
    };
}

```

- ❶ We transition from using enumerations to just plain numbers before entering the foreground side of the processing.

Making Device Realm Requests

Just as for the system realm, a function to execute the SVC instruction with the arguments in the appropriate registers and with the SVC immediate operand set to 1 is defined.

```

<<system_services: interface definitions>>=
/// The `devRealmSvcCall` function make a device service request via the SVC
/// mechanism.
pub fn devRealmSvcCall(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    output: ?*anyopaque,
    err: ?*anyopaque,
) SvcError!void {
    const result = asm volatile (
        "svc #1"
        : [ret] "{r0}" (-> SvcResult),
        : [req] "{r0}" (req),
        [input] "{r1}" (input),
        [output] "{r2}" (output),
        [err] "{r3}" (err),
        : {.memory = true}
    );
    return result.mapToSvcError();
}

```

Handling Device Realm Requests

The device realm, like the system realm, has a handler function which is selected and invoked directly by the SVC handler. The implementation follows the same pattern as `sysSvcHandler`. In this case, the first level of decomposition is based on the class of the device and the jump table selects the handler function based on device class.

```

<<system_services: interface definitions>>=
/// The type of a device service request foreground proxy function. This follows

```

```

/// a similar pattern as the system service request proxy functions.

pub const DevSvcRequestProxy = *const fn (
    req: DevSvcRequest,
    input: ?*const anyopaque,
    output: ?*anyopaque,
    err: ?*anyopaque,
) SvcResult;

```

Like the system SVC handler function, the literate program chunks collect the switch prongs needed to dispatch the device class handler function.

```

<<svc handler>>=
/// The `devSvcHandler` function receives control from the SVC exception
/// handler when a device request is made. Since devices have a two
/// level hierarchy before an operation is reached, this function serves
/// as a proxy dispatcher for the device classes.

export fn devSvcHandler(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    output: ?*anyopaque,
    err: ?*anyopaque,
) SvcResult {
    const proxy = switch (req.class) { ❶
        <<dev svc service request prongs>>
    };
    return proxy(req, input, output, err);
}

```

❶ It is the `class` field of the request that determines the first stage handler for devices.

Background Notification Interface

The last section presented the interface used by background processing to request device realm services. The third major component of the foreground/background interface is the background notification queue.

As shown in the [overview figure](#), a **Background Notification Queue** is the mechanism provided to IRQ handlers to make requests to a corresponding background notification proxy. The notification consist of:

- The device request value.
- A pointer to a background notification proxy function which acts upon the notification.
- A caller supplied *closure* value which is returned as part of the notification.
- Any other parameters which are considered important or are captured by the device interaction with the environment can be attached to the notification data structure.

When a device is initialized or opened and that device can send notifications, the caller must supply a notification proxy function whose job it is to act upon the notification. In code, the notification proxy is typed as:

```
<<system_services: interface definitions>>=
```

```
/// The data type for a function that receives notifications from a device.
```

```
pub const DevSvcNotifyProxy = *const fn (notification: *const anyopaque) void; ①
```

- ① The anyopaque type is used to avoid circular references. The true type of the notification is dependent upon the device notification and is recovered in the notification proxy function.

To facilitate transferring notification information between the foreground and background, device notification data is passed in a struct that contains both common notification data along with a tagged union that contains device specific notification data. This implies that we must gather together the union of the various notification types required by the various devices in the system. A type function does the heavy lifting.

```
<<system_services: interface definitions>>
/// The `DeclareDevNotifications` function is a type function that generates
/// a `struct` data type that is suitable for use as a device notification.
/// The generated struct contains a common portion and a tagged union that
/// contains additional notification parameters that are specific to each
/// device notification. The `param_specs` argument is taken as a list
/// of 2-tuples. Each 2-tuple describes a specific set of additional parameters
/// by giving a string name to serve as an enumeration tag and a corresponding
/// data type, e.g. `param_specs` might appear as:
///
/// .{
///     .{ "foodev", struct { a: u8, b: u32} },
///     .{ "bardev", u1 },
///     ...
/// }
///
/// The name portion of the 2-tuple is used as an enumeration tag and a tagged
/// union is created for parameters of all the devices given in the `param_specs`
/// argument. The return value is a `struct` type that contains a base portion of the
/// notification plus the tagged union of specific parameters for particular devices.
/// This type gives the structure for device notifications of all devices.
pub fn DeclareDevNotifications(
    comptime param_specs: anytype, // list of doubles, name / structure
) type {
    var param_type_fields: [param_specs.len]std.builtin.Type.EnumField = undefined;
    var param_union_fields: [param_specs.len]std.builtin.Type.UnionField = undefined;
    // Build up fields for the enumeration and union created below.
    inline for (param_specs, 0..) |param_spec, param_index| {
        if (param_spec.len != 2) @compileError("bad notification parameter specification");

        const param_name = param_spec.@"0";
        const param_type = param_spec.@"1";

        param_type_fields[param_index] = .{
            .name = param_name,
            .value = param_index,
        };
        param_union_fields[param_index] = .{
            .name = param_name,
            .type = param_type,
            .alignment = @alignOf(param_type),
        };
    };
}
```

```

    };
}

// The enumeration used to tag the device specific parameters.
const NotificationType = @Type(.{
    .@enum" = .{
        .tag_type = math.IntFittingRange(0, param_specs.len),
        .fields = &param_type_fields,
        .decls = &.{},
        .is_exhaustive = true,
    },
});

// The tagged union of all the device specific parameters.
const Parameters = @Type(.{
    .@union" = .{
        .layout = .auto,
        .tag_type = NotificationType,
        .fields = &param_union_fields,
        .decls = &.{},
    },
});

return struct {
    notify_size: u32,
    class: DevSvcClass,
    instance: DevSvcInstance,
    proxy: DevSvcNotifyProxy,
    closure: usize,
    params: Parameters,
    const Self = @This();

    pub fn init(
        class: DevSvcClass,
        instance: DevSvcInstance,
        proxy: DevSvcNotifyProxy,
        closure: usize,
        params: Parameters,
    ) Self {
        return .{
            .notify_size = @sizeOf(Self),
            .class = class,
            .instance = instance,
            .proxy = proxy,
            .closure = closure,
            .params = params,
        };
    }
};
}

```

Using literate program chunks, we can collect the notification parameter specifications for all the devices in the system.

```

<<system_services: interface definitions>>=
/// The type of a device notification is constructed for all devices
/// which send background notifications.
pub const DevNotification = DeclareDevNotifications(.{
    <<dev notification specs>>
});

```

The device notification structure includes a closure value that is large enough to hold a pointer (i.e. `usize`). Note that the lifetime of the memory associated with any pointer must be managed by the background. The notification mechanism simply passes along the value which it was originally given. Background processing which sets up the binding to the notification proxy must make sure any pointer values are still valid by the time the notification is issued. That's a lot of words to say, "don't use pointers to automatic variables as notification closures."

By analogy to background requests, a peripheral device that places a notification into the queue has one or more (usually one) proxy functions that are run in the background to handle the request. The background notification proxy can be considered as a continuation of the IRQ handler which computes the system response to the environmental condition detected by the IRQ. The peripheral device code and proxy function must agree to the structure of the notification since typing is lost during the transit through the queue and must recovered by the background proxy function. It is an *element of the design* of the device control code as to which notifications it offers. Device control code must clearly document the intent and meaning of any notifications it generates. Any additional data beyond that provided by the base notification structure must also be carefully specified.

The remaining consideration of the notification design is to specify the time when the notification proxy function is bound to a notification offered by a device. For background *requests*, the binding between the background function which issues requests and the foreground proxy function which services them is made at compile time. This is the simplest approach and, given the characteristics of foreground requests, additional flexibility is not required. For background *notifications*, the proxy function is bound at run time. This implies that at least part of the configuration of a peripheral device is to specify the notification proxy functions to be executed in the background when the device detects an actionable condition. Devices may offer to issue multiple types of notifications, depending upon the specific semantic actions associated with the device.

The advantages of using run time binding of the background callback are:

- Although the system is expected to run a single program at a time, during development many test programs and integration programs are run. If the notification proxy functions were bound at compile time, all application would be required to supply an implementation of the functions which complicates the build/test process.
- In some cases, the condition detected by the device control code is not interesting to the application and it would be more convenient *not* to require specifying a notification proxy function for an unused condition, i.e. to specify a *optional* function pointer. This knowledge may allow the peripheral device code to mask certain interrupt sources or take other actions to reduce its interaction with the environment.

Run time binding does imply that the peripheral device code must store the binding somewhere. This is usually of little consequence since device code must also store many other parameters used for controlling the peripheral itself.

It may seem dangerous to pass around "raw" function pointers between foreground and background. Certainly passing an incorrect function pointer can result in uncontrolled execution. It is not strictly necessary to use a function pointer. The background notifications could be encoded as integers (as they are for background requests). This requires keeping a mapping from the integer

encoding to the corresponding proxy function which is used during dispatching the background notification. Such an encoding would also have to be constructed at run time and supplying an incorrect encoded integer when building the mapping would result in the same uncontrolled execution. In practice, the use of function pointers is simpler and does not cause significant problems.

Using function pointers does not jeopardize the separation between privileged and unprivileged execution. The notification proxy function is *not* executed by privileged code. Invocation of notification proxy functions is performed by background processing in unprivileged mode. Both the notification proxy function pointer and the notification closure data value traverse through the device control code to the background notification queue modified and uninterpreted.

Background Notification Queue Operations

The background notification queue works as a temporary, ordered buffer allowing multiple IRQ handlers to post notifications while background processing is executing. Recall that interrupts are enabled during background execution to enable the foreground to cope with the concurrency of the system environment. The notification queue acts as an “accordion”, expanding when IRQ handlers run during background execution and contracting when the foreground has no addition notifications. We expect the queue to be empty most of the time. If the rate of production of notifications exceeds the rate of handling them in the background, eventually no amount of buffering will prevent overflow.

These characteristics suggest a simple design of the notification queue that can be tailored to the specific needs of device notification. Since execution flow through the notification is an important area, we are willing discard any notions of generality and implement a simple array with a read index and a write index. The array is *not* treated in a circular manner, i.e. there is no index wrap around. Rather, both indices are reset to the beginning of the array when it becomes empty. Since there is only a single queue for the entire system, it is implemented as a namespace **struct**.

The size of the array used for the notification queue is fixed at compile time. Sizing the background notification queue involves estimating the maximum number of notifications (and therefore the maximum number of notifying IRQ handlers) that will be sent during the execution of any background notification proxy. It is difficult to estimate the appropriate number of queue elements with no prior knowledge of the application characteristics.

Experience shows most background notifications are small in size, typically the default notification size with no device specific parameters. Passing large amounts of data through the queue is not advisable, especially since there are other techniques for passing data and any data passed through the queue must be copied into the background. The worst case scenario for interrupts is they all go off at the same time (or nearly so; the processor core ensures that IRQ's are activated in a deterministic order). The buffer needs to cover burst or peek behavior and should *never* overflow. However, overflow can be seen during hardware failures which produce a rapid burst of interrupts into the system, unexpectedly high traffic on a particular device, or background processing that might be hung in an infinite loop. How to handle overflow is dependent upon the semantics of the operation. Sometimes, e.g. when handling communications packets, data can be dropped and the communications protocol handles the necessary retries. Other system semantics are such that the inability to make background notifications for a critical system function implies a *panic* condition, from which there may be little recourse for recovery. It is necessary to characterize the dynamics of the system when it is functionally complete to tune the buffer memory allocation. This can be said about many aspects of any microcontroller system. To ease this burden, there is a system configuration constant, `notify_queue_elements`, to allow changing the number of element given to the notification queue.

```
<<system_config: constants>>=
```

```
// The number of elements in the device notification queue.  
pub const notify_queue_elements: usize = 10;
```

```
<<notify queue>>=  
/// The `dev_notify_queue` data structure holds the device notifications from  
/// IRQ handlers until they are retrieved by background processing.  
/// Retrieval is in FIFO order. The data structure is a simple queue  
/// implemented in an array and specialized to the requirements of IRQ handlers and  
/// the system service which fetches the notifications to the background.  
/// The code provides two techniques for sending notifications. The `sendNotification`  
/// function copies pre-constructed `DevNotification` data into the queue.  
/// Alternatively, the `reserveNotification` and `commitNotification` can be  
/// used to acquire space for the notification data which can be used directly to  
/// place the notification data into queue storage.  
const dev_notify_queue = struct {  
    /// The actual size of the queue is the next highest power of two to allow  
    /// us to simpler modulus arithmetic.  
    /// A system configuration option determines the base count.  
    const queue_size_bits = std.math.log2_int_ceil(usize, system_config  
.notify_queue_elements);  
    const queue_size = 1 << queue_size_bits;  
    const Index = std.meta.Int(.unsigned, queue_size_bits);  
  
    var storage: [queue_size]DevNotification = undefined;  
    var send_index: Index = 0;  
    var recv_index: Index = 0;  
  
    /// The `sendNotification` function places the device notification pointed to  
    /// by the `notification` argument into the notification queue.  
    ///  
    /// NOTE: This function must be invoked at IRQ priority level. This means that  
    /// the function is called from an IRQ handler or the current base priority has  
    /// been set to the IRQ priority level. This function assumes that it will _not_ be  
    preempted by  
    /// other IRQ handlers. This is true for the flat IRQ priority scheme implemented  
    /// in this design.  
    fn sendNotification(  
        notification: *const DevNotification,  
    ) SvcResult {  
        const next_index = nextSendIndex();  
        if (next_index == recv_index) return .operation_failed;  
        storage[send_index] = notification.*;  
        send_index = next_index;  
        return .success;  
    }  
  
    /// The `reserveNotification` function returns the next available queue  
    /// location. The result contains an error if the queue overflows.  
    /// The notification data can be written via the returned pointer.  
    /// After the notification data is completely transferred,  
    /// the `commitNotification` function must be invoked.
```

```

fn reserveNotification() struct {SvcResult, *DevNotification} {
    // Note that if the reservation fails, the address of the last
    // queue storage slot is returned. In extreme cases, where it
    // is necessary to push through a notification, the last place
    // in the queue could be overwritten. A rare circumstance!
    const next_index = nextSendIndex();
    return if (next_index == recv_index)
        {.operation_failed, &storage[send_index]}
    else
        {.success, &storage[send_index]};
}

/// The `commitNotification` notification function is called after
/// a notification is written to the queue. It is used in conjunction
/// with the `reserveNotification` function to allow the caller to
/// build the notification information directly into the queue and
/// avoid a possible copy.
fn commitNotification() void {
    const next_index = nextSendIndex();
    if (next_index != recv_index) send_index = next_index;
}

/// The `recvNotification` function retrieves a device notification from the
notification
/// queue and places it in the memory pointed to by the `notification` argument.
/// Transfer of the data from the queue to its destination is done in an unprivileged
/// manner so that the data may be written directly from the foreground to the
background.
///
/// NOTE: This function is provided for use by the system foreground proxy function
/// that retrieves device notification functions. That function runs at SVC priority
/// which is lower than IRQ priority. Therefore, this function implements a priority
/// section to ensure that no IRQ handlers run while the queue data transfer
/// is taking place.
fn recvNotification(
    notification: *DevNotification,
) SvcResult {
// BEGIN PRIORITY SECTION
{
    const section =
        v7m.basepri.PrioritySection.enter(system_config.irq_encoded_priority);
    defer section.leave();

    if (empty()) return .retry_operation;

    memcpyToUnpriv(
        mem.asBytes(notification), // destination
        mem.asBytes(&storage[recv_index]), // source
    );
    recv_index += 1;
}
// END PRIORITY SECTION

```

```

    return .success;
}

/// The `empty` function tests if the notification queue given by the
/// `queue` argument is empty. It returns `true` if `queue` contains no
/// elements and `false` otherwise.
///
/// NOTE: This function is provided for the use in system foreground
/// processing. When the function is invoked it assumes the execution priority
/// is at IRQ priority or greater.
fn empty() bool {
    return recv_index == send_index;
}

/// The `nextSendIndex` function computes the index where the next notification
/// will be placed in the queue. Note this algorithm is simple. It is based on
/// power of 2 modulus arithmetic and and consumes a slot in the queue to detect
overflow.
fn nextSendIndex() Index {
    return send_index +% 1;
}
};

```

Watchdog Timer

As a first demonstration of device services, a simple peripheral is illustrative. A watchdog timer is nonetheless an important peripheral that is needed as part of the basic protections for any system. In later chapters, many other examples of peripherals which generate interrupts and issue background notifications are shown.

A watchdog timer is a device which helps to ensure that some fundamental function of a system is operational by repeatedly requiring the system to *service* (or *acknowledge* or *feed* or *pet*) the watchdog. Typically, a watchdog timer is used to ensure that overall execution flow in the system is functioning. Should some code sequence result in a long compute bound, possibly infinite loop, then other operations in the system may be starved for access to the CPU and essential functions of the system might not execute. Sometimes a watchdog is used to ensure that a particular system function, in contrast to the system as a whole, works correctly. Such system functions are usually periodic and a watchdog may be used to ensure that the function happens at the correct period. For example, cardiac pacemakers have a watchdog timer that specifically monitors the issuing of pace pulses to the heart. Failure to issue a pace pulse regularly and within a certain time frame would be a fundamental system failure and be hazardous to the patient. Potentially, the pacemaker could fail to issue pace pulses yet its overall control flow might still be functional. Some SOC's supply multiple watchdog timers to allow monitoring overall system execution as well as specialized system operations.

Watchdog timers peripherals work by counting the cycles of a clock. If the timer count ever reaches 0 (if counting down) or a specified value (if counting up), then the watchdog issues a system reset signal. Software is expected to service the watchdog to restart its counting. The specifics of watchdog capabilities varies between SOC's, but most watchdog timer peripherals usually include a means for the watchdog to notify software that the reset period expires soon. This takes the form of an interrupt to prompt the system software to service the watchdog. This gives software a maximum deadline in which it must accomplish the watchdog service. Some watchdog timers also have the facility to cause a reset if the timer is serviced too soon, i.e. it demands servicing happen

in a time window before the reset time. There are still other variations on the theme.

It is particularly important that code which services the watchdog exercise the critically important paths that are being protected. This implies that you *must not* restart the watchdog in the IRQ handler associated with a watchdog notification interrupt. This would negate any protection offered by a watchdog timer. It is easy to conceive of an infinite loop running somewhere that is preempted by the watchdog interrupt but is otherwise preventing essential system operations.

Watchdog Timer Peripheral

The figure below, taken from the Apollo 3 datasheet, shows the design for the watchdog timer peripheral.

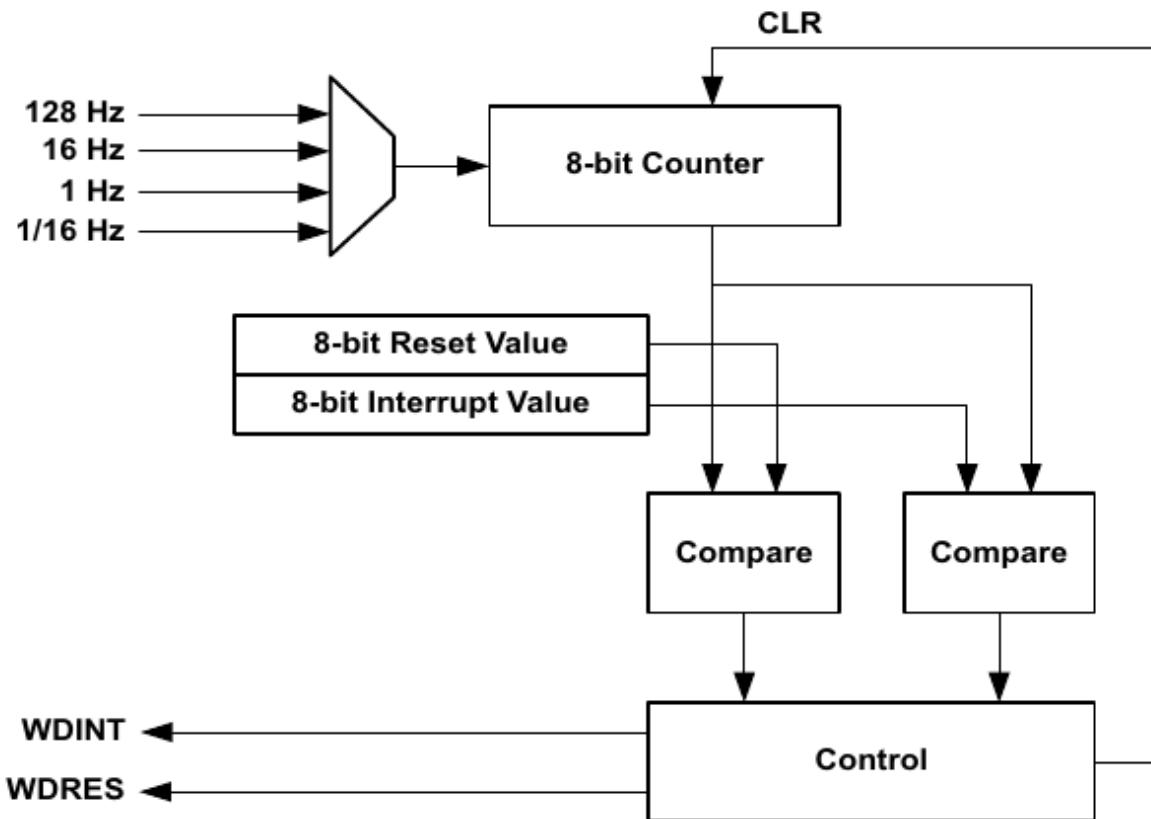


Figure 7. Schematic of Apollo 3 Watchdog Timer Peripheral

This peripheral operates by using a selectable clock to increment a counter. The incrementing counter can trigger a comparison match against the values of two compare registers. A match between the values in the Counter and Interrupt Value registers causes an interrupt. A match between the Counter and Reset Value registers causes a system reset.

SOC designers parameterize the operations of peripheral devices to make them adaptable to a wider range of applications. The goal here is to tailor the usage of the devices to the particular requirements of the system being built. There is no intent to support every possible configuration. For example, the Apollo 3 watchdog timer supports *locking* the configuration registers so that once set, they may not be changed until the system is reset. The paranoia level does not extend to the need to prevent highly improbable register writes, especially as that possibility has been excluded from the background processing as a result of separating privileged and unprivileged execution.

One quirk of the Apollo 3 watchdog does require some special arrangements and accommodation. Many SOC watchdog timers have a setting which disables the watchdog timer clock when a debugger is active. This makes breakpoint style debugging much easier. The Apollo 3 watchdog does *not* have this ability. It is a pain point that requires a work around.

Watchdog Timer Logical Device

When designing the interface to a peripheral device, a drawing is beneficial, even for a device as simple as a watchdog timer. The following figure shows the components of the watchdog timer interface.

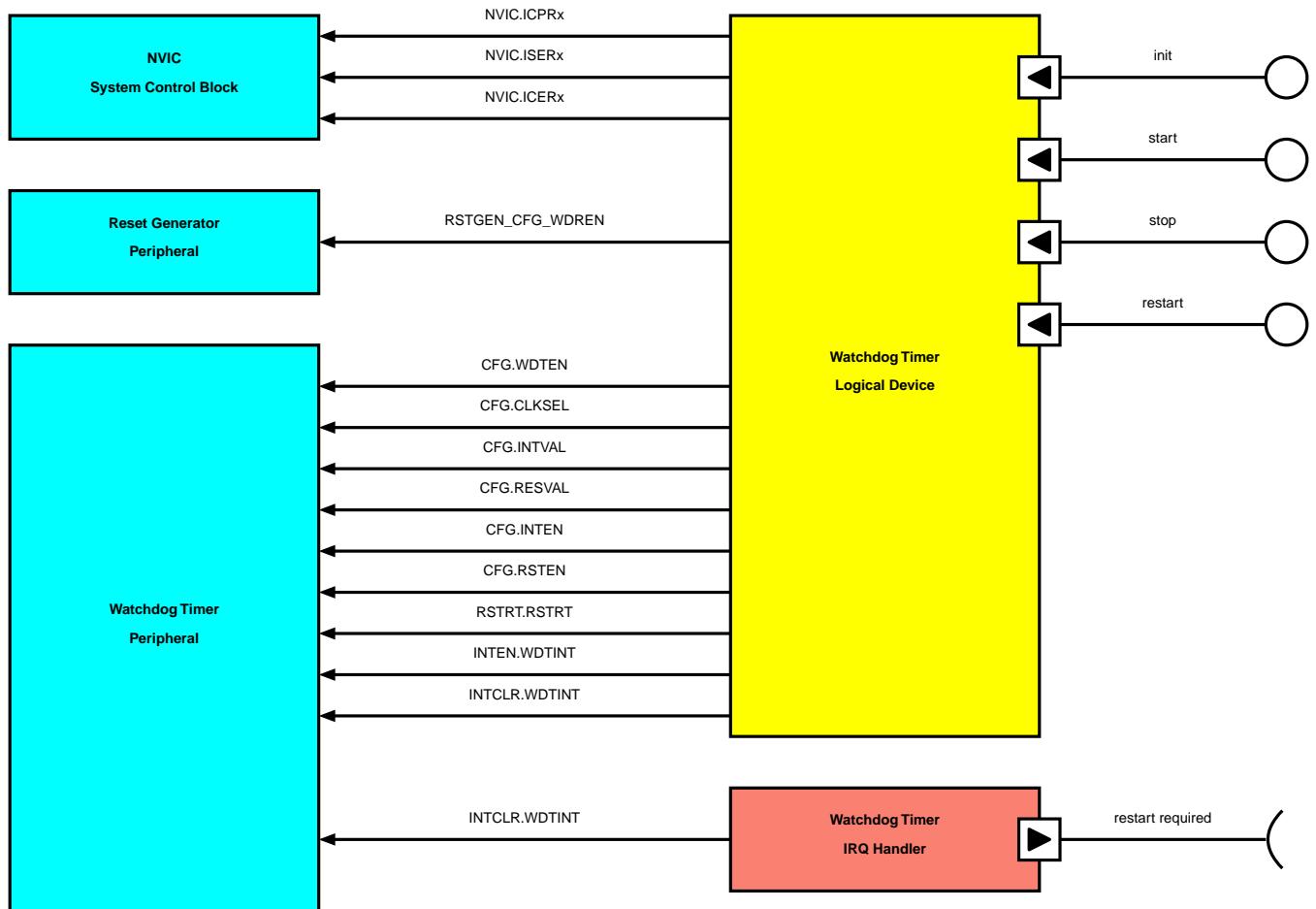


Figure 8. Watchdog Timer Logical Device Components

In the previous figure, the yellow box represents the watchdog timer from the point of view of the background services. The available functions are:

init

Configure the device and make it ready to run.

start

Start the watchdog timer running.

stop

Stop the watchdog timer.

restart

Reset the watchdog timer to zero. This operation acknowledges the watchdog preventing it from causing a system reset.

The red box represents the watchdog timer IRQ handler. It issues a single background notification indicating that the watchdog expires soon and needs to be restarted.

The cyan boxes represent Apollo 3 peripheral devices. The arrows represent access (read or write) to the peripheral registers or register fields as given by the label.

NVIC

The NVIC is used clear and enable the interrupt globally in the processor core.

Reset Generator

For the watchdog timer to cause a system reset, it must be enabled to do so at the reset generator peripheral.

Watchdog Timer

The watchdog timer peripheral has a set of registers to control its functions.

Device Control

Since this is the first situation where we have need to control interrupt requests, we factor out some common code used to handle interrupts at the NVIC. All foreground proxies that handle peripherals that can pend interrupts have need of functions to handle the interrupt at the NVIC. Note we must still deal with interrupts control registers in the peripheral device itself.

```
<<device proxies>>=
/// The `IrqControl` type holds that information for handling interactions with the
/// NVIC that are specific to a particular interrupt request.
const IrqControl = struct {
    irq_number: v7m.IrqNumber,

    fn enable(
        self: IrqControl,
    ) void {
        v7m.nvic.enableIrq(self.irq_number);
    }

    fn disable(
        self: IrqControl,
    ) void {
        v7m.nvic.disableIrq(self.irq_number);
    }

    fn clearPending(
        self: IrqControl,
    ) void {
        v7m.nvic.clearPendingIrq(self.irq_number);
    }
};
```

Following the diagram in the previous figure, we create data structures to manage the interactions between the watchdog timer logical device and its peripheral components.

```
<<watchdog proxies>>
const irq_control: IrqControl = .{.irq_number = apollo3.WatchdogTimer.irq_number};
```

Note that there is only one Reset Generator in the Apollo 3. Here we simply create a namespace struct to present declarative function whose names showing the intent of the operation.

```
<<watchdog proxies>>
const reset_control = struct {
    fn enable() void {
        apollo3.rstgen.cfg.setBitField(.wdr_en);
    }

    fn disable() void {
        apollo3.rstgen.cfg.clearBitField(.wdr_en);
    }
};
```

Again, there is only one Watchdog Timer in the Apollo 3 and we follow the same pattern as for the Reset Generator.

```
<<watchdog proxies>>
const wdog_control = struct {
    fn enable() void {
        apollo3.wdt.cfg.setBitField(.wdt_en);
    }

    fn disable() void {
        apollo3.wdt.cfg.clearBitField(.wdt_en);
    }

    fn restart() void {
        apollo3.wdt.rstrt.write(.restart_key); ①
    }

    fn clearInterrupt() void {
        apollo3.wdt.int_clr.writeBitField(.wdt_int);
    }

    fn enableInterrupt() void {
        apollo3.wdt.int_en.writeBitField(.wdt_int);
    }

    fn disableInterrupt() void {
        apollo3.wdt.int_en.writeDirect(0);
    }

    fn enableReset() void {
        apollo3.wdt.cfg.setBitField(.res_en);
    }
};
```

```

    }

    fn disableReset() void {
        apollo3.wdt.cfg.clearBitField(.res_en);
    }

    fn configure(
        notify_time: u8,
        reset_time: u8,
    ) void {
        apollo3.wdt.cfg.writeFields(.{
            .clk_sel = .hz_16,
            .int_val = notify_time,
            .res_val = reset_time,
            .int_en = 1,
        });
    }
}

```

- ① A magic “key” value is required to restart the watchdog.

Watchdog Timer Requests

There are four functions provided to background processing to control the watchdog timer. This implies that eight functions must be provided, four used by background processing to make watchdog requests and four that serve as proxies to the background requests. The sections below cover each watchdog timer operation.

Here, as with system requests, we establish a pattern for how the background / foreground interactions are presented in code.

First, each device service places background requests in a `struct` container.

```

<<system_services: device service definitions>>=

/// Watchdog services
pub const wdog = struct {
    <<watchdog services>>
};

```

As mentioned above, there are four watchdog requests available to the background.

```

<<watchdog services>>=
pub const WdogOperations = DefineDeviceOperations(
    .watchdog,
    &{ "init", "start", "stop", "restart" },
);

```

The Apollo 3 has only a single watchdog timer peripheral.

```
<<watchdog services>>=
```

```
pub const instances: DevSvcInstance = 1;
```

Init

It is necessary to initialize the watchdog timer before use.

```
<<watchdog services>>=
/// The `init` function initializes a watchdog timer instance.
/// This function must be called before any other watchdog timer function.
/// This function initialized the given watchdog timer so that
/// the timer will expire and potentially reset the system after `reset_time`
/// milliseconds have elapsed since the watchdog was started or restarted.
/// The `reset_time` must be between 63 ms and 15900 ms and values exceeding that
/// range are silently saturated to the boundaries.
/// When `notify_time` milliseconds have elapsed, the watchdog timer issues
/// a notification to the background that the watchdog needs to be restarted
/// to prevent a system reset.
/// The `notify_time` has the same range boundaries as the `reset_time` but
/// must be less than the `reset_time`
pub fn init(
    /// Note that the function provides an instance argument despite there
    /// being only a single instance of watchdog timer in the Apollo 3. It is included
    /// in the interface for consistency and future compatibility.
    _: DevSvcInstance,
    reset_time: u16,
    notify_time: u16,
    /// A pointer to a function that is to be invoked during background processing when
    /// the watchdog timer notification is dispatched.
    proxy: DevSvcNotifyProxy,
    /// A value passed to `notify_proxy` when it is dispatched. The value is large
    /// enough to hold a pointer. If a pointer value is used, its type must be
    /// recovered in the proxy function and the lifetime of the memory pointed to is the
    /// responsibility of the caller (_i.e._ if `notify_closure` is a pointer,
    /// background processing must ensure the memory is still valid by the time the
    /// notification is dispatched, _e.g._ using a pointer to an automatic variable is
    /// undefined). If no additional closure information is needed by `notify_proxy`,
    /// it is recommended that `notify_closure` have a value of zero.
    closure: usize,
) SvcError!void {
    const input_params = InitInputParam.marshal(.{
        .reset_time = reset_time,
        .notify_time = notify_time,
        .notify_proxy = proxy,
        .notify_closure = closure,
    });
    const request = WdogOperations.makeRequest(.init, 0); // There is only 1 instance.
    return devRealmSvcCall(request, @ptrCast(&input_params), null, null);
}
```

The input parameter for the `init` function carries the arguments from the background request.

```
<<watchdog services>>=
pub const InitInputParam = DefineSvcRequestParam(struct {
    reset_time: u16,
    notify_time: u16,
    notify_proxy: DevSvcNotifyProxy,
    notify_closure: usize,
});
```

Similar to the way background services organized the corresponding foreground proxies, each set of device foreground proxies is placed in a `struct` namespace.

```
<<device proxies>>=

/// Watchdog service proxies
const wdog = struct {
    <<watchdog proxies>>
};
```

The watchdog device issues a device notification when the notification time has expired. There are no additional parameters for this notification.

```
<<dev notification specs>>=
.{ "watchdog", void },
```

During initialization, the information used to issue the notification to the background when the watchdog timer `notify_time` interrupt occurs must be saved. Since there is only one instance of the watchdog timer, the information can be saved in a single variable.

```
<<watchdog proxies>>=
var notification: DevNotification = undefined;
```

The following function is the foreground proxy function for `init`. This function receives control from the SVC exception handler indirectly through the [watchdog timer operation handler](#).

```
<<watchdog proxies>>=
/// The `init` function is the foreground proxy for the `init` background
/// request of the watchdog device.
fn init(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const InitInputParam = svc_services.wdog.InitInputParam;

    const init_input, const result = InitInputParam.receive(input.?);
    if (result != .success) {
        return result;
    }
```

```

if (init_input.reset_time == 0 or
    init_input.notify_time == 0 or
    init_input.reset_time <= init_input.notify_time)
{
    return .invalid_param;
}

irq_control.disable(); ①
wdog_control.disable();

notification = DevNotification.init(
    req.class,
    req.instance,
    init_input.notify_proxy,
    init_input.notify_closure,
    {.watchdog = {} },
);

wdog_control.configure(
    msToTicks(init_input.notify_time),
    msToTicks(init_input.reset_time),
);
wdog_control.clearInterrupt();
wdog_control.enableInterrupt();
irq_control.clearPending();
irq_control.enable();

return .success;
}

```

① In case the watchdog is re-initialized, prevent any interrupt and disable the timer.

The time arguments for the watchdog have a range from approx 0.0625 to 15.9 seconds. After starting the watchdog timer using `start`, the difference between the `reset_time` and `notify_time` is the amount of time software has to invoke the `restart` function before the watchdog timer causes a system reset.

The units for the notify and reset times are raw device units of clock ticks. The conversion from milliseconds to watchdog time units is accomplished by the following function. Note that this conversion depends upon the setting of the clock driving the watchdog timer. That clock value is pre-selected and is not changeable. It value was selected to provide a convenient range of millisecond time ranges for our intended applications.

```

<<watchdog proxies>>
/// The `msToTicks` function converts the `ms` value into watchdog timer ticks
/// in the form of an unsigned UQ4.4 fixed binary point number. The maximum
/// amount of time which can be specified is 15.9375 seconds. The minimum time
/// amount is 0.0625 seconds. Any value of `ms` which causes an overflow of the
/// allowed range is silently truncated to the maximum time value. Any value
/// of `ms` which converts to zero ticks returns 1.
fn msToTicks(
    /// A number of milliseconds to convert into watchdog timer ticks.
    ms: u32,

```

```

) u8 {
    var ticks = math.divCeil(u32, ms << 4, std.time.ms_per_s) catch unreachable;
    ticks = @min(ticks, 0xff);
    ticks = @max(ticks, 1);
    return @truncate(ticks);
}

```

Start

After the watchdog is initialized it must be started.

```

<<watchdog services>>=
/// The `start` function starts the watchdog timer running.
/// This function must be called only after invoking `init`.
/// After invoking `start`, background notifications are issued
/// periodically according to the value of `notify_time` value given
/// to the `init` function.
/// The `reset_enable` argument is a boolean value which determines whether
/// system is reset when the watchdog timer expires.
/// A value of `true` implies that an unacknowledged watchdog
/// causes a system reset. A value of `false` implies that the watchdog timer
/// never causes a system reset. The use case for this parameter is to allow
/// breakpoint style debugging since the Apollo 3 SOC does not halt the watchdog
/// timer clock when stopped for debugging purposes.
pub fn start(
    _: DevSvcInstance,
    enable_reset: bool,
) SvcError!void {
    const input_params = StartInputParam.marshal(.{
        .enable_reset = enable_reset,
    });
    const request = WdogOperations.makeRequest(.start, 0);
    return devRealmSvcCall(request, @ptrCast(&input_params), null, null);
}

```

The single argument, `enable_reset` argument is given as an input to the proxy function.

```

<<watchdog services>>=
pub const StartInputParam = DefineSvcRequestParam(struct {
    enable_reset: bool,
});

```

The proxy function implementation follows the pattern established by the `init` function.

```

<<watchdog proxies>>=
/// The `start` function is the foreground proxy for the `start` background
/// request of the watchdog device.
fn start(
    _: DevSvcRequest,
    input: ?*const anyopaque,
)

```

```

    _ : ?*anyopaque,
    _ : ?*anyopaque,
) SvcResult {
    const StartInputParam = svc_services.wdog.StartInputParam;
    const start_input, const result = StartInputParam.receive(input.?);
    if (result != .success) {
        return result;
    }

// INTERRUPT SECTION BEGIN ①
{
    irq_control.disable();
    defer irq_control.enable(); ②

    wdog_control.disable();
    if (start_input.enable_reset) {
        wdog_control.enableReset();
        reset_control.enable();
    } else {
        wdog_control.disableReset();
        reset_control.disable();
    }
    wdog_control.restart(); ③
    wdog_control.clearInterrupt();
    wdog_control.enableInterrupt();
    wdog_control.enable();

}
// INTERRUPT SECTION END

return .success;
}

```

- ① The foreground proxy functions run at SVC exception priority. To allow starting an already running watchdog, it is safer to prevent the watchdog timer interrupt from happening since it would preempt the proxy.
- ② Make sure the IRQ is enabled after start processing.
- ③ Issue a restart to the watchdog in case there is any timing boundary conditions on start up.

Stop

The watchdog Stop request ceases watchdog operations. Generally, this is never called. It is provided for test uses.

```

<<watchdog services>>=
/// The `stop` function stops the given watchdog timer.
/// After invoking `stop` no further notifications from the device are
/// issued until it is started again.
pub fn stop(
    _: DevSvcInstance,
) SvcError!void {
    const request = WdogOperations.makeRequest(.stop, 0);

```

```

    return devRealmSvcCall(request, null, null, null); ①
}

```

① There are no parameters for the stop proxy function.

The proxy function for `stop` is also simpler because it lacks any input or output parameters.

```

<<watchdog proxies>>=
/// The `stopProxy` function is the foreground proxy for the `stop` background
/// request of the watchdog device.
fn stop(
    _: DevSvcRequest,
    _: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    irq_control.disable();

    wdog_control.disable();
    reset_control.disable();
    wdog_control.disableInterrupt();
    irq_control.clearPending();

    return .success;
}

```

[Restart](#)

```

<<watchdog services>>=
/// The `restart` function restarts the given watchdog timer
/// ensuring that the watchdog does not cause a system reset.
pub fn restart(
    _: DevSvcInstance,
) SvcError!void {
    const request = WdogOperations.makeRequest(.restart, 0);
    return devRealmSvcCall(request, null, null, null);
}

```

The implementation of the proxy function is just as simple. A single register write is sufficient.

```

<<watchdog proxies>>=
/// The `restartProxy` function is the foreground proxy for the `restart` background
/// request of the watchdog device.
fn restart(
    _: DevSvcRequest,
    _: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    wdog_control.restart();
    return .success;
}

```

```
}
```

Dispatching Watchdog Requests

The device realm has an additional [level of dispatch](#) based on the class of the device. Another function is required to find the proxy function associated with the device operation.

Like the service realm dispatch function, operation dispatching for a device is also based on a jump table.

We define a name for the device in the enumeration for all the device classes.

The watchdog timer device must be assigned a class identifier and a pointer to the operation dispatch function must be placed in the jump table for device realm dispatch.

```
<<svc device classes>>=
watchdog,
```

```
<<dev svc service request prongs>>=
.watchdog => &wdog.devClassHandler,
```

Finally, the watchdog class handler performs the dispatch to the proxy functions corresponding to the requests.

```
<<watchdog proxies>>=
/// The `devClassHandler` function dispatches watchdog operations based
/// on the value of the `req` argument. This function is invoked as a
/// class level dispatcher for the watchdog device.
fn devClassHandler(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    output: ?*anyopaque,
    err: ?*anyopaque,
) SvcResult {
    const Operation = svc_services.wdog.WdogOperations.Operation;
    const wdog_operation: Operation = @enumFromInt(req.operation);
    const proxy = switch (wdog_operation) {
        .init => &init,
        .start => &start,
        .stop => &stop,
        .restart => &restart,
    };
    return proxy(req, input, output, err);
}
```

Watchdog Timer IRQ Handler

The last part of the watchdog timer device code is the IRQ handler. This is the first example that shows how an IRQ handler posts background notifications. This notification informs background processing that it should service the watchdog device soon or else the system may reset

(depending upon how the watchdog was started). This notification is intended to exercise critical parts of the system control flow which is discussed in the [next section](#).



This is only part of the overall execution path that needs to be exercised by the watchdog notification. How the notification is handled by background processing is essential to closing the monitoring loop that ensures the system behaves within expected bounds.

During initialization (as shown previously in `init`), necessary information was stored away for later use in formulating the notification from the device. The implementation of the IRQ Handler uses the previously saved notification information and inserts it into the background notification queue. Note it is not necessary to protect access to the background request queue since an IRQ handler is running and all IRQ's have the same priority. No other IRQ can preempt the execution and no higher priority exceptions access the background notification queue.

```
<<watchdog proxies>>
/// The `wdtIrqHandler` function receives control when the watchdog notification
/// timer has expired.
export fn wdtIrqHandler() void {
    wdog_control.clearInterrupt();

    if (dev_notify_queue.sendNotification(&notification) != .success)
        panic.privPanic(null, "Watchdog notification failed");
}
```

As this is the first IRQ Handler shown, note that there is no additional processing required upon entry into or exit from the IRQ Handler. The vector table contains a direct pointer to the function just shown. There is no need to inform the execution architecture that an IRQ has occurred. The only operation an IRQ Handler can perform which has an effect on the system is to post a background notification. An IRQ Handler may *not* need to post a notification or may do so conditionally.

Foreground to Background Control Flow

Thus far, this chapter has described techniques to build a system control mechanism to mediate between foreground processing and background processing. To recap, execution is initiated by interrupt requests which come from conditions detected in the environment as sensed by peripheral devices. The background processing can control the behavior of peripheral devices by making requests to the foreground proxies of a device. In the example just shown, the watchdog timer is controlled by specific services requests from background code.

When the peripheral device control code detects changes in the environment, it generates notifications to background processing indicating the changes. These notifications are placed in the **Background Notification Queue**. The sole control transfer mechanism between foreground and background processing is for the background run time to repeatedly request the next notification and then dispatch it. When the notification queue is empty and when background processing has determined there is no additional work to perform as a result of having dispatched all the notifications, it issues a request to the foreground to *wait* until an IRQ handler posts a new notification indicating actionable conditions that have arisen in the environment. The wait request is interpreted in the foreground as a request to go into low power mode.

There are a couple of ideas that need reiteration.

- The mechanism provided here is strictly to allow *injecting* conditions detected in the environment into background processing so they may be acted upon.
- This is *not* the only control mechanism needed. How background processing is to handle the notifications it receives has *not* been presented. That is another interesting topic and Part II of this book is devoted to explaining the background processing execution scheme. For now, working up from the bottom, the focus is only on the mechanism used to have foreground processing be responsive to the environment and to inform background processing of what has happened. As you might guess, since a queue was involved in the transport of notifications to the background, a queue is used in the background to create the system reaction to those notifications.

In the next two sections, fundamental background requests that support the flow of processing for the system are shown. First, we consider the problem of retrieving background notifications into background processing. [Previously](#), we saw how IRQ handlers, running in the foreground, place notifications in the background queue. Now we show how those notifications are retrieved into background processing so they may be acted upon.

Second, we consider the problem of how to handle the situation when there is no processing going on at all. When there is no work to be done, we want to place the system into a low power mode.

Retrieving Notifications into the Background

When background processing makes a request for system or device realm service, those requests are directed to a proxy function which executes the request for the background. Both sides must agree on the exact structure of data sent between the functions and the exact semantics of the operation performed. The SVC exception is used to “make the leap” from background to foreground processing. The flow of control from background to foreground is synchronous and runs to completion. Although the execution flow from background through to foreground proxy may be preempted by an IRQ, from the point of view of the background it appears as a conventional synchronous function invocation, *i.e.* nothing else goes on in the background until the foreground proxy function returns.

Similarly, when an IRQ handler detects an actionable condition, it issues a notification to the background. The notification requests background processing to execute a background notification proxy function which handles the notification. The particular background notification proxy function executed is one that was configured previously by background processing. The background processing must pre-arranged which notifications will arrive. Ultimately, it must be background processing that dispatches the notification proxy function to execute as part of the background reaction to the notification.

The support for this execution flow is achieved by a set of system services. We start with a service to obtain a notification from the **Background Notification Queue**.

```
<<system_services: system service definitions>>

/// Notification queue service.
pub const notify_queue = struct {
    <<notify queue service>>
};
```

We add the service name to the request enumeration.

```
<<sys svc service requests>>
```

```
get_notification,
```

```
<<sys svc service request prongs>>
.get_notification => &notify_queue.getNotification,
```

The following function, executed in the background, retrieves a notification.

```
<<notify queue service>>=
/// The `getNotification` service obtains the next background notification queued by
/// an IRQ handler. The request is placed in the memory pointed to by `notification`.
/// The function returns an error if it fails to obtain a notification. Otherwise,
/// the `notification` memory contains the device notification data.
pub fn getNotification(
    notification: *DevNotification,
) SvcError!void {
    const req_input = GetNotificationInputParam.marshal(.{
        .notification = notification,
    });

    try sysRealmSvcCall(.get_notification, @ptrCast(&req_input), null, null);
}
```

The effect of invoking `getNotification` copies a notification into the caller supplied location. Do not be confused by using a system realm request to retrieve a device realm notification. The notification queue is a *system realm* resource despite holding *device realm* information.

Since this service requests the notification be copied to the memory object pointed to by `notification`, we specify the pointer as part of an input parameter.

```
<<notify queue service>>=
pub const GetNotificationInputParam = DefineSvcRequestParam(struct {
    notification: *DevNotification,
});
```

The required actions by the notification foreground proxy function are resolved by the operations on the `dev_notify_queue`. The proxy function is little more than a convenient interface to the notification queue.

```
<<device proxies>>=
const notify_queue = struct {
    <<notify queue proxy>>
};
```

```
<<notify queue proxy>>=
/// The `getNotification` function is the foreground proxy function
/// for the `getNotification` service.
fn getNotification(
    : SysSvcRequest,
    input: ?*const anyopaque,
```

```

    _ : ?*anyopaque,
    _ : ?*anyopaque,
) SvcResult {
    const GetNotificationInputParam = svc_services.notify_queue.GetNotificationInputParam;

    const req_input, const result = GetNotificationInputParam.receive(input.?) ;
    if (result != .success) {
        return result;
    }

    return dev_notify_queue.recvNotification(req_input.notification);
}

```

Waiting for New Background Notifications

The other fundamental primitive required to complete the control flow handles how the core is directed to wait until new notification requests arrive.

When background processing has determined there is no addition work to be done, it waits until an interrupt posts a new background notification. To emphasize, simply dispatching background notifications is *not* the totality of the work performed by background processing. In whatever manner background processing determines that all the actions started by notifications are completed, it must then wait for new notifications to arrive. A new notification arrives only as a result of an IRQ handler placing a notification in the background notification queue.

The fundamental processor instruction for waiting is WFI, Wait For Interrupt. On most ARMv7-M SOC's, this causes the processor to go to sleep. Exactly what happens when the processor goes to sleep is not determined by the core, but by the SOC design which uses the core. In the case of the Apollo 3, the processor is put into a sleep mode and various clocks and other peripherals are powered down. Most SOC's use the WFI instruction as an indication to turn off the high speed clock to the processor core as the first step in reducing power consumption. The high speed clocks are some of the more power consuming components of an SOC. Some SOC's have more elaborate levels of power reduction that allow power management to be more than a simple binary choice. For the Apollo 3, the SOC performs all the hard work to manage power consumption using the execution of WFI as the hint that there is no ongoing computing.

```

<<system_services: system service definitions>>=

/// System wait service
pub const wait = struct {
    <<system wait service>>
};

```

Again, define a new service name and its foreground proxy.

```

<<sys svc service requests>>=
wait,

```

```

<<sys svc service request prongs>>=
.wait => &wait.wait,

```

The implementation of the background request is trivial since there are no arguments to the foreground proxy function which implements the wait.

```
<<system wait service>>=
/// The `wait` function requests the processor to cease execution until a
/// background notification has been inserted into the background notification
/// queue.
pub fn wait() SvcError!void {
    return sysRealmSvcCall(.wait, null, null, null);
}
```

The foreground proxy function requires not additional parameters. In this particular case, only the req parameter is of interest and the other parameters are passed null.

```
<<device proxies>>=
// System wait proxy
const wait = struct {
    <<system wait proxy>>
};
```

The corresponding proxy functions has a deceptively simple implementation that carries a lot of subtle information.

```
<<system wait proxy>>=
/// The `wait` function is the proxy counterpart of the `wait` service
/// and implements setting the processor into low power mode.
fn wait(
    : SysSvcRequest,
    : ?*const anyopaque,
    : ?*anyopaque,
    : ?*anyopaque,
) SvcResult {
    while (true) {
        // BEGIN CRITICAL SECTION
        {
            v7m.disableIrq(); ❶
            defer v7m.enableIrq();

            if (dev_notify_queue.empty()) {
                const apollo_specific: *volatile u32 = @ptrToInt(0x5fff_0000); ❷
                _ = apollo_specific.*;

                v7m.dsb(); ❸
                v7m.wfi();
            } else {
                break;
            } ❹
        }
        // END CRITICAL SECTION
    }
}
```

```

    return .success;
}

```

- ① After this point, no IRQ's will cause preemption. It is safe to test the state of the background notification queue.
- ② There is a strange code sequence in the Ambiq HAL code which reads a memory location which is given the symbol, SYNC_READ in the code. A comment states that the read "... will hold the bus until all the queued write operations have completed." The address does not appear in the SVD file and there seems to be no mention of it in the reference manual. The implementation in the HAL is rather strangely coded, but the intent seems to be to read the memory location and discard any value. Presumably, the read action has some side effect in the hardware. The memory read is included here because it was found in the Ambiq HAL code.
- ③ ARM recommends a data synchronization barrier instruction to ensure that all data writes have been completed before executing WFI. In this particular case, this is not necessary, but at the cost of one instruction, the recommendation is followed.
- ④ It is possible for an IRQ handler to run and *not* place a notification in the background notification queue, *i.e.* an interrupt source may service device internals and *not* need to issue a notification. The core will have awakened on any interrupt. For this reason, the code loops back and tests the queue state again. The loop is terminated only when there is a notification in the queue.

The implementation of wait proxy has several circumstances that it must accommodate.

- There is a *race condition* between when the background processing determines that the background notification queue is empty and when the foreground proxy code for waiting executes. An interrupt can occur between those times. Typically, when this function is invoked, the background notification queue has been emptied. Therefore, before sleeping it is necessary to check that the queue remains empty of any pending notifications.
- The WFI instruction has a set of conditions^[4] which must be met before the processor core wakes back up once it has been put to sleep. For this use case, a pending exception whose priority is such that it would preempt any currently active exceptions governs what happens. This is one reason why the SVC exception priority was set to be lower than that of IRQ's.
- When the processor goes to sleep in the Apollo 3 SOC, clocks are turned off and the memory buses no longer function. This means that attached debuggers are not able to read information out of the system.

The key logic used to put the core to sleep and *not* lose the race with an interrupt rests with the fact that the WFI instruction ignores the effect of the PRIMASK register when deciding if a pending exception will cause the processor to wake back up. The logic works as follows.

1. Set PRIMASK to 1 (as is done in the v7m.disableIrq function). This starts a critical section where no exception with a configurable priority will preempt execution.
2. Check if the background notification queue is still empty. If so, then the race between the background deciding there is no additional work to do and an interrupt coming along and pending an exception has been "won". With PRIMASK set to 1, conditions are set to use WFI to put the core to sleep and there will be no preemption before the WFI instruction is executed^[5].
3. After executing WFI, the core wakes up if any exception is pending which has a higher execution priority than the current exception (in this case the current exception is the SVC exception). This determination ignores the setting of PRIMASK, *i.e.* WFI ignores the fact that the pending exception may not be made active immediately when the processor wakes back up. This is critical, because between executing the instructions which determine that the background notification queue is empty and executing the WFI instruction, an exception can be made pending. Because PRIMASK is set to 1, the exception is not made active, but if there is a

pending, higher priority exception when WFI executes, the execution continues since that is one of the conditions of WFI to wake up the processor. This behavior ensures that the race between executing the instructions which decide that the conditions for sleeping are appropriate and executing WFI does not affect the execution path even if an exception is made pending after the test but before executing WFI. Thus there are two races which must be “won” before the core is put to sleep. Arbitrating this situation is like arbitrating between simultaneous interrupt requests at the same priority, in that it can only be handled by innate core logic.

4. If there is a pending, higher priority exception either when WFI is executed or anytime thereafter, the core resumes (or simply continues) execution.
5. When execution resumes, PRIMASK is still set to 1 and, despite the fact that a higher priority exception is pending, the effects of PRIMASK prevents the exception from becoming active and no execution preemption occurs. This allows us to perform any other actions necessary to get the core fully running. On some SOC’s this might involve restarting clocks, crystals, or any of a number of SOC specific actions. For the Apollo 3, there are no additional actions to be performed.
6. When all is ready, code can set PRIMASK back to 0, which ends the critical section and enables exception preemption. The highest priority pending exception is then made active.

Despite all the convolutions in the logic about putting the core to sleep, the code to implement it is deceptively simple.



The wait proxy function is the only place in the system where PRIMASK is set to one for the purposes of preventing all preemption and interleaving of instruction execution for the purposes of controlling execution flow. Avoiding the disabling all exceptions helps to maintain the best responsiveness to the system environment and the most protection available from system fault detection. Where there is shared access to data between preemptive and non-preemptive execution, minimizing the scope of the effect is preferred. For example, raising the current execution priority to either inhibit interrupts in general or disabling a particular interrupt narrows the scope of inhibiting preemption. The goal is to minimize the time when all configurable exceptions are inhibited and to let the finer-grained execution prioritization available from the processor architecture provide the arbitration for execution preemption.

Control Services

Background processing must treat notifications coming from the foreground with a sense of urgency and dispatch the notifications in a timely manner. After all, notifications indicate conditions to which a reactive system must take action. Timely execution of the notifications implies dispatching all the notifications at once until there are no more in the notification queue. In this section, we show the code which controls the fundamental operations of the system in its response to the environment.

```
<<system_services: system service definitions>>

/// Execution Control services.
/// Services in this group do _not_ have foreground proxies so are
/// in some sense pseudo system services. However, they have
/// precise interactions with other system services and are
/// fundamental to execution flow control.
pub const exec_control = struct {
    <<service background service>>
    <<run wait service>>
```

```
<<variable sync service>>
};
```

Interleaving Background Notifications and Background Processing

All the pieces needed to design the mechanism for interleaving background notifications with the background processing that responds to those notifications are in place. For simpler applications (e.g. the examples in this part of the book), the background notification proxy functions can often perform all the computations necessary to respond to the notification. But for non-trivial applications, it is necessary to allow background processing an additional opportunity to complete the processing required by a notification. For example, a notification may start some behavior, but additional computations and perhaps additional notifications are required to complete the system's response.

To interleave background processing with the dispatch of background notifications, the concept of a background service function is introduced. The purpose of the background service function is to perform a *quantum of work* to resolve the response to background notifications. The definition of quantum of work is purposely vague at this time. In Part II of the this book, a detailed definition is presented.

The background service function returns a boolean value to indicate that a work quantum was actually performed. A true return value indicates some work was done and there might be more that needs to be done. So, the background service function need to be invoked again. A false return value indicates that the function performed no work and there is no additional work remaining. A default background service function that performs no work at all and simply returns false is provided here. By using a weak declaration, it can be overridden for more complex background processing purposes.



The `serviceBackground` function is the only way in which background processing is given control of the processor. There is no other mechanism in this design for background processing to gain control of the flow of execution.

Applications which perform background service work, supply their own version of this function. Again, in Part II of this book the background service function which completes the execution model for this design is shown. For now, the default function is sufficient to exercise examples and testing code.

The fundamental execution sequencing is then a “run/wait” loop which:

1. Dispatches all queued background notifications.
2. Invokes the background service function.
3. Checks if there has been a request to exit the execution loop.
4. If the background service indicated that it did not perform any work, then wait for new background notifications.

The following figure is a schematic diagram of the logic.

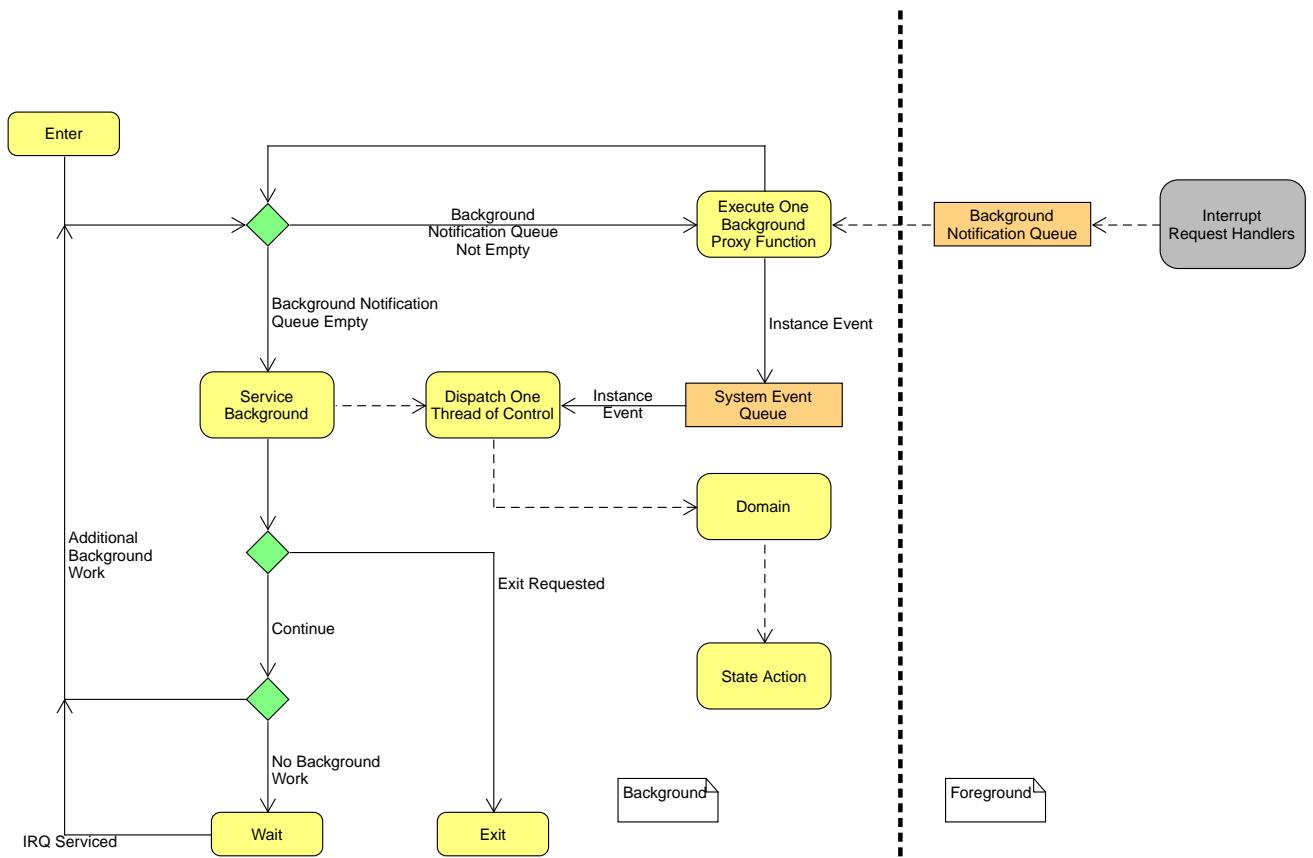


Figure 9. Overview of Run / Wait Execution Loop

In this diagram, the decisions around the first green diamond (labeled "1") create a loop that executes all the background notifications contained in the Background Notification Queue. This means that notifications coming from the environment (via the IRQ handlers) take priority. These notifications are used to trigger background processing. It is important to keep in mind that interrupts are always enabled when running background code.

After the notification queue is empty, the background is serviced and possibly performs a quantum of work.

At the second green diamond, the decision is made as to exit the control loop or not. We use a simple boolean variable to make the determination. Starting with a false value, if any background notification proxy or any code executed when the background is serviced set the value of the variable to true, then the control loop exits and returns to its caller. This can be used, for example if the control loop is entered from `main`, to exit a program gracefully giving the caller a chance to decide how to terminate gracefully (the example given [below](#) does this).

The decision at the third green diamond tests whether there is additional background servicing that is required. If not, then the control loop waits for new background notifications. If the background service indicated that it execution a quantum of work, then control flows back to the loop that empties the background notification queue.

With this diagram in mind, three functions are provided to form the system control loop.

```

<<run wait service>>
/// The `dispatchNotifications` function retrieves and dispatches any

```

```

/// notifications in the Background Notification Queue until the
/// queue is empty.
pub fn dispatchNotifications() SvcError!void {
    var notification: DevNotification = undefined;
    while (true) {
        notify_queue.getNotification(&notification) catch |err| {
            switch (err) {
                SvcError.retry_operation => break, ①
                else => return err,
            }
        };
        notification.proxy(@ptrCast(@alignCast(&notification)));
    }
    // At this point, the background notification queue is empty.
}

```

- ① The `retry_operation` error indicates that the call was successful but no notifications were present at this time and thus the request should be tried again some time later.

The default servicing of the background does nothing.

```

<<service background service>>=
pub fn serviceBackground() callconv(.c) bool {
    return false;
}
comptime {
    @export(&serviceBackground, .{
        .name = "serviceBackground",
        .linkage = .weak,
    });
}

```

Finally, we can put together the pieces into the function which runs the control loop for the system.

```

<<run wait service>>=
/// The `runWait` function is the fundamental execution loop for the system.
/// It operates by dispatching all background notifications and then servicing
/// the background. If there is no remaining work to be done, either as
/// device notifications or other background processing, then
/// it executes the `wait` system service which halts the processor until
/// additional device notifications are made. If any background processing,
/// either as part of executing a device notification or background servicing,
/// sets the value pointed to by `exit_req` to `true`, then
/// the execution loop is exited. This function returns an `SvcError` error value
/// if any system calls fail unexpectedly.
pub fn runWait(
    exit_req: *volatile bool,
) SvcError!void {
    exit_req.* = false;
    while (true) {
        try dispatchNotifications();
    }
}

```

```

const performed_bg_action = serviceBackground(); ①

if (exit_req.*)
    break; ②

if (!performed_bg_action) {
    try wait.wait(); ③
}
}
}

```

① Transfer control to background processing to allow it to generate any required responses.

② It is important to check for termination after all notifications have been dispatched and the system background has been serviced. These are the two actions which cause code to run which could modify the value of the variable pointed to by `terminate`.

③ If dispatching of the background notifications did not result in the background service performing any work, then there is no remaining work to do, so it is time to wait. Note there is a race between deciding there is nothing left to do and an IRQ Handler posting a background notification (interrupts are enabled here). If an IRQ handler did “win the race” and post a background notification, `wait` returns immediately.

There are a couple of noteworthy implications about the `runWait` function:

- The execution loop exits if background processing writes to the variable pointed to by `terminate`. This allows for graceful termination of the loop. When `runWait` is used as the “main” loop for an application, it is typically invoked from `main` and if it returns, the application has an opportunity to execute clean up code before returning from `main` and thus terminating the program. This is also useful for test code where it is necessary to enter the execution loop but it is also necessary to regain control over the execution flow to determine the status of a test.
- It is possible to invoke `runWait` recursively, *i.e.* it is possible for background processing to “busy wait” in place and contrive to use the termination variable to synchronize, eventually, to a particular linear flow. This usage is subject to abuse and should generally be avoided, *i.e.* there should be only a single invocation of `runWait` in a program. But when a particular interactive situation arises where processing must wait for some notification and then proceed in sequence, it is an available tool.

Simple Synchronization with the Run / Wait Loop

For simple situations, the background notification closure value can be set to a pointer to a termination variable and then have the background notification proxy use the closure data pointer to request termination of the run / wait loop. The following background proxy function implements this simple case.

```

<<variable sync service>>=
/// The `svcVarSyncNotifier` function is a simple variable synchronization proxy
/// which assumes the closure data in `notification` is a pointer to a boolean
/// variable. When dispatched, the function sets the variable pointed to by the
/// closure data to `true`.
pub fn svcVarSyncNotifier(
    /// A pointer to the notification information for a background notification request.
    notification: *const anyopaque,
) void {
    const dev_notify: *const DevNotification = @ptrCast(@alignCast(notification));

```

```

const sync_var_ptr: *volatile bool = @ptrToInt(dev_notify.closure);
sync_var_ptr.* = true;
}

```

Code Layout

The layout of code files for this chapter is more complicated than seen previously. In the [next chapter](#), the foreground/background split is used to control the placement in memory that enforces memory usage constraints between foreground and background execution. Since foreground and background requests and proxies must execute in different memory privilege regions, the code must be split between them.

The arrangement of this section starts with the system configuration file that gives a set of parameters that can be used to control system resources. Next is the code required to make system or device service requests from the background. This is followed by the code which contains the proxies required to fulfill the system and device requests.

System Config

```

<<system_config.zig>>=
<<edit warning>>
<<copyright info>>

///! The `system_config.zig` file contains a set of constants that are used
///! to configure various aspects of the processor code to the manner
///! required by this design.

const v7m = @import("v7m");

<<system_config: constants>>

```

System Services

```

<<svc_services.zig>>=
<<edit warning>>
<<copyright info>>

///! The `svc_services.zig` file contains the definition of services available from
///! the system foreground and accessed via the SVC instruction.

const std = @import("std");
const mem = std.mem;
const math = std.math;
const fmt = std.fmt;
const meta = std.meta;
const time = std.time;
const mmio = @import("mmio");
const v7m = @import("v7m");
const apollo3 = @import("apollo3");

```

```

<<system_services: interface definitions>>

///////////////////
// Available system services.
///////////////////

<<system_services: system service definitions>>

///////////////////
// Available device services.
///////////////////

<<system_services: device service definitions>>

```

System Proxies

```

<<svc_proxies.zig>>=
<<edit warning>>
<<copyright info>>

///! The `svc_proxies.zig` file contains the functions which serve as the
///! foreground proxies for both system realm services and device realm
///! services.

const std = @import("std");
const Io = std.Io;
const mem = std.mem;
const meta = std.meta;
const math = std.math;
const enums = std.enums;
const time = std.time;
const builtin = @import("builtin");
const mmio = @import("mmio");
const v7m = @import("v7m");
const apollo3 = @import("apollo3");
const config = @import("config");
const svc_services = @import("svc_services");
const system_config = @import("system_config.zig");

const SvcResult = svc_services.SvcResult;
const SysSvcRequest = svc_services.SysSvcRequest;
const SysSvcRequestProxy = svc_services.SysSvcRequestProxy;

const DevSvcClass = svc_services.DevSvcClass;
const DevSvcInstance = svc_services.DevSvcInstance;
const DevSvcRequest = svc_services.DevSvcRequest;
const DevSvcRequestProxy = svc_services.DevSvcRequestProxy;
const DevNotification = svc_services.DevNotification;
const DevSvcNotifyProxy = svc_services.DevSvcNotifyProxy;

const memcpyToUnpriv = svc_services memcpyToUnpriv;
const memcpyFromUnpriv = svc_services memcpyFromUnpriv;

```

```
<<svc handler>>
<<system proxies>>
<<device access>>
<<device proxies>>
<<notify queue>>
```

Example

The following example shows all the concepts from this part of the book in action. The program initializes the watchdog timer and starts it. As the watchdog IRQ posts notifications, they are dispatched and the system is put to sleep waiting for the next notification.

```
<<crossing-the-divide-test.zig>>=
<<edit warning>>
<<copyright info>>

//! This file contains a simple test application demonstrating the use
//! of the system `runWait` function to synchronize to the notification
//! that the watchdog timer needs to be restarted.
comptime {
    _ = @import("start_main");
}

const std = @import("std");
const config = @import("config"); // This file contains command line option values.
const svc_services = @import("svc_services");
const DevNotification = svc_services.DevNotification;

pub fn main() !void {
    try svc_services.wdog.init(0, 4000, 3500, wdogDevNotifier, 0);
    try svc_services.wdog.start(0, config.allow_wdog_reset); ①

    var forever: bool = false;
    try svc_services.exec_control.runWait(&forever); ②
}
```

- ① Recall that the Apollo 3 watchdog timer cannot be set to halt when the debugger is active. To allow for breakpoint debugging, we make this an option.
- ② Since `forever` is an automatic variable and there is no watchdog notification proxy which can set its value to true, the call to `runWait` only returns if there is an error.

The background proxy function for the watchdog performs the restart of the watchdog timer along with printing some information. Again, we admonish that in a larger system, the watchdog restart must be performed as part of the background service and *not* as part of the background notification proxy. In this example, restarting as part of the background notification is necessary because there is no other place. But, in general, this is insufficient to deal with potential problems in background service code.

```
<<crossing-the-divide-test.zig>>=
/// The `wdogDevNotifier` function is dispatched when the notify timeout of the
/// watchdog timer happens. It issues a restart to the watchdog timer to ensure
```

```

/// that the system is not reset.
fn wdogDevNotifier(
    notification: *const anyopaque,
) void {
    const wdogNotification: *const DevNotification = @ptrCast(@alignCast(notification));
    svc_services.wdog.restart(wdogNotification.instance) catch
        @panic("Failed to restart watchdog timer");
}

```

- [1] More properly, the system is configured so that only handler mode can perform system and peripheral register access.
- [2] This is strictly true in a flat interrupt priority design. But even in a nested interrupt design, many interrupts run at the same priority.
- [3] Of course, the reality is more complicated. There are means for application code to request an IRQ to be made pending. But even that requires system code to have initialized a register to allow it.
- [4] The details of the semantics and recommendations for using WFI are contained in section B1.5.19 of the ARMv7-M Architecture Reference Manual, (ARM DDI 0403E.d ID070218, p. B1-562).
- [5] Well, not absolutely none. If things go really wrong, *priority escalation* will force a HardFault which would preempt the execution. But a HardFault is considered to be unrecoverable and the system is destined for a reset operation

Batten Down the Hatches

In the [second chapter of this book](#), a significant system design decision was implemented that separated privileged from unprivileged execution. Some of the implications of that decision were shown when the [foreground/background](#) background request / foreground proxy / background notification scheme was built. This chapter shows how execution is further constrained by using the Memory Protection Unit (MPU).

Partitioning Memory Usage

The Background Notification Queue concept should not be conflated with the concept of execution privilege. The queue would be necessary even if privileged and unprivileged execution had not been separated. An IRQ handler must respond to the environment and queuing a notification allows the system to track the environment while other work is ongoing. The separation between privileged and unprivileged execution forced the use of the SVC instruction to make requests to the foreground. The SVC instruction is the mechanism specifically designed into the processor architecture to support obtaining privileged services.

As part of the foreground/background interactions, care was taken to ensure that all data moved from the foreground into the background was done in an unprivileged manner. Specially designed processor instructions were used to ensure that the background could not *trick* the foreground into writing into memory to which it otherwise should have no access.

It is also fitting to reiterate that the continuing efforts to constrain the application execution are *not* directed primarily to thwart malicious coding or to make the system suitable for executing application code of arbitrary provenance. The efforts are directed at early detection of software bugs and creating an environment where the undiscovered bugs in a deployed system are less likely to cause damage and are more easily diagnosed.

Until now, the mitigation efforts have used processor architectural features to constrain execution and trap any faults raised by the processor when it detects that software is operating the core improperly. The mitigation given by processing privilege alone is not enough to keep background execution constrained in terms of its memory accesses. Now, attention turns to how memory is used and to design a scheme where unprivileged execution is denied access to privileged memory areas.

Types of Memory Usage

It would be most convenient if all memory were the same and it was not necessary to make up special rules for memory usage. As a first order approximation, addresses are just addresses. Loads and stores from and to memory are not generally concerned with all the different characteristics that the underlying memory can have.

Upon closer examination, the following types of values are placed in processor memory.

Instructions

All modern computers store their program in memory and the processor fetches and executes instructions from that memory^[1].

Data

Data values manipulated by the running program are stored in processor memory. Some values that don't change during the running of an application are stored in memory which does not necessarily allow writing. This distinction is typically driven by the economics of read-only

memory being significantly less expensive (in both material cost and power) than read-write memory.

Peripheral access

Memory also serves as the communications interface between the processor core and its peripheral devices.

Ignoring run time code generation and abominations such as self-modifying code, memory devoted to instructions is only ever read as part of an instruction fetch by the processor. In a microcontroller system where there typically is no external storage, memory holding instructions must survive power cycles. Modern microcontroller chips usually use some form of programmable memory which survives power cycles and which can be erased and re-programmed, e.g. flash memory.

Assuming processor instructions are not generated during run time, memory can be further restricted so that no execution from memory devoted to data can occur. This is *not* a general rule for all systems. There are rarer situations where code is placed in RAM for execution. For example, on some microcontroller chips, programming flash memory at run time requires executing the programming code from RAM since you would otherwise be erasing and controlling the same memory area from which instructions are fetched. Some flash memory controllers in chips cannot support instruction fetch and programming access concurrently.

Modern microcontroller chips map peripheral device control to memory locations. Older microprocessor designs sometimes used a processor I/O bus for device access. This too was driven by the economics of the situation. Older designs did not include built-in peripheral devices, and simplified I/O bus designs required less circuitry to integrate external devices to the system. Those circumstances no longer apply and devices present an interface which is mapped into the processor address space as “registers.”

With these considerations, a first partitioning of memory is:

- Read only memory for instructions.
- Read only memory for constant data.
- Read/write memory for mutable data.
- Peripheral device access.

Consideration for separating privileged and unprivileged execution leads to further partitioning of memory usage. It does not exactly double the categories since all peripheral device access is only suitable for privileged execution. Considering two different execution privileges, a suitable partitioning of memory would be:

- Privileged read only memory for instructions.
- Privileged read only memory for constant data.
- Privileged read/write memory for mutable data.
- Privileged peripheral device access.
- Unprivileged read only memory for instructions.
- Unprivileged read only memory for constant data.
- Unprivileged read/write memory for mutable data.

The following figure shows the layout of the memory partitioning.

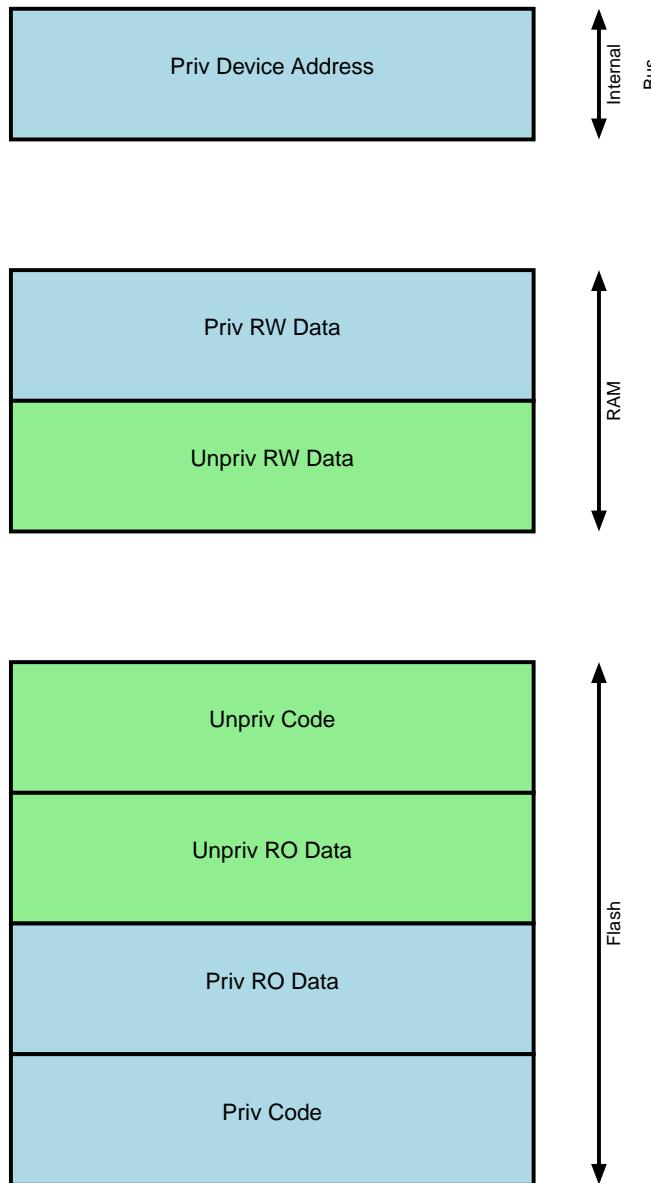


Figure 10. Memory Usage Partitioning

Enforcing Memory Usage

Given the goals presented in the previous section, the design scheme to enforce the separation of code and data between privileged and unprivileged execution involves:

- Programming the Memory Protection Unit (MPU) of the Cortex-M4 core to enforce the separation scheme.
- Separating the privileged from the unprivileged code/data so they may be placed in different memory areas.
- Constructing a linker script to place the code and data in the proper memory locations.

The details of programming the MPU are not discussed here. Code is presented to accomplish the MPU programming later. There are many good references as to what the MPU does and how it is controlled. The functions provided by CMSIS are used to get the register and bit fields correct. The focus in this section is on those controls of the MPU which affect how the designed memory usage

scheme is enforced.

MPU Control Parameters

There are three characteristics about the operation of the MPU which are of concern.

Memory region

A memory region is the MPU concept of how portions of memory are treated. The MPU defines eight memory regions. Memory regions have a size which can range from 32 bytes to 4 GiB. For a given region size, the memory address for the region *must* be on an address boundary which is aligned to the size. This manner in which the MPU operates constrains allocation of memory regions and their placement by the linker and implies that larger regions have fewer places in memory where they can be placed. Memory regions also have a priority, where the highest numbered region takes precedence if multiple regions overlap a given portion of memory. This design does not use overlapping memory regions, so region numbering is somewhat arbitrary.

Access permissions

The access permissions on a region have the property that if any unprivileged access is allowed to a region, some form of privileged access is also granted. A region may prohibit unprivileged access, but the region remains accessible by privileged code. For this design, memory regions intended for unprivileged access, are granted the same permissions to privileged access. For memory regions intended for privileged access, no unprivileged access is granted.

Disallowing execution

Whether or not instructions may be fetched from a region is an independent attribute of an MPU region. In this design, all execution is disallowed except for memory regions specifically designated to hold instructions.

The following table shows how the design requirements for memory protection are mapped onto MPU regions and gives the needed MPU access permissions to implement the protection scheme.

Region	Usage	Privileged Access	Unprivileged Access
0	Privileged Code	Read Only	None
1	Privileged RO Data	Read Only, No Execute	None
2	Privileged RW Data	Read/Write, No Execute	None
3	Peripheral Devices	Read/Write, No Execute	None
4	Unprivileged Code	Read Only	Read Only
5	Unprivileged RO Data	Read Only, No Execute	Read Only, No Execute
6	Unprivileged RW Data	Read/Write, No Execute	Read/Write, No Execute

Table 3. MPU Region Usage

There remain two tasks to consider before the necessary code to program the MPU can be designed.

1. Privileged code/data must be separated from unprivileged.
2. A linker script must be written to place the previously characterized memory sections into the physical memory provided by the SOC.

Linker Script

The linker script presented here is quite different from the one used in [Chapter 1](#). That linker script was not concerned about execution privilege and treated memory as either flash or RAM. The only complications in the script arose from our defining memory regions which survive reset.

The linker script presented here has the same general layout, but with some important differences pointed out below.

```
<<apollo3_priv.ld>>=
/*
<<copyright info>>
*/
<<apollo3 priv: entry point>>
<<apollo3 priv: memory layout>>
<<apollo3 priv: cross reference checks>>
<<apollo3 priv: section descriptions>>
```

Entry point

The entry point remains the same, namely the `Reset_Handler`.

```
<<apollo3 priv: entry point>>=
ENTRY(Reset_Handler)
```

Memory regions

As mentioned previously, MPU regions must be aligned to an address which is a multiple of their length, e.g. a 64K code section must be aligned to a 64K address boundary. This complicates the placement of memory regions in the flash and RAM of the SOC. When region alignment was not a concern, it did not matter into which parts of flash or RAM the various bits of code and data were placed. Now it matters greatly.

The chosen mechanism to solve the alignment of allocated memory to MPU region addresses is to define, as part of the memory specified to the linker, a separate block of memory with its own location and size. It is not an ideal solution. All the allocation calculations must be done by hand. *A priori* it is difficult to know how much memory should be allocated to a particular usage. However, managing the sections manually makes clear any holes in our memory map and the linker emits an error if any of the defined memory boundaries overflows.

```
<<apollo3 priv: memory layout>>=
MEMORY
{
    priv_text (rx) : ORIGIN = 0x00010000, LENGTH = 128K
    priv_rodata (r) : ORIGIN = ORIGIN(priv_text) + LENGTH(priv_text), LENGTH = 64K
    unpriv_rodata (r) : ORIGIN = ORIGIN(priv_rodata) + LENGTH(priv_rodata), LENGTH = 64K
    unpriv_text (rx) : ORIGIN = 0x00080000, LENGTH = 512K

    unpriv_rwdata (rw) : ORIGIN = 0x10000000, LENGTH = 256K
    priv_rwdata (rw) : ORIGIN = ORIGIN(unpriv_rwdata) + LENGTH(unpriv_rwdata), LENGTH = 64K
}
```

- ① The `unpriv_text` section must be on a 512K boundary because of the rules of the memory protection unit.

This scheme allocates all of memory to each type of memory. (note peripheral device access has a defined memory location and is not part of the program linkage). The memory region sizes are usually 64K with larger sizes for unprivileged code and data as well as privileged code. These sizes were determined considering the sizes of code and data generated by the Zig compiler in its various debug and release modes. The expressions giving the origin address of the memory blocks ensures they are placed sequentially in either flash or RAM memory.

Prohibiting cross references

The linker `NOCROSSREFS_T0` command is used to request the linker to ensure no symbol cross references occur from any of the unprivileged memory blocks to privileged ones. The command can be used to give link-time errors if unprivileged code tries to reference directly any privileged memory. Recall, that all access to privileged execution happens via the SVC interface which is a specific processor instruction and *not* a function with an external symbol.

```
<<apollo3 priv: cross reference checks>>
NOCROSSREFS_T0(.priv_text .unpriv_text .unpriv_rodata .unpriv_rwdata)
NOCROSSREFS_T0(.priv_rodata .unpriv_text .unpriv_rodata .unpriv_rwdata)
NOCROSSREFS_T0(.priv_data .unpriv_text .unpriv_rodata .unpriv_rwdata)
NOCROSSREFS_T0(.main_stack .unpriv_text .unpriv_rodata .unpriv_rwdata)
NOCROSSREFS_T0(.noinit_data .unpriv_text .unpriv_rodata .unpriv_rwdata)
NOCROSSREFS_T0(.priv_bss .unpriv_text .unpriv_rodata .unpriv_rwdata)
NOCROSSREFS_T0(.noinit_bss .unpriv_text .unpriv_rodata .unpriv_rwdata)
```

Defining the linker sections

For each memory block, whose access is controlled with the MPU, has a section definition to describe the content of the block. Note that the names for linker sections are the same as the name of the memory block, adding the conventional leading period. This is just a convention meant to help keep track of what is going where.

When defining the contents of the memory blocks, linker generated are used to generate symbols giving the starting address and length of the various memory blocks. These linker symbols are used to configure the MPU. This gives a single location, namely the linker script, where the location and usage of memory is defined.

There are a separate set of linker script commands for each memory block defined above. Note, there are no linker commands associated with the memory for peripheral devices. Peripheral devices addresses are only known by *dead reckoning*, i.e. by pre-processor defined constants.

```
<<apollo3 priv: section descriptions>>
SECTIONS
{
    /DISCARD/ : { *(.ARM.extab* .gnu.linkonce.armextab.*) }
    /DISCARD/ : { *(.ARM.exidx* .gnu.linkonce.armexidx.*) }
    <<priv text section>>
    <<priv rodata section>>
    <<unpriv text section>>
    <<unpriv rodata section>>
    <<priv ram sections>>
```

```

<<unpriv ram sections>>
<<mpu region symbols>>

PROVIDE(end = .) ;
}

```

As in the previous linker script, the exception vector table must be placed at the first locations of memory loaded.

```

<<priv text section>>=
.priv_text :
{
    KEEP(*startup_apollo3.o(.vectors))

    . = ALIGN(4) ;
    *startup_apollo3.o(.text*)
    *system_mem_protect_init.o(.text*)
    *system_apollo3.o(.text*)
    *svc_proxies.o(.text*)

    . = ALIGN(4) ;
    PROVIDE(__priv_text_end__ = .) ;
} > priv_text

```

Following the same pattern, the privileged read only data is loaded. Additionally, the various copy tables used to initialize RAM at start up time are created. Recall that the [code](#) used to copy initializers to RAM supports multiple entries in order to place the initializers into multiple segments of RAM. This is not by accident.

```

<<priv rodata section>>=
.priv_rodata :
{
    *startup_apollo3.o(.rodata*)
    *system_mem_protect_init.o(.rodata*)
    *system_apollo3.o(.rodata*)
    *svc_proxies.o(.rodata*)

    <<memory initialization tables>>

    PROVIDE(__priv_rodata_end__ = .) ;
} > priv_rodata

```

The definitions of the copy and zero table descriptors follows the same conventions as our previous linker script. The only difference is the addition of multiple segments to accommodate the restrictions on access to the memory.

```

<<memory initialization tables>>=
. = ALIGN(4) ;

__data_copy_table_start__ = . ;
LONG(LOADADDR(.priv_data))

```

```

LONG(ADDR(.priv_data))
LONG(SIZEOF(.priv_data) / 4)

LONG(LOADADDR(.unpriv_data))
LONG(ADDR(.unpriv_data))
LONG(SIZEOF(.unpriv_data) / 4)
__data_copy_table_end__ = . ;

__zero_table_start__ = . ;
LONG(ADDR(.priv_bss))
LONG(SIZEOF(.priv_bss) / 4)

LONG(ADDR(.unpriv_bss))
LONG(SIZEOF(.unpriv_bss) / 4)
__zero_table_end__ = . ;

__stack_table_start__ = . ;
LONG(ADDR(.process_stack))
LONG(SIZEOF(.process_stack) / 4)

LONG(ADDR(.main_stack))
LONG(SIZEOF(.main_stack) / 4)
__stack_table_end__ = . ;

__noinit_data_copy_table_start__ = . ;
LONG(LOADADDR(.noinit_data))
LONG(ADDR(.noinit_data))
LONG(SIZEOF(.noinit_data) / 4)
__noinit_data_copy_table_end__ = . ;

__noinit_bss_table_start__ = . ;
LONG(ADDR(.noinit_bss))
LONG(SIZEOF(.noinit_bss) / 4)
__noinit_bss_table_end__ = . ;

```

Unprivileged code and data continues to be loaded in the previous manner. Note here the `EXCLUDE_FILE(*privileged.o)` construct ensures that no privileged code or data is placed in the unprivileged memory areas.

```

<<unpriv text section>>=
.unpriv_text :
{
    *(EXCLUDE_FILE ( *startup_apollo3.o *system_apollo3.o *system_mem_protect_init.o
*svc_proxies.o) .text*)

    . = ALIGN(4) ;
    PROVIDE(__unpriv_text_end__ = .) ;
} > unpriv_text

```

```

<<unpriv rodata section>>=
.unpriv_rodata :

```

```
{
    *(EXCLUDE_FILE ( *startup_apollo3.o *system_apollo3.o *system_mem_protect_init.o
*svc_proxies.o) .rodata*)
} > unpriv_rodata
```

Access to the Build ID is allowed as unprivileged. Application code should be able to report the Build ID without having to resort to privileged execution.

```
<<unpriv rodata section>>=
.build_id : ALIGN(4)
{
    PROVIDE(SystemBuildIDNote = .) ;
    *(.note.gnu.build-id)
} > unpriv_rodata
.= ALIGN(4);
PROVIDE(__unpriv_rodata_end__ = .);
```

Privileged RAM consists of not only the data and bss sections for the privileged code, but also of the main stack and those memory areas which survive reset. This implies that access to the contents of memory which survives reset is intended to go through privileged code.

```
<<priv ram sections>>=
.main_stack (NOLOAD) : ALIGN(8)
{
    PROVIDE(__main_stack_limit__ = .) ;

    KEEP(*(.main_stack*))

    . = ALIGN(8) ;
    PROVIDE(__main_stack_top__ = .) ;
} > priv_rwdata

.priv_data : AT(LOADADDR(.priv_rodata) + SIZEOF(.priv_rodata)) ALIGN(4)
{
    PROVIDE(__priv_data_start__ = .) ;

    *startup_apollo3.o(.data*)
    *system_mem_protect_init.o(.data*)
    *system_apollo3.o(.data*)
    *svc_proxies.o(.data*)

    . = ALIGN(4) ;
    PROVIDE(__priv_data_end__ = .) ;
} > priv_rwdata

.noinit_data : AT(LOADADDR(.priv_data) + SIZEOF(.priv_data)) ALIGN(4)
{
    PROVIDE(__noinit_data_start__ = .) ;

    *(.noinit_data*)

    . = ALIGN(4) ;
```

```

    PROVIDE(__noinit_data_end__ = .) ;
} > priv_rwdata

.priv_bss (NOLOAD) : ALIGN(4)
{
    PROVIDE(__priv_bss_start__ = .) ;

    *startup_apollo3.o(.bss*)
    *system_mem_protect_init.o(.bss*)
    *system_apollo3.o(.bss*)
    *svc_proxies.o(.bss*)

    . = ALIGN(4) ;
    PROVIDE(__priv_bss_end__ = .) ;
} > priv_rwdata

.noinit_bss (NOLOAD) :
{
    PROVIDE(__noinit_bss_start__ = .) ;
    PROVIDE(noinit_status = .) ;
    . += 8 ; /* 8 == sizeof(NoinitSegmentStatus) */

    *(.noinit_bss*)

    . = ALIGN(4) ;
    PROVIDE(__noinit_bss_end__ = .) ;
} > priv_rwdata

```

Unprivileged RAM access is restricted to the data and bss segments for unprivileged code and the process stack used by unprivileged code.

```

<<unpriv ram sections>>=
.process_stack (NOLOAD) : ALIGN(8)
{
    __process_stack_limit__ = . ;
    KEEP(*(.process_stack*))

    . = ALIGN(8) ;
    __process_stack_top__ = . ;
} > unpriv_rwdata

.unpriv_data : AT(__unpriv_rodata_end__) ALIGN(4)
{
    PROVIDE(__unpriv_data_start__ = .) ;

    *(EXCLUDE_FILE ( *startup_apollo3.o *system_apollo3.o *system_mem_protect_init.o
*svc_proxies.o) .data*)

    . = ALIGN(4) ;
    PROVIDE(__unpriv_data_end__ = .) ;
} > unpriv_rwdata

```

```

.unpriv_bss (NOLOAD) : ALIGN(4)
{
    PROVIDE(__unpriv_bss_start__ = .) ;

    *(EXCLUDE_FILE ( *startup_apollo3.o *system_apollo3.o *system_mem_protect_init.o
*svc_proxies.o ) .bss*)

    . = ALIGN(4) ;
    PROVIDE(__unpriv_bss_end__ = .) ;
} > unpriv_rwdata

```

Global symbols provided by the linker are used to configure the MPU. The values of these symbols are used directly for the MPU configuration and to accomplish that the linker must do some computations. Recall that the MPU requires knowledge of the size of a memory region to which protections are applied. The region size is encoded in a particular way since it is actually a bit field in a larger register context. Hardware encodings are usually made for the convenience of hardware operation. The encoding of region size is a small unsigned integer number which represents the number of bits in a memory address which are ignored when considering whether a given address is controlled by an MPU region **minus 1**. For example, a 32 Byte MPU memory region size is specified as 4. Aligning to a 32 Byte boundary means that the lower 5 bits of an address are discarded. Similarly, a 256 Byte region size is specified as 7 since the 256 Byte boundaries are determined by ignoring the low order 8 bits. Mathematically, the region size is encoded as: $size_{encoded} = \lceil \log_2(size_{region}) \rceil - 1$.

Conveniently, the linker has a command which computes $\lceil \log_2 \rceil$ so global symbols can be created for the MPU region size which is encoded in such a way that the symbol value may be directly used to configure the MPU. This usage is, admittedly, obscure, but has the advantage of keeping all the MPU region specification in a single place, namely the linker script.

```

<<mpu region symbols>>=
__priv_text_start__ = ORIGIN(priv_text) ;
__priv_text_size__ = LOG2CEIL(LENGTH(priv_text)) - 1 ;

__priv_rodata_start__ = ORIGIN(priv_rodata) ;
__priv_rodata_size__ = LOG2CEIL(LENGTH(priv_rodata)) - 1 ;

__priv_rwdata_start__ = ORIGIN(priv_rwdata) ;
__priv_rwdata_size__ = LOG2CEIL(LENGTH(priv_rwdata)) - 1 ;

__unpriv_text_start__ = ORIGIN(unpriv_text) ;
__unpriv_text_size__ = LOG2CEIL(LENGTH(unpriv_text)) - 1 ;

__unpriv_rodata_start__ = ORIGIN(unpriv_rodata) ;
__unpriv_rodata_size__ = LOG2CEIL(LENGTH(unpriv_rodata)) - 1 ;

__unpriv_rwdata_start__ = ORIGIN(unpriv_rwdata) ;
__unpriv_rwdata_size__ = LOG2CEIL(LENGTH(unpriv_rwdata)) - 1 ;

```

To refer to the linker symbols in code, a set of external references are required. The data type given to these external declarations is of little consequence. When used in code to initialize the MPU, the symbol address is cast to the appropriate bit pattern.

```

<<mpu init: declarations>>=
const RegionSize = v7m.MemoryProtectionUnit.RegionSize;

const priv_text_start =
    @extern(*const u32, .{ .name = "__priv_text_start__", .linkage = .strong });
const priv_text_size =
    @extern(*const RegionSize, .{ .name = "__priv_text_size__", .linkage = .strong });

const priv_rodata_start =
    @extern(*const u32, .{ .name = "__priv_rodata_start__", .linkage = .strong });
const priv_rodata_size =
    @extern(*const RegionSize, .{ .name = "__priv_rodata_size__", .linkage = .strong });

const priv_rwdata_start =
    @extern(*const u32, .{ .name = "__priv_rwdata_start__", .linkage = .strong });
const priv_rwdata_size =
    @extern(*const RegionSize, .{ .name = "__priv_rwdata_size__", .linkage = .strong });

const unpriv_text_start =
    @extern(*const u32, .{ .name = "__unpriv_text_start__", .linkage = .strong });
const unpriv_text_size =
    @extern(*const RegionSize, .{ .name = "__unpriv_text_size__", .linkage = .strong });

const unpriv_rodata_start =
    @extern(*const u32, .{ .name = "__unpriv_rodata_start__", .linkage = .strong });
const unpriv_rodata_size =
    @extern(*const RegionSize, .{ .name = "__unpriv_rodata_size__", .linkage = .strong });

const unpriv_rwdata_start =
    @extern(*const u32, .{ .name = "__unpriv_rwdata_start__", .linkage = .strong });
const unpriv_rwdata_size =
    @extern(*const RegionSize, .{ .name = "__unpriv_rwdata_size__", .linkage = .strong });

const peripheral_device_start: *const u32 = @ptrToInt(0x4000_0000);
const peripheral_device_size: RegionSize = .size_512mb;

```

Configuring the MPU

The following table shows the memory attributes used for the seven defined memory regions. The values for TEX, Shareable, Cacheable, and Bufferable are taken from [\[defguide\]](#), Table 11.10, p364.

Region	TEX	Shareable	Cacheable	Bufferable
priv text	0	0	1	0
priv rodata	0	0	1	0
priv wrdata	0	1	1	0
peripherals	0	1	0	1
unpriv text	0	0	1	0
unpriv rodata	0	0	1	0
unpriv wrdata	0	1	1	0

Table 4. MPU Memory Attributes

The values for the MPU registers that implement the memory protection scheme can now be specified. The following data structure encodes the memory attributes into the register values using the CMSIS MPU functions.

```

<<mpu config>>=
const RasrReg = v7m.MemoryProtectionUnit.RasrReg;
const MpuRegionSpec = struct {
    number: u8,
    address: *const u32,
    attrs: RasrReg,
};

// zig fmt: off
const apollo3_mpu_regions = [_]MpuRegionSpec {
    // Privileged text
    .{
        .number = 0,
        .address = priv_text_start,
        .attrs = .{ .enable = 1, .size = @enumFromInt(@intFromPtr(priv_text_size)),
            .srd = 0, .b = 0, .c = 1, .s = 0, .tex = 0, .ap = .pro, .xn = 0 },
    },
    // Privileged R0 Data
    .{
        .number = 1,
        .address = priv_rodata_start,
        .attrs = .{ .enable = 1, .size = @enumFromInt(@intFromPtr(priv_rodata_size)),
            .srd = 0, .b = 0, .c = 1, .s = 0, .tex = 0, .ap = .pro, .xn = 1 },
    },
    // Privileged RW Data
    .{
        .number = 2,
        .address = priv_rwdata_start,
        .attrs = .{ .enable = 1, .size = @enumFromInt(@intFromPtr(priv_rwdata_size)),
            .srd = 0, .b = 0, .c = 1, .s = 1, .tex = 0, .ap = .priv, .xn = 1 },
    },
    // Peripheral Devices
    .{
        .number = 3,
    }
}

```

```

    .address = peripheral_device_start,
    .attrs = {.enable = 1, .size = peripheral_device_size,
              .srđ = 0, .b = 1, .c = 0, .s = 1, .tex = 0, .ap = .priv, .xn = 1 },
},
// Unprivileged text
.{
    .number = 4,
    .address = unpriv_text_start,
    .attrs = {.enable = 1, .size = @enumFromInt(@intFromPtr(unpriv_text_size)),
              .srđ = 0, .b = 0, .c = 1, .s = 0, .tex = 0, .ap = .ro, .xn = 0 },
},
// Unprivileged R0 Data
.{
    .number = 5,
    .address = unpriv_rodःta_start,
    .attrs = {.enable = 1, .size = @enumFromInt(@intFromPtr(unpriv_rodःta_size)),
              .srđ = 0, .b = 0, .c = 1, .s = 0, .tex = 0, .ap = .ro, .xn = 1 },
},
// Unprivileged RW Data
.{
    .number = 6,
    .address = unpriv_rwdata_start,
    .attrs = {.enable = 1, .size = @enumFromInt(@intFromPtr(unpriv_rwdata_size)),
              .srđ = 0, .b = 0, .c = 1, .s = 1, .tex = 0, .ap = .full, .xn = 1 },
},
};

// zig fmt: on

```

Initializing the MPU

```

<<system_apollo3: systemMemProtectInit>>=
fn systemMemProtectInit() void {
    <<mpu init: declarations>>
    <<mpu config>>

    v7m.mpu.disable();
    for (apollo3_mpu_regions) |region| {
        v7m.mpu.rnr.writeDirect(region.number);
        v7m.mpu.rbar.writeDirect(@intFromPtr(region.address));
        v7m.mpu.rasr.write(region.attrs);
    }
    v7m.mpu.enable({.enable = 1, .hf_nmi_ena = 0, .priv_def_ena = 1 });
}

```

- ① Privileged code is allowed to access the default region map. This facilitates access to the system control block without having to define another MPU region.

Summary

This chapter shows how to configure the Cortex-M4 MPU to segregate memory usage between privileged and unprivileged execution. This design uses seven MPU memory regions to separate

privileged access from unprivileged access across code, read-only data, and read-write data. The seventh region is used to restrict peripheral device access to privileged execution. The linker script was rewritten to layout memory on specific boundaries to accommodate the manner in which the MPU operates. Linker script symbols were defined whose values were directly used to configure the MPU registers.

[1] Historically, this is not the case. ENIAC was first programmed external to its memory, which was used only to hold data. Later improvements added the ability to store a program.

The Tyranny of the Pins

In the late 1950's and into the 1960's CE, computer designers were struggling against the shear [number of wires](#) which had to be connected in order to build a computer from the electronic components available at the time.

For some time now, electronic man has known how 'in principle' to extend greatly his visual, tactile, and mental abilities through the digital transmission and processing of all kinds of information. However, all these functions suffer from what has been called 'the tyranny of numbers.' Such systems, because of their complex digital nature, require hundreds, thousands, and sometimes tens of thousands of electron devices.

— Jack Morton, *The Tyranny of Numbers*

This time was before the invention of integrated circuits, which went a long way toward solving these problems. However, computer chips, especially for systems which interact directly with the outside environment, must be electrically wired to that environment. That is usual done by connecting a *pin* from the chip to some external circuitry. There are only so many pins which can be reasonably accommodated on the physical chip carrier. Physical size is always a constraint in an overall system design and genuine electrical and mechanical constraints apply. The external chip pins must be connected to the internal chip peripheral which uses them. The total number of pins required if all a chip's peripherals were in use usually exceeds the number of pins on the physical package. Chips are designed to be used in a wide range of systems and most chip designs depend upon systems which do not need every peripheral designed into an SOC. Some form of multiplexing of the physical I/O pins to the peripheral I/O signals is a common solution. The mechanisms and interfaces to select the mapping between the physical I/O pin and its internal function vary significantly between chip vendors.

It would be convenient if I/O pins could be treated the same. Unfortunately, the electrical characteristics required don't allow that. The use for a pin must be specified as input or output. There are usually controls over how much current an output pin can drive. Inevitably there are *special* rules about certain pin usage and analog I/O also imposes different conditions.

To compound the problem, when a chip is used on a particular board, the I/O connections are usually given names which indicate their use on the board. So, usually there is a name mapping which must be considered. Tracing through the variety of names a pin might have in different contexts is an annoyingly necessary undertaking.

Much of the details of this part of the book is necessarily specific to the Apollo 3 and the manner in which it controls I/O functionality. Consider the built-in LED on the Artemis processor board as an example. The following figure is from the processor board schematic diagram.

Status LED

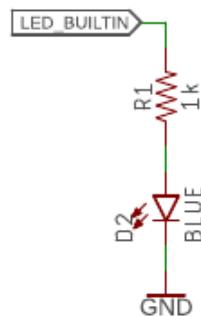


Figure 11. Schematic Diagram for Micromod Status LED

The Apollo 3 has 50 I/O pins. Each I/O pin has up to 7 different selectable functions. In addition, when the Apollo 3 is packaged as the SparkFun Artemis module and used as a MicroMod processor, only certain pins are available and they are mapped to MicroMod naming conventions.

We assume the symbology of the diagram is understood. The information of interest shows that a pin named, LED_BUILTIN, is connected to an LED which is mounted on the processor board. To control the LED^{footnote::[From the schematic it appears that setting the LED_BUILTIN pin to 1 causes a blue LED to illuminate.]}, the Apollo 3 SOC pin that is connected to LED_BUILTIN must be found.

The following figure shows the Artemis module.

Artemis

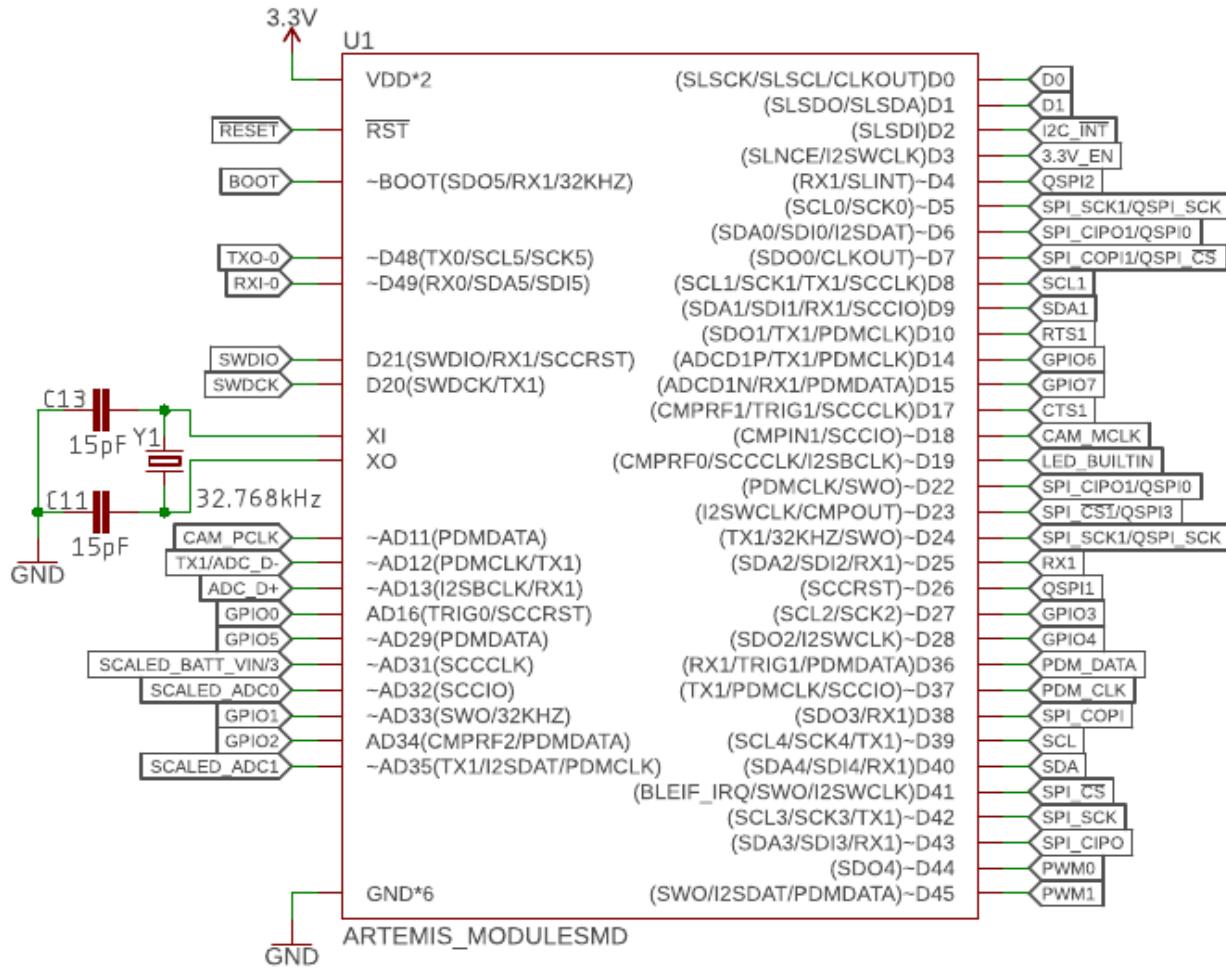


Figure 12. Schematic Diagram for Micromod Artemis Module

The annotations on the inside of the *Artemis module box* show the pin labeling from the point of view of the Apollo 3 processor. The tags outside the module box show the labeling from the point of view of the processor board schematic. Careful examination shows on the right hand side of the module box that the 16th label from the top is called, **LED_BUILTIN**, and connects to the **D19(CMPPRF0/SCCCLK/I2SBCLK)**. D19 is the pin designation. CMPPRF0/SCCCLK/I2SBCLK are abbreviations for some of the internal functions for which D19 may be used (usually each pin has 7 possible functions). So, to control the on-board LED, pin D19 must be configured as a general purpose output.

This chapter shows a set of background request functions and their corresponding foreground proxy functions which form the interface for controlling processor pins for General Purpose Input/Output (GPIO) functionality. Note that no attempt is made to provide a means to configure all possible functions of the GPIO pins. The Apollo 3 GPIO pins have many different and specialized configurations. Many pins have specialized uses. The complexity of the general case is somewhat dizzying. Rather, configuring and using the processor pins as simple I/O pins is sufficient. Later, when other peripheral devices are included, those request functions will handle the required configuration of I/O pins specific to the peripheral function. Some generic pin functions are provided which are useful for both this general purpose I/O pin usage and pin usage by other peripherals.

Pin Function definitions

One of the complexities of the Apollo 3 pin function selection scheme is that each pin has a distinct mapping of a 3-bit number to its set of uses. If the pin is used as an external I/O connection, then its function control is encoded as 3. This applies to all the pins. But for any internal peripheral use, the encoding of the function is not the same across all pins and a detailed mapping is required. For example, pin 19, can be used as the receive pin for UART #1 if its function selector is set to 6. Setting the function selector for pin 18 to 6 connects the pin to the transmit pin for UART #2. The point here is that you must have the mapping table from in the Apollo 3 data sheet to determine the function selector value that allows the pin to be used with a particular peripheral. The following figure, taken from the Apollo 3 data sheet, shows the allowed pad configurations. As you see, the mapping is neither simple nor orthogonal.

Pad	PADnFNCSEL								CSP PKG
	0	1	2	3	4	5	6	7	
0	SLSC1	SLSC2	CLKOUT	GPIO00		MSPI4		NCE0	X
1	SLSDAWIR3	SLMOSI	UART0TX	GPIO01		MSPI5		NCE1	X
2	UART1RX	SLMISO	UART0RX	GPIO02		MSPI6		NCE2	X
3	UA0RTS	SLnCE	NCE3	GPIO03		MSPI7	TRIG1	I2SWCLK	X
4	UA0CTS	SLINT	NCE4	GPIO04		UART1RX	CT17	MSPI2	X
5	M0SCL	M0SCK	UA0RTS	GPIO05				CT8	X
6	M0SDAWIR3	M0MISO	UA0CTS	GPIO06		CT10		I2SDAT	X
7	NCE7	M0MOSI	CLKOUT	GPIO07	TRIG0	UART0TX		CT19	X
8	M1SCL	M1SCK	NCE8	GPIO08	SCCCLK		UART1TX		X
9	M1SDAWIR3	M1MISO	NCE9	GPIO09	SCCIO		UART1RX		X
10	UART1TX	M1MOSI	NCE10	GPIO10	PDMCLK	UA1RTS			X
11	ADCSE2	NCE11	CT31	GPIO11	SLINT	UA1CTS	UART0RX	PDMDATA	X
12	ADCD0NSE9	NCE12	CT0	GPIO12		PDMCLK	UA0CTS	UART1TX	X
13	ADCD0PSE8	NCE13	CT2	GPIO13	I2SBCLK		UA0RTS	UART1RX	X
14	ADCD1P	NCE14	UART1TX	GPIO14	PDMCLK		SWDCK	32KHzXT	X
15	ADCD1N	NCE15	UART1RX	GPIO15	PDMDATA		SWDIO	SWO	X
16	ADCSE0	NCE16	TRIG0	GPIO16	SCCRST	CMPIN0	UART0TX	UA1RTS	X
17	CMPRF1	NCE17	TRIG1	GPIO17	SCCCLK		UART0RX	UA1CTS	X
18	CMPIN1	NCE18	CT4	GPIO18	UA0RTS		UART1TX	SCCIO	X
19	CMPRF0	NCE19	CT6	GPIO19	SCCCLK		UART1RX	I2SBCLK	X
20	SWDCK	NCE20		GPIO20	UART0TX	UART1TX	I2SBCLK	UA1RTS	X
21	SWDIO	NCE21		GPIO21	UART0RX	UART1RX	SCCRST	UA1CTS	X
22	UART0TX	NCE22	CT12	GPIO22	PDMCLK		MSPI0	SWO	X
23	UART0RX	NCE23	CT14	GPIO23	I2SWCLK	CMPOUT	MSPI3		X
24	UART1TX	NCE24	MSPI8	GPIO24	UA0CTS	CT21	32KHzXT	SWO	X
25	UART1RX	NCE25	CT1	GPIO25	M2SDAWIR3	M2MISO			X
26		NCE26	CT3	GPIO26	SCCRST	MSPI1	UART0TX	UA1CTS	X
27	UART0RX	NCE27	CT5	GPIO27	M2SCL	M2SCK			X
28	I2SWCLK	NCE28	CT7	GPIO28		M2MOSI	UART0TX		X
29	ADCSE1	NCE29	CT9	GPIO29	UA0CTS	UA1CTS	UART0RX	PDMDATA	X
30		NCE30	CT11	GPIO30	UART0TX	UA1RTS	BLEIF_SCK	I2SDAT	
31	ADCSE3	NCE31	CT13	GPIO31	UART0RX	SCCCLK	BLEIF_MISO	UA1RTS	
32	ADCSE4	NCE32	CT15	GPIO32	SCCIO		BLEIF_MOSI	UA1CTS	
33	ADCSE5	NCE33	32KHzXT	GPIO33	BLEIF_CSN	UA0CTS	CT23	SWO	
34	ADCSE6	NCE34	UA1RTS	GPIO34	CMPRF2	UA0RTS	UART0RX	PDMDATA	
35	ADCSE7	NCE35	UART1TX	GPIO35	I2SDAT	CT27	UA0RTS	BLEIF_STATUS	
36	TRIG1	NCE36	UART1RX	GPIO36	32KHzXT	UA1CTS	UA0CTS	PDMDATA	
37	TRIG2	NCE37	UA0RTS	GPIO37	SCCIO	UART1TX	PDMCLK	CT29	
38	TRIG3	NCE38	UA0CTS	GPIO38		M3MOSI	UART1RX		
39	UART0TX	UART1TX	CT25	GPIO39	M4SCL	M4SCK			X
40	UART0RX	UART1RX	TRIG0	GPIO40	M4SDAWIR3	M4MISO			X
41	NCE41	BLEIF_IRQ	SWO	GPIO41	I2SWCLK	UA1RTS	UART0TX	UA0RTS	X
42	UART1TX	NCE42	CT16	GPIO42	M3SCL	M3SCK			
43	UART1RX	NCE43	CT18	GPIO43	M3SDAWIR3	M3MISO			
44	UA1RTS	NCE44	CT20	GPIO44		M4MOSI	UART0TX		X
45	UA1CTS	NCE45	CT22	GPIO45	I2SDAT	PDMDATA	UART0RX	SWO	
46	I2SBCLK	NCE46	CT24	GPIO46	SCCRST	PDMCLK	UART1TX	SWO	
47	32KHzXT	NCE47	CT26	GPIO47		M5MOSI	UART1RX		X
48	UART0TX	NCE48	CT28	GPIO48	M5SCL	M5SCK			X
49	UART0RX	NCE49	CT30	GPIO49	M5SDAWIR3	M5MISO			X

Figure 13. GPIO Pad Function Selections

The mapping shown in the previous diagram relates the I/O Pad number and its function selection code to a particular usage of the pin associated to the pad. This is not quite the mapping we need. We need a mapping from Pad number and pad function to the function selection code. That mapping is provided as part of the register descriptions of the GPIO peripheral.

Sharing Pins

There is only one GPIO peripheral in the Apollo 3 SOC, yet many peripherals need access to particular pins. For example, I/O pins are used by any peripheral device wired to the environment and not just GPIO pin usage. This is our first encounter with multiple peripheral instances that may be used across different foreground proxy functions. We set up a minimal pin allocation scheme to detect conflicts between differing uses of the I/O Pins.

```
<<device access>>=
/// The `DeviceAllocator` type function generates a data type suitable to use
/// for controlling the allocation of multiple instances of peripheral
/// devices. The maximum number of instances is given by the `instance_count`
/// argument. Any instances that are pre-reserved are given by the
/// `reserved_instances` argument. Instance in the `reserved_instances`
/// slice will already have been marked as allocated when the function returns.
fn DeviceAllocator(
    comptime instance_count: usize,
    comptime reserved_instances: []const u8,
) type {
    return struct {
        var allocation = ablk: {
            var result = std.StaticBitSet(instance_count).initEmpty(); ①
            for (reserved_instances) |reserved| {
                result.set(reserved);
            }
            break :ablk result;
        };

        /// The `allocate` function marks the device `instance` as allocated.
        /// The function returns an error if the instance is already allocated.
        pub fn allocate(
            instance: u8,
        ) !void {
            if (allocation.isSet(instance)) return error.Allocated;
            allocation.set(instance);
        }

        /// The `allocateAny` function finds an unallocated instance of the
        /// device, marks it as allocated and returns the instance number.
        /// If no unallocated device can be found, an error is returned.
        pub fn allocateAny() !u8 {
            if (findFirstUnset()) |device| {
                allocation.set(device);
                return device;
            } else {
                return error.OutOfDevices;
            }
        }
    };
}
```

```

        }
    }

    /// The `free` function marks the device `instance`
    /// as free so that it may be reallocated in the future.
    pub fn free(
        instance: u8,
    ) void {
        allocation.unset(instance);
    }

    /// The `isAllocated` function returns `true` if `instance` is allocated and
    /// `false` otherwise.
    pub fn isAllocated(
        instance: u8,
    ) bool {
        return allocation.isSet(instance);
    }

    /// The `StaticBitSet` does not have a function to find the first unset bit.
    /// So, we supply our own implementation.
    fn findFirstUnset() ?usize {
        const iter = allocation.iterator(.{ .kind = .unset });
        return iter.next();
    }
};

}

```

- ① The standard library `StaticBitSet` is an appropriate choice of data structure to track device instance allocations.

Allocating GPIO Pins

Which pins are in use is information used by many foreground proxy functions. Foreground proxies for the various devices are expected to allocate and free pins as they are needed and to check the allocation they want to use.

```

<<device access>>=
const gpio_allocator = DeviceAllocator(svc_services.gpio.instances, &.{ 20, 21 }); ①

```

- ① Pads 20 and 21 are connected the software debug clock and data, respectively at reset time. They are *not* to be reallocated except with considerable deliberation.

GPIO Logical Device

Following the pattern established for the Watchdog Timer, the following diagram shows the usage of peripherals for the GPIO logical device.

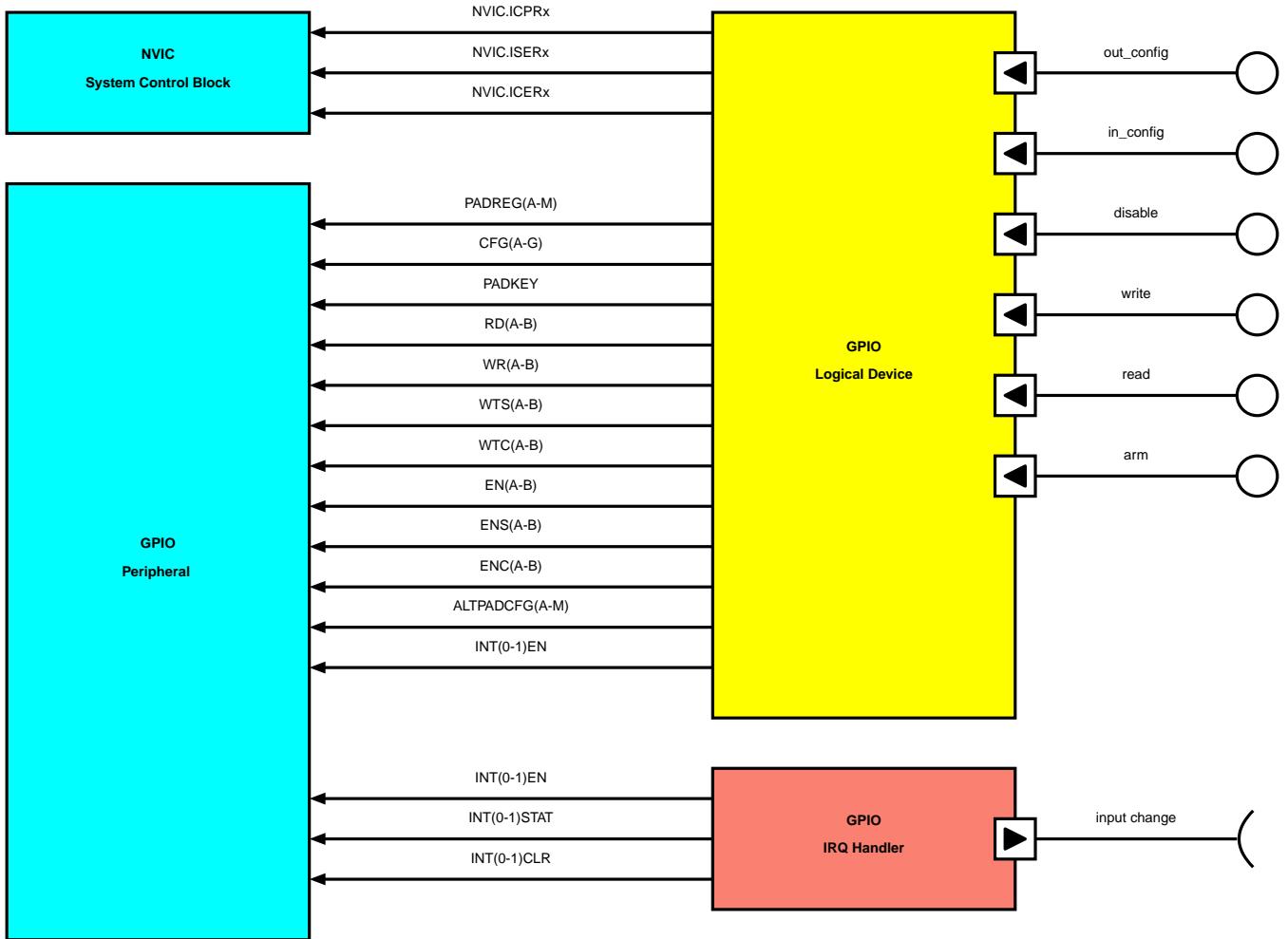


Figure 14. GPIO Logical Device Components

Note that the remainder of this section discusses the logical device for individual pins used for general purposes. Pins used for dedicated peripherals are described with each peripheral as they come up.

There are a number of GPIO register operations which can be factored into common code for use here and for the configuration needs of peripheral devices. This circumstance is true of several peripheral devices where there are multiple instances of the device and some management over the use of the devices instances is needed to coordinate the interactions of the foreground proxy functions among themselves.

Following the diagram in the previous figure, we create data structures to manage the interactions between the watchdog timer logical device and its component peripherals. We start with the common interrupt control functions provided by the NVIC.

```
<<gpio proxies>>
const irq_control: IrcControl = .{.irq_number = apollo3.Gpio.irq_number};
```

GPIO Peripheral Operations

This section shows a set of operations for a single GPIO pin. Since the state of a pin can be represented by a single bit, the operations usually result in familiar bit mask expressions. The

Apollo 3 GPIO's also supply registers which can set or clear a pin while not affecting other pins. These registers save a read/modify/write transaction on the GPIO register. Where applicable, these functions take advantage of such specialized registers.

```
<<gpio proxies>>=
pub const pin_control = struct {
    const IoPin = u8;

    <<pin control operations>>
};
```

```
<<pin control operations>>=
fn locate(
    pin: IoPin,
) struct {u8, u5} {
    const index = pin / @bitSizeOf(mmio.WordRegister);
    const offset: u5 = @truncate(pin % @bitSizeOf(mmio.WordRegister));
    return .{ index, offset };
}
```

```
<<pin control operations>>=
/// The `set` function set the output value of the given pin to 1.
fn set(
    pin: IoPin,
) void {
    const index, const offset = locate(pin);
    apollo3.gpio.wts[index].writeBitField(offset);
}
```

```
<<pin control operations>>=
/// The `clear` function set the output value of the given pin to 0.
fn clear(
    pin: IoPin,
) void {
    const index, const offset = locate(pin);
    apollo3.gpio.wtc[index].writeBitField(offset);
}
```

```
<<pin control operations>>=
/// The `toggle` function set the output value of the given pin to the opposite
/// its current value.
fn toggle(
    pin: IoPin,
) void {
    const index, const offset = locate(pin);
    apollo3.gpio.wt[index].toggleBitField(offset);
}
```

```
<<pin control operations>>
/// The `setTriState` function sets the tri-state of the given pin.
fn setTriState(
    pin: IoPin,
) void {
    const index, const offset = locate(pin);
    apollo3.gpio.ens[index].writeBitField(offset);
}
```

```
<<pin control operations>>
/// The `clearTriState` function clears the tri-state of the given pin.
fn clearTriState(
    pin: IoPin,
) void {
    const index, const offset = locate(pin);
    apollo3.gpio.ens[index].writeBitField(offset);
}
```

```
<<pin control operations>>
/// The `toggleTriState` function sets the tri-state of the given pin to
/// the opposite of its current value.
fn toggleTriState(
    pin: IoPin,
) void {
    const index, const offset = locate(pin);
    apollo3.gpio.ens[index].toggleBitField(offset);
}
```

```
<<pin control operations>>
/// The `read` function returns the current value of `pin`.
fn read(
    pin: IoPin,
) u1 {
    const index, const offset = locate(pin);
    return apollo3.gpio.rds[index].readBitField(offset);
}
```

```
<<pin control operations>>
/// The `disableIrq` function disables the interrupt associated with `pin`.
fn disableIrq(
    pin: IoPin,
) void {
    const index, const offset = locate(pin);
    apollo3.gpio.int_ctrl[index].int_en.clearBitField(offset);
    apollo3.gpio.int_ctrl[index].int_clr.writeBitField(offset);
}
```

```
<<pin control operations>>
```

```

/// The `enableIrq` function enables the interrupt associated with `pin`.
fn enableIrq(
    pin: IoPin,
) void {
    const index, const offset = locate(pin);
    apollo3.gpio.int_ctrl[index].int_clr.writeBitField(offset);
    apollo3.gpio.int_ctrl[index].int_en.setBitField(offset);
}

```

```

<<pin control operations>>=
/// The `configure` function sets the pad and pin configuration information
/// associated with `pin`.
fn configure(
    pin: IoPin,
    pad_config: PadConfigType,
    pin_config: GpioConfigType,
    alt_pad_config: AltPadConfigType,
) void {
    // Unlock configuration registers.
    apollo3.gpio.pad_key.write(.pad_key);

    var index = pin / PadConfigType.packing_factor;
    var offset: u5 = @truncate(pin % PadConfigType.packing_factor);
    apollo3.gpio.pad_regs[index].updateIndexedElement(offset, pad_config);

    index = pin / GpioConfigType.packing_factor;
    offset = @truncate(pin % GpioConfigType.packing_factor);
    apollo3.gpio.cfg_regs[index].updateIndexedElement(offset, pin_config);

    index = pin / AltPadConfigType.packing_factor;
    offset = @truncate(pin % AltPadConfigType.packing_factor);
    apollo3.gpio.alt_pad_cfg[index].updateIndexedElement(offset, alt_pad_config);

    // Re-lock the configuration register.
    apollo3.gpio.pad_key.write(.{});
}

```

```

<<pin control operations>>=
/// The `disable` function returns `pin` to a quiescent state.
fn disable(
    pin: IoPin,
) void {
    disableIrq(pin);
    configure(
        pin,
        .{
            .pull = 0,
            .inp_en = 0,
            .strng = .low,
            // Every pad can have its function selection set to .gpio.
            .fnc_sel = apollo3.Gpio.padFunctionSelector(pin, .gpio) catch unreachable,
        }
    );
}

```

```

        .r_sel = .@"1.5K",
    }, // pad configuration
    .{
        .in_cfg = .rd_zero,
        .out_cfg = .dis,
        .int_dir = .int_lh,
    }, // pin configuration
    .{
        .ds = 0,
        .sr = 0,
    }, // alt pad configuration
);
}

```

GPIO Services

GPIO pin to register field mapping

Pin configuration for the Apollo 3 is controlled primarily with three registers, PADREG, CFG, and ALTPADCFG. Note that there is a difference between configuring the *pad* of the GPIO and configuring the *pin*. The hardware description groups these registers as arrays since they are located sequentially in the memory. There are 13 PADREG and ALTPADCFG instances and 7 CFG instances. Each PADREG register is further divided into 8-bit groups which control the settings for one pin. The 8-bit groups are further divided into 4 bit fields which apply to all pins and a fifth field is used to control functions which are particular to only certain pins. The ALTPADCFG registers are also divided into 8-bit groups with additional bit fields in the group, but the controls are the same for all pins. The CFG registers are divided into 4-bit groups with additional control bit fields in each group. Some of the control bits in the CFG groups are overloaded in their control meaning. Yes, it is a complicated interfacing scheme.

Notice that there are separate configuration requests for output and input pins. There is some overlap in configuration settings, but generally you know whether a pin is to be used for input or output from the context of the application. Separating the different configurations makes for a simpler interface.

Following our pattern from the last chapter, we set up the device services and proxies for GPIO control. First, we start with background service requests.

```

<<system_services: device service definitions>>=
/// GPIO services
pub const gpio = struct {
    <<gpio services>>
};

```

```

<<gpio services>>=
pub const instances: DevSvcInstance = apollo3.Gpio.pin_count; ①

```

- ① We consider each GPIO pin as a distinct instance of the logical device formed by the single GPIO controller hardware block.

```

<<gpio services>>=

```

```

pub const GpioOperations = DefineDeviceOperations(
    .gpio,
    &{ "output_config", "input_config", "disable", "write", "read", "arm" },
);

```

Next, we set up the necessary structure for the foreground proxy functions that carry out GPIO requests.

```

<<device proxies>>=

/// GPIO service proxies
const gpio = struct {
    <<gpio proxies>>
};

```

As usual, we need some names defined in other namespaces.

```

<<gpio proxies>>=
const PinOutputType = svc_services.gpio.PinOutputType;
const PullupResistance = svc_services.gpio.PullupResistance;
const DriveStrength = svc_services.gpio.DriveStrength;

```

```

<<gpio proxies>>=
const PadConfigType = apollo3.Gpio.PadConfigType;
const GpioConfigType = apollo3.Gpio.GpioConfigType;
const AltPadConfigType = apollo3.Gpio.AltPadConfigType;
const PullupSelector = apollo3.Gpio.PullupSelector;

```

The following sections present the background requests for the GPIO device. The familiar pattern of presenting the request function which executes unprivileged in the background followed by the foreground proxy function which manipulates the peripherals is followed.

Configure Output Pin Service

```

<<gpio services>>=
/// The `outputConfig` function configures the GPIO given by, `pin`, to be an
/// output pin. The manner in the output is driven is given by `type`. The amount
/// of current which the pin can dirve is given by `drive_strength`. An pull-up
/// resistor value for the GPIO pad is specified by `pullup_resistor`.
pub fn outputConfig(
    /// The GPIO pin number of the pin to be configured as an output.
    pin: DevSvcInstance,
    /// An indicator of the electrical characteristics of the pin output.
    output_type: PinOutputType,
    /// The amount of current the pin is capable of driving.
    drive_strength: DriveStrength,
    /// A value for any pull up resistor to be associated with the pin pad. _N.B.-
    /// for pin #20, the resistor is actually a pull-down resistor. For all other
    /// pins, the pad resistor serves as a pull-up resistor.
)

```

```

        pullup_resistance: PullupResistance,
) SvcError!void {
    std.debug.assert(pin < instances);

    const input_params = OutputConfigInputParam.marshal(.{
        .type = output_type,
        .drive_strength = drive_strength,
        .pullup_resistance = pullup_resistance,
    });
    const request = GpioOperations.makeRequest(.output_config, pin);
    return devRealmSvcCall(request, @ptrCast(&input_params), null, null);
}

```

Some encodings of the various arguments for the parameters to the function are required.

```

<<gpio services>>=
pub const PinOutputType = enum {
    push_pull,
    open_drain,
    tri_state,
};

```

Internal pull up resistors on the GPIO pad are only available on certain pads. The following enumeration unifies the naming which is then checked and split out again when the register fields are computed.

```

<<gpio services>>=
pub const PullupResistance = enum {
    none,
    weak,
    @"1.5K",
    @"6K",
    @"12K",
    @"24K",
};

```

```

<<gpio services>>=
pub const DriveStrength = enum {
    @"2mA",
    @"4mA",
    @"8mA",
    @"12mA",
};

```

Output pin configuration requires passing additional input parameters to the foreground proxy.

```

<<gpio services>>=
pub const OutputConfigInputParam = DefineSvcRequestParam(struct {
    type: PinOutputType,
    drive_strength: DriveStrength,
});

```

```
    pullup_resistance: PullupResistance,  
});
```

Configure Output Pin Proxy

```
<<gpio proxies>>=  
/// The `outputConfig` function receives control of the GPIO output  
/// configuration request via the SVC exception and the GPIO device class  
/// dispatch function.  
pub fn outputConfig(  
    req: DevSvcRequest,  
    input: ?*const anyopaque,  
    _: ?*anyopaque,  
    _: ?*anyopaque,  
) SvcResult {  
    const OutputConfigInputParam = svc_services.gpio.OutputConfigInputParam;  
  
    const input_params, const result = OutputConfigInputParam.receive(input.?);  
    if (result != .success) return result;  
  
    const pin = req.instance;  
    gpio_allocator.allocate(pin) catch return .operation_failed;  
  
    // The strategy is to compute the value of the three configuration registers  
    // based on the input parameter arguments and then perform the update  
    // the registers below.  
  
    var pad_config: PadConfigType = .{};  
    pad_config.pull, pad_config.r_sel =  
        configPullupResistance(pin, input_params.pullup_resistance) catch  
            return .invalid_param;  
    pad_config.inp_en = 0;  
    pad_config.fnc_sel = apollo3.Gpio.padFunctionSelector(pin, .gpio) catch unreachable;  
  
    var alt_pad_config: AltPadConfigType = .{};  
    switch (input_params.drive_strength) {  
        .@"2mA" => {  
            pad_config.strng = .low;  
            alt_pad_config.ds = 0;  
        },  
        .@"4mA" => {  
            pad_config.strng = .high;  
            alt_pad_config.ds = 0;  
        },  
        .@"8mA" => {  
            pad_config.strng = .low;  
            alt_pad_config.ds = 1;  
        },  
        .@"12mA" => {  
            pad_config.strng = .high;  
            alt_pad_config.ds = 1;  
        },
```

```

        },
    }

    var pin_config: GpioConfigType = @bitCast(@as(u4, 0));
    pin_config.out_cfg = switch (input_params.type) {
        .push_pull => .pushpull,
        .open_drain => .od,
        .tri_state => .ts,
    };

    // These two fields work together and disable interrupts on an output pin.
    pin_config.in_cfg = .read_en;
    pin_config.int_dir = .int_lh;

    pin_control.configure(pin, pad_config, pin_config, alt_pad_config);

    return .success;
}

```

The Apollo 3 GPIO configuration is complicated and many pins have special rules and uses. The first one encountered here is with respect to pull-up resistors on the GPIO pads. All pads have the ability to enable a pull-up resistor. For some pads, the pull-up is either disabled or, if enabled, is only a *weak* pull up. For pads which are can be used to drive I²C or SPI output, there is a range of pull-up resistor values which can be set. A check is made whether the pull-up resistor value given in the input arguments is suitable for a given pin. These special rules about pull-up resistance are encoded in the function, `apollo3.Gpio.PadPullups.hasAlternatePullup` A function to handle pad configuration for the pull-up resistors is factored out.

```

<<gpio proxies>>=
fn configPullupResistance(
    pin: DevSvcInstance,
    resistance: PullupResistance,
) !struct {u1, PullupSelector} {
    switch (resistance) {
        .none => return {.0, .@"1.5K" },
        .weak => {
            if (apollo3.Gpio.PadPullups.hasAlternatePullup(pin))
                return error.invalid_resistance;
            return {.1, .@"1.5K" };
        },
        .@"1.5K" => {
            if (!apollo3.Gpio.PadPullups.hasAlternatePullup(pin))
                return error.invalid_resistance;
            return {.1, .@"1.5K" };
        },
        .@"6K" => {
            if (!apollo3.Gpio.PadPullups.hasAlternatePullup(pin))
                return error.invalid_resistance;
            return {.1, .@"6K" };
        },
        .@"12K" => {
            if (!apollo3.Gpio.PadPullups.hasAlternatePullup(pin))

```

```

        return error.invalid_resistance;
    return .{ 1, @"12K" };
},
@"24K" => {
    if (!apollo3.Gpio.PadPullups.hasAlternatePullup(pin))
        return error.invalid_resistance;
    return .{ 1, @"24K" };
},
}
}
}

```

Configure Input Pin Service

```

<<gpio services>>=
/// The `inputConfig` function configures the GPIO `pin` to be an input pin.
pub fn inputConfig(
    /// The GPIO pin number of the pin to be configured as an input.
    pin: DevSvcInstance,
    /// The transitions of the input pin which are to generate an interrupt.
    intr_config: IntrConfig,
    /// A boolean value indicating whether the pin interrupt is to be disarmed when
    /// it occurs.
    disarm_on_active: bool,
    /// A value for any pull up resistor to be associated with the pin pad. _N.B._
    /// for pin #20, the resistor is actually a pull-down resistor. For all other
    /// pins, the pad resistor serves as a pull-up resistor.
    pullup_resistance: PullupResistance,
    /// A pointer to the background proxy function which is to be executed when the
    /// background notification associated with the pin interrupt is dispatched.
    /// If the `intr_config` argument is specified as `.none` then a `null` value
    /// must be given for `notify_proxy`.
    notify_proxy: ?DevSvcNotifyProxy,
    /// A caller supplied data value which is passed along in the background
    /// notification.
    closure: usize,
) SvcError!void {
    std.debug.assert(pin < instances);

    if (notify_proxy == null and intr_config != .none) return SvcError.invalid_param;

    const input_params = InputConfigInputParam.marshal(.{
        .intr_config = intr_config,
        .disarm_on_active = disarm_on_active,
        .pullup_resistance = pullup_resistance,
        .notify_proxy = notify_proxy,
        .closure = closure,
    });
    const request = GpioOperations.makeRequest(.input_config, pin);
    return devRealmSvcCall(request, @ptrCast(&input_params), null, null);
}

```

The following type gives the choices of when interrupts are generated for input pins.

```
<<gpio services>>
pub const IntrConfig = enum {
    none,
    both,
    low_high, // on the transition from 0 to 1
    high_low, // on the transition from 1 to 0
};
```

```
<<gpio services>>
pub const InputConfigInputParam = DefineSvcRequestParam(struct {
    intr_config: IntrConfig,
    disarm_on_active: bool,
    pullup_resistance: PullupResistance,
    notify_proxy: ?DevSvcNotifyProxy, // If not configured to interrupt, no proxy is
needed.
    closure: usize,
});
```

Configure Input Pin Proxy

When a GPIO pin is used as input, it can be used to generate an interrupt. In that case, a device notification is sent to the background.

Interrupts can be generated on either transition, both transitions or no interrupt need be generated if the pin is to be polled only. Note if no interrupt is requested, then the proxy function argument must be `null`.

We need storage for the information about input pin interrupts and notifications. Simple worst case static allocation is used, despite the fact that it would be a rare system indeed where all of its pins are configured as inputs.

```
<<gpio proxies>>
const GpioInputStatus = struct {
    disarm_on_active: bool,
    proxy: DevSvcNotifyProxy,
    closure: usize,
};

var notifications: [svc_services.gpio.instances]GpioInputStatus = undefined;
```

```
<<gpio proxies>>
/// The `inputConfig` function receives control of the GPIO input
/// configuration request via the SVC exception and the GPIO device class
/// dispatch function.
pub fn inputConfig(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    _: ?*anyopaque,
```

```

_: ?*anyopaque,
) SvcResult {
    const InputConfigInputParam = svc_services.gpio.InputConfigInputParam;

    const input_params, const result = InputConfigInputParam.receive(input.?);
    if (result != .success) {
        return result;
    }

    const pin = req.instance;
    if ((input_params.intr_config == .none and input_params.notify_proxy != null) or
        input_params.notify_proxy == null) return .invalid_param;

    gpio_allocator.allocate(pin) catch return .invalid_param;

    // Following the same design strategy as for output configuration,
    // compute the values of the three required registers and write the
    // to the chip after unlocking the registers.

    var pad_config: PadConfigType = .{};
    pad_config.pull, pad_config.r_sel =
        configPullupResistance(pin, input_params.pullup_resistance) catch
            return .invalid_param;
    pad_config.inp_en = 1;
    pad_config.strng = .low;
    pad_config.fnc_sel = apollo3.Gpio.padFunctionSelector(pin, .gpio) catch unreachable;

    var pin_config: GpioConfigType = .{};
    pin_config.out_cfg = .dis;
    switch (input_params.intr_config) {
        .none => {
            pin_config.int_dir = .int_lh;
            pin_config.in_cfg = .read_en;
        },
        .both => {
            pin_config.int_dir = .int_hl;
            pin_config.in_cfg = .read_en;
        },
        .low_high => {
            pin_config.int_dir = .int_lh;
            pin_config.in_cfg = .rd_zero;
        },
        .high_low => {
            pin_config.int_dir = .int_hl;
            pin_config.in_cfg = .rd_zero;
        },
    }
}

pin_control.configure(pin, pad_config, pin_config, .{});

if (input_params.intr_config != .none) {
    notifications[pin].disarm_on_active = input_params.disarm_on_active;
    notifications[pin].proxy = input_params.notify_proxy.?;
}

```

```

        notifications[pin].closure = input_params.closure;
        pin_control.enableIrq(pin);
        irq_control.enable();
    }

    return .success;
}

```

Disable GPIO Pin Service

```

<<gpio services>>=
/// The `disable` function disables `pin` and frees `pin` to be reused.
/// Callers should disable any pins for which they have no further use to minimize
/// power consumption.
pub fn disable(
    /// The GPIO pin number of the pin to be disabled.
    pin: DevSvcInstance,
) SvcError!void {
    std.debug.assert(pin < instances);

    const request = GpioOperations.makeRequest(.disable, pin);
    return devRealmSvcCall(request, null, null, null);
}

```

Disable GPIO Pin Proxy

```

<<gpio proxies>>=
pub fn disable(
    req: DevSvcRequest,
    _: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const pin = req.instance;
    pin_control.disable(pin);
    gpio_allocator.free(pin);
    notifications[pin].proxy = undefined;
    notifications[pin].closure = undefined;

    return .success;
}

```

Write Output Pin Service

Since each pin can be represented by a single bit, rather than having a separate request for each output operation, this design uses a single function with a parameter which states the operation to perform.

```
<<gpio services>>=
```

```

const WriteOp = enum {
    pin_set,
    pin_clear,
    pin_toggle,
    tristate_set,
    tristate_clear,
    tristate_toggle,
};

}

```

The set, clear, and toggle operations are conventional and control the output seen at the pin. For those output pins which have been configured as tri-state pins, the ability to control the tri-state setting is also given set, clear, and toggle operations on the tri-state setting.

```

<<gpio services>>=
/// The `write` function performs the operation indicated by, `operation`, on the
/// pin given by, `pin`.
pub fn write(
    /// The GPIO pin number of the pin to be written.
    pin: DevSvcInstance,
    /// The write operation to perform on `pin`.
    write_op: WriteOp,
) SvcError!void {
    std.debug.assert(pin < instances);

    const input_params = WriteInputParam.marshal(.{
        .write_op = write_op,
    });
    const request = GpioOperations.makeRequest(.write, pin);
    return devRealmSvcCall(request, @ptrCast(&input_params), null, null);
}

```

```

<<gpio services>>=
pub const WriteInputParam = DefineSvcRequestParam(struct {
    write_op: WriteOp,
});

```

Write Output Pin Proxy

```

<<gpio proxies>>=
pub fn write(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const WriteInputParam = svc_services.gpio.WriteInputParam;

    const input_params, const result = WriteInputParam.receive(input.?);
    if (result != .success) {
        return result;
    }
}

```

```

    }

    const pin = req.instance;
    switch (input_params.write_op) {
        .pin_set => pin_control.set(pin),
        .pin_clear => pin_control.clear(pin),
        .pin_toggle => pin_control.toggle(pin),
        .tristate_set => pin_control.setTriState(pin),
        .tristate_clear => pin_control.clearTriState(pin),
        .tristate_toggle => pin_control.toggleTriState(pin),
    }

    return .success;
}

```

Read Input Pin Service

```

<<gpio services>>=
/// The `read` function returns the current input value from `pin`.
pub fn read(
    /// The GPIO pin number of the pin to be read.
    pin: DevSvcInstance,
) SvcError!u1 {
    std.debug.assert(pin < instances);

    var output_params: ReadOutputParam = .initial;

    const request = GpioOperations.makeRequest(.read, pin);
    try devRealmSvcCall(request, null, @ptrCast(&output_params), null);

    return output_params.unmarshal().pin_value;
}

```

```

<<gpio services>>=
pub const ReadOutputParam = DefineSvcRequestParam(struct {
    pin_value: u1,
});

```

Read Input Pin Proxy

```

<<gpio proxies>>=
pub fn read(
    req: DevSvcRequest,
    _: ?*const anyopaque,
    output: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const pin = req.instance;

```

```

const ReadOutputParam = svc_services.gpio.ReadOutputParam;
return ReadOutputParam.send({.pin_value = pin_control.read(pin)}, output.?);
}

```

Arm GPIO Pin Service

Part of the configuration for an input pin is whether it is to be disarmed when it is triggered. This request re-arms the pin when the application has decided it is ready to receive another input from the pin.

```

<<gpio services>>=
/// The `arm` function readies the input pin given by, `pin`, to notify changes in state.
pub fn arm(
    /// The GPIO pin number of the pin to be armed.
    pin: DevSvcInstance,
) SvcError!void {
    std.debug.assert(pin < instances);

    const request = GpioOperations.makeRequest(.arm, pin);
    return devRealmSvcCall(request, null, null, null);
}

```

Arm GPIO Pin Proxy

```

<<gpio proxies>>=
pub fn arm(
    req: DevSvcRequest,
    _: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const pin = req.instance;
    pin_control.enableIrq(pin);

    return .success;
}

```

Dispatching GPIO Requests

In keeping with the device foreground proxy design, a class level handler is used to dispatch background requests to the appropriate proxy that handles the request.

```

<<svc device classes>>=
gpio,

```

```

<<dev svc service request prongs>>=
 gpio => &gpio.devClassHandler,

```

Finally, the GPIO class handler performs the dispatch to the operation functions of requests.

```
<<gpio proxies>>=
/// The `devClassHandler` function dispatches watchdog operations based
/// on the value of the `req` argument. This function is invoked as a
/// class level dispatcher for the GPIO device.
fn devClassHandler(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    output: ?*anyopaque,
    err: ?*anyopaque,
) SvcResult {
    const Operation = svc_services.gpio.GpioOperations.Operation;
    const gpio_operation: Operation = @enumFromInt(req.operation);
    const proxy = switch (gpio_operation) {
        .output_config => &outputConfig,
        .input_config => &inputConfig,
        .disable => &disable,
        .write => &write,
        .read => &read,
        .arm => &arm,
    };
    return proxy(req, input, output, err);
}
```

GPIO IRQ Handler

The notification from the GPIO device for an input pin contains the value of the pin when the interrupt happened.

```
<<dev notification specs>>=
.{ "gpio", u1 },
```

The IRQ handler for the GPIO pins has to find which pins are currently interrupting. At any given time, that is most probably only 1 pin. The interrupt information about the pins is in separate register arrays. With 50 pins and 32 bit registers, it takes two array elements to hold all the information. The layout of the GPIO controller interrupt registers is in the form of a structure of bits, composed into an array of two. No matter, two nested loops are used. The first loop iterates over the controls which are grouped by 32 pins. So, there is at most two times through the outer loop. The inner loop handles at most 32 pins, matching encoding of pins in a 32-bit register.

```
<<gpio proxies>>=
/// The `gpioIrqHandler` function receives control when the gpio interrupt
/// is made active.
export fn gpioIrqHandler() void {
    var pin_offset: u8 = 0; ①

    for (apollo3.gpio.int_ctrl[0...]) |*int_ctrl| {
        var int_pending =
            int_ctrl.int_stat.readDirect() & int_ctrl.int_en.readDirect(); ②
```

```

    while (int_pending != 0) {
        const lead_bit: u5 = @truncate(@as(u6, 31) - @clz(int_pending)); ③

        // Handle processing for one pin.
        int_ctrl.int_clr.writeBitField(lead_bit);
        const pin = pin_offset + lead_bit;
        if (notifications[pin].disarm_on_active) {
            pin_control.disableIrq(pin);
        }

        const result: SvcResult, const notification: *DevNotification = ④
            dev_notify_queue.reserveNotification();
        if (result != .success)
            panic.privPanic(null, "GPIO notification failed");

        notification.* = DevNotification.init(
            .gpio, // operation
            pin, // instance
            notifications[pin].proxy,
            notifications[pin].closure,
            // GPIO specific notification information
            .{ .gpio = pin_control.read(pin) },
        );
        dev_notify_queue.commitNotification();

        int_pending &= ~(@as(u32, 1) << lead_bit); ⑤
    }
    pin_offset += @bitSizeOf(mmio.WordRegister) ;
}
}

```

- ① The first 32 pins are handled by the first iteration and the remaining 18 pins are accounted for in the second iteration of the loop. Note this implies we are dependent upon the unused bits of the second interrupt register to read as zero.
- ② Find those interrupt sources that are both enabled and asserted. Most Apollo 3 interrupt status registers are *not* masked by the enabled interrupts (*cf.* the UART interrupts where there is a masked interrupts status).
- ③ Counting the leading zeros in the status register can be used to determine the pin number causing an interrupt. This saves examining the status register bit by bit in a loop, only to find that most of the pins are not interrupting at this moment. *N.B.* this services the pins in descending order of pin number. Servicing in ascending pin order can be achieved by reversing the bits in the `int_pending` variable using the `@bitReverse` built-in function.
- ④ GPIO pins only send notifications from the IRQ handler, so it is not necessary to raise the execution priority.
- ⑤ Since we managing the loop with integer values, we have to resort to bit twiddling to clear out the interrupt indication of the bit just serviced.

Examples

In this chapter, four examples of using GPIO pins are shown.

Driving an Input Pin

This example demonstrates using a GPIO pin as an input and we will drive the input using an external square wave generator. The intent is to raise the frequency of the generated square wave until the system is overloaded and fails.

The measurements were made with an Analog Discovery2 instrument, driving a square wave into pin 27 (GPIO 3 on the MicroMod ATP carrier board).

```
<<drive-input: main>>=
pub fn main() !void {
    const input_pin: svc_services.DevSvcInstance = 27; // micromod GPIO 3
    var input_changed: bool = false;

    try svc_services.gpio.inputConfig(
        input_pin, // pin number
        .low_high, // interrupt configuration
        false, // disarm when activates
        .none, // pullup resistance
        svc_services.exec_control.svcVarSyncNotifier, // simple variable synchronization
        @intFromPtr(&input_changed), // variable used for synchronization
    );

    while (true) {
        try svc_services.exec_control.runWait(&input_changed);
        std.debug.assert(input_changed == true);
        input_changed = false;
    }

    try svc_services.gpio.disable(input_pin);
}
```

The following are the results of increasing the square wave frequency until failure for the various release modes.

- ReleaseFast — 95 kHz, 10.5 μ s
- ReleaseSmall — 70 kHz 14.3 μ s
- ReleaseSafe — 95 kHz 10.5 μ s

We can approximate the time to handle the IRQ request, queue a background notification and dispatch the notification through the runWait execution control to be approximately the times listed above for the various optimization modes. This gives an approximate bound on how soon background execution can be made aware of input from the system environment.

Code Layout

```
<<drive-input-test.zig>>=
<<edit warning>>
<<copyright info>>

comptime {
    _ = @import("start_main");
```

```

}

const std = @import("std");
const svc_services = @import("svc_services");

<<drive-input: main>>

```

In and Out

GPIO pins are an extraordinarily useful means to provide insight into a running program with minimal effect on the program's execution. The usual strategy is to change the state of an I/O pin and measure the outcome using an external device such as a logic analyzer or oscilloscope.

In this example, the cycle time is measured from taking an interrupt on an input pin through the background notification and then back to the foreground to toggle an output pin. A waveform generator is attached to an input pin and the generator is configured to create a square wave. By measuring the waveform created on the output pin, a *phase shift* is observed between the rising edge of the input and the rising edge of the output. The following diagram shows a simplified timing diagram of what the example does.

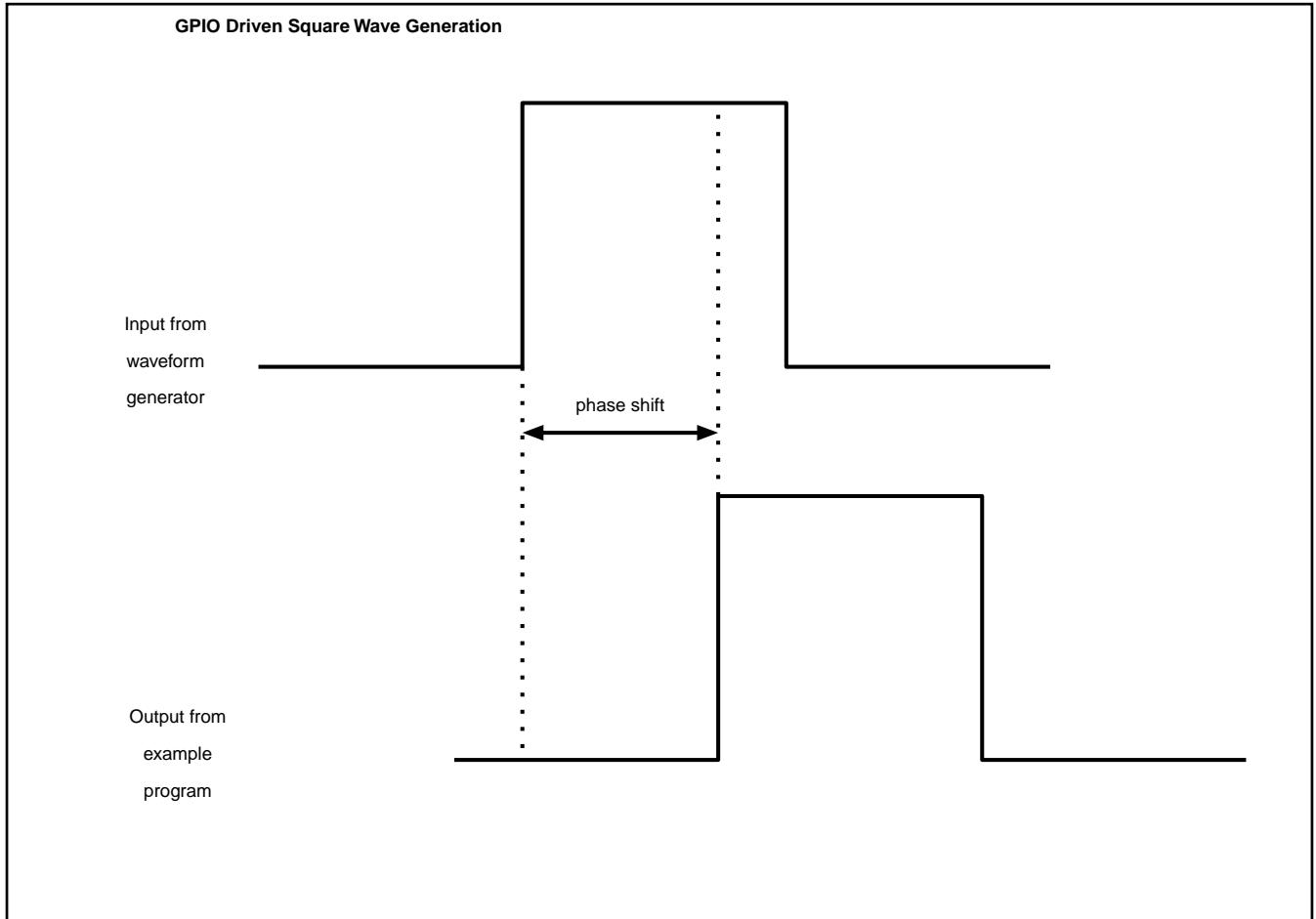


Figure 15. GPIO Example Timing Diagram

The amount of time between corresponding rising edges gives a measure of the time required to process:

1. the input pin IRQ
2. queue a background notification
3. retrieve and dispatch the notification
4. process a background request to change the state of the output pin.

The choice of GPIO pins to use in the example is for convenience. In this case, pins 14 and 15 are used.

```
<<in and out: main>>
const input_pin: svc_services.DevSvcInstance = 27; // micromod GPIO 3
const output_pin: svc_services.DevSvcInstance = 28; // micromod GPIO 4
```

Pin 14 is configured as an input which is triggered on both edges. Pin 15 is configured as an output. The background notification function for pin 14 simply requests immediately that pin 15 be toggled.

```
<<in and out: main>>
fn in_triggered(
    notification: *const anyopaque,
) void {
    const notify: *const DevNotification = @ptrCast(@alignCast(notification));
    svc_services.gpio.write(output_pin, if (notify.params gpio == 1)
        .pin_set
    else
        .pin_clear
    ) catch @panic("failed to write GPIO pin");
}
```

The `main` program configures the GPIO pins and runs an infinite event loop.

```
<<in and out: main>>
pub fn main() !void {
    try svc_services.gpio.inputConfig(
        input_pin, // pin number
        .both, // interrupt configuration
        false, // disarm when activates
        .none, // pullup resistance
        in_triggered, // background notification proxy
        0, // proxy closure
    );
    try svc_services.gpio.outputConfig(
        output_pin, // pin number
        .push_pull, // output type
        .@"2mA", // drive strength
        .none, // pull up resistance
    );
    var forever: bool = false;
}
```

```

try svc_services.exec_control.runWait(&forever);
}

```

The following figure shows the results of the waveform generated on the output pin as the example program attempts to mimic the changes in the input pin state. The code was built in safe mode. The waveform generator driving the input is running at 20 kHz. The phase shift between the input waveform and the output is approximately 25 μ s

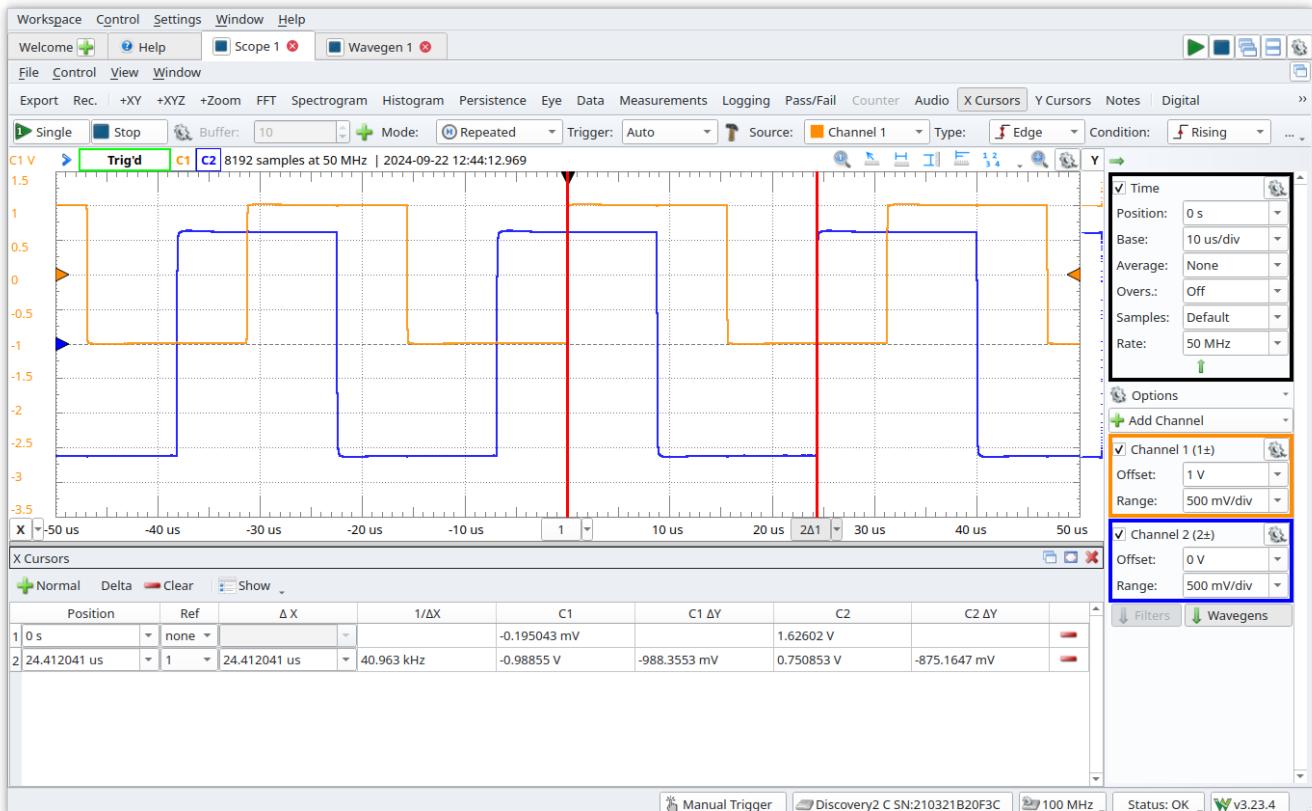


Figure 16. Waveform Generation Results

Taking a conservative value of 25 μ s as the turn around time, then the maximum frequency input waveform is approximately 20 kHz. This sets the lower time bound that the system can execute a response in background processing to a change in the system environment.

It is interesting to consider what happens to this program if the input generated waveform is greater than 20 kHz. In that case, background notifications cannot be removed and processed from the background notification queue quickly enough and the system is unable to reproduce the waveform. Eventually at a high enough frequency of input, in this case approximately 90 kHz, the processing cannot keep up and the background notification queue overflows. As the code is written, the GPIO IRQ handler detects the overflow and treats it as a **panic** condition.

Code Layout

```

<<top-in-out-test.zig>>
<<edit warning>>
<<copyright info>>

comptime {

```

```
_ = @import("start_main");
}

const std = @import("std");
const svc_services = @import("svc_services");
const DevNotification = svc_services.DevNotification;

<<in and out: main>>
```

Speak To Me

Asynchronous serial communications was invented long before microcontrollers. Universal Asynchronous Receiver-Transmitter (UART) peripherals are commonly included in microcontrollers SOC's. They support the lowest common denominator for communications. With the advent of USB-to-UART bridge chips, UART peripherals are much more convenient to use, eliminating the need for intricate cable connections.

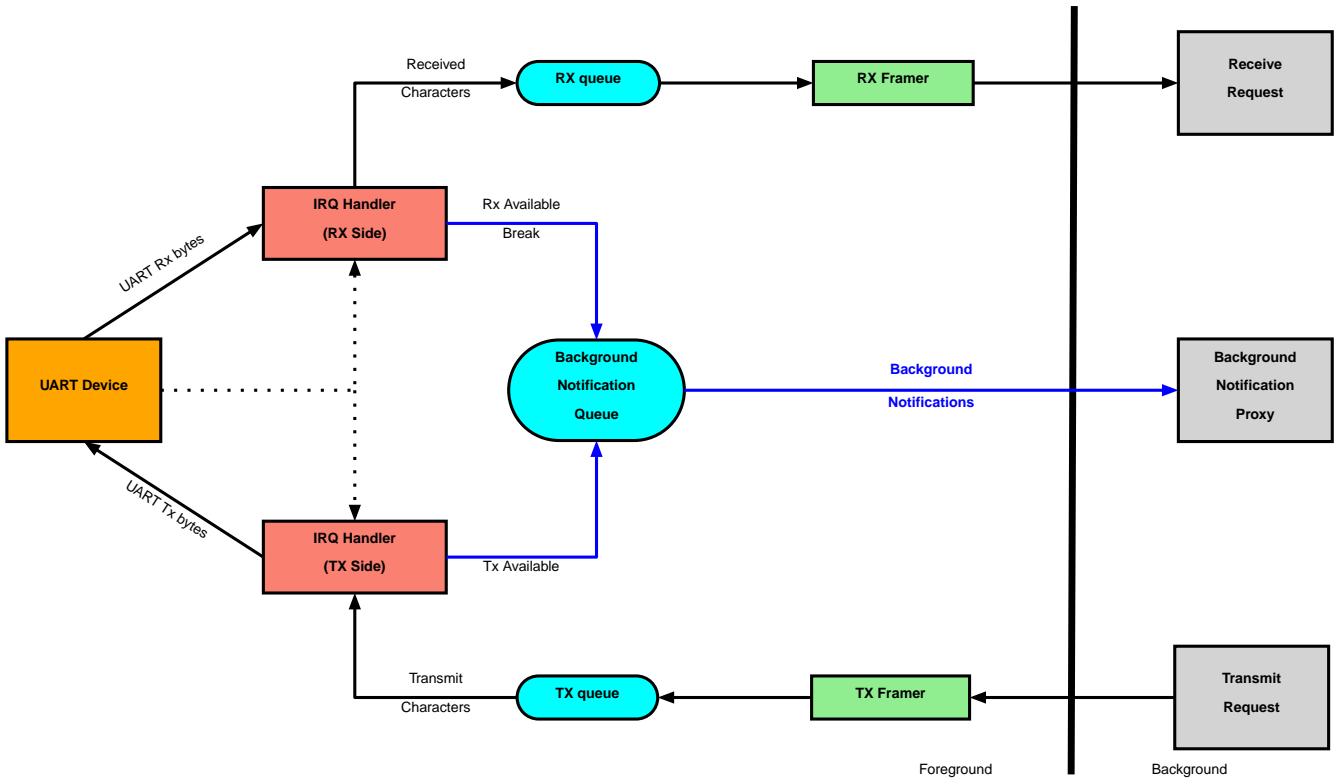
This chapter shows how asynchronous serial communication is implemented in the background/foreground scheme using the Apollo 3 UART module. UART's have a long and trying history of usage which has resulted in many variations of the communication schemes which use them. Serial communications by UART depends upon many external factors. For example, it is necessary for both sides in the communications to agree on speed, character size, parity inclusion, and character framing. This agreement must be done *a priori* and *out of band* [1]. The lack of any canonical link frame is another distinct problem. For example, an Ethernet packet is framed by the protocol so that the boundary of a packet can be determined as it is received. For UART communications, many such framings are in common use and for ordinary terminal I/O meant for a human, there are several conventions used to determine how lines (framing in this case) are terminated. This legacy of open-loop control creates substantial "accidental" complexity, but the simplicity and common availability of a UART warrants a workaround for its shortcomings. Again, no attempt is made to provide a comprehensive implementation which supports all the many variations. Only those configuration features of a UART that are most commonly used are implemented and those projects which have specific needs bear the burden of the additional work involved.

Design Overview

The UART device code is designed to meet the following requirements:

- Receiving and transmitting data is interrupt driven. The IRQ handler code will transfer data to and from the UART FIFO registers. The UART peripheral on the Apollo 3 does not support DMA operations, so the processor must move the data on and off the peripheral.
- Since there are several possible link frames formats, receive and transmit data are buffered in the foreground to allow for determining if a link frame has been received or to provide additional formatting for building a frame for transmission. Unlike other cases where buffering or memory management in foreground processing was avoided, this design uses a buffer on both the transmit and receive sides.
- Potential input frames are detected by a set of framing procedures. For transmitting, the TX framing adds bytes into the outgoing data to make up a frame. For receiving, the raw bytes received may have characters removed to match the type of the framing. Passing through all characters unmodified is also supported so that background processing can implement its own link framing conventions. The provided framing mechanism allows us to handle raw receive/transmit, terminal oriented I/O, and human interactive framing schemes.
- Options to configure the UART device characteristics are limited. The baud rate and hardware flow control can be configured. The peripheral device is otherwise set to 8-bit characters, no parity, and one stop bit as these are common in current usage.

The following figure shows a simplified schematic of how data is transferred to and from the UART.



Consider the receive path in the previous diagram. The foreground code has an internal queue it uses to hold the bytes as received from the UART. When the receive FIFO has reached its threshold, the UART device raises an interrupt and control transfers to the portion of the IRQ handler used for receiving. The handler reads bytes from the UART device FIFO and writes them to a receive queue. If the queue becomes full, then reception either overruns or, if hardware flow of control is supported, the transmitting peer is signaled that it must hold up transmission. When receive requests are made by the background, the RX framing removes characters from the RX queue to enforce link frame boundaries. Notifications indicate that additional characters are available and are posted when the previous receive request was not completely filled (*i.e.* a “short” read) and new characters have arrived.

Background processing is responsible for requesting incoming data until it receives what it expects and to use RX available notifications when more is expected.

The transmit path is similar. Background processing makes transmit requests to send data. The transmit data is placed in the TX queue by the TX framing. The UART is fed characters from the TX queue when it has space in its FIFO for more outgoing characters. Background notifications are issued when a transmit request was not completely fulfilled (*i.e.* a “short” write) and there is now space available in the transmit buffer.

There is also a concept of a session with the UART. Before sending or receiving data, the application must *open* the UART. Opening the UART provides the address of a background proxy function which receives the RX and TX notifications.

Applications should take care to only open the UART for an ongoing communications session and to close it when finished in order to conserve power. The objective is to open the device, promptly do what is required, and close the device immediately when no longer needed. However, for some UART needs, *e.g.* console I/O used for status reporting and debugging, a UART may be expected to be left open for the lifetime of the program.

The following sections describe the details of how the UART requests are designed to operate and

give the Zig source code. The presentation order of the implementation follows from a left to right viewing of the previous diagram.

Representing a UART

Continuing our established pattern, this section describes the logical characteristics of the UART device and how those characteristics are represented in code.

UART Logical Device Control

This section describes the data and code required to operate on the logical UART peripheral device. This involves the settings and controls that combine all the physical peripheral devices into a coherent unit.

The necessary controls needed to implement data transfer across the UART involves much more than just the UART peripheral itself. Additional considerations are:

NVIC

The NVIC is used clear and enable the interrupt globally in the processor core.

GPIO Peripheral

Because a UART uses wired communications, there is a connection to the outside world. The configuration of the I/O pins used by the UART peripheral to connect to the physical transmission medium must be handled.

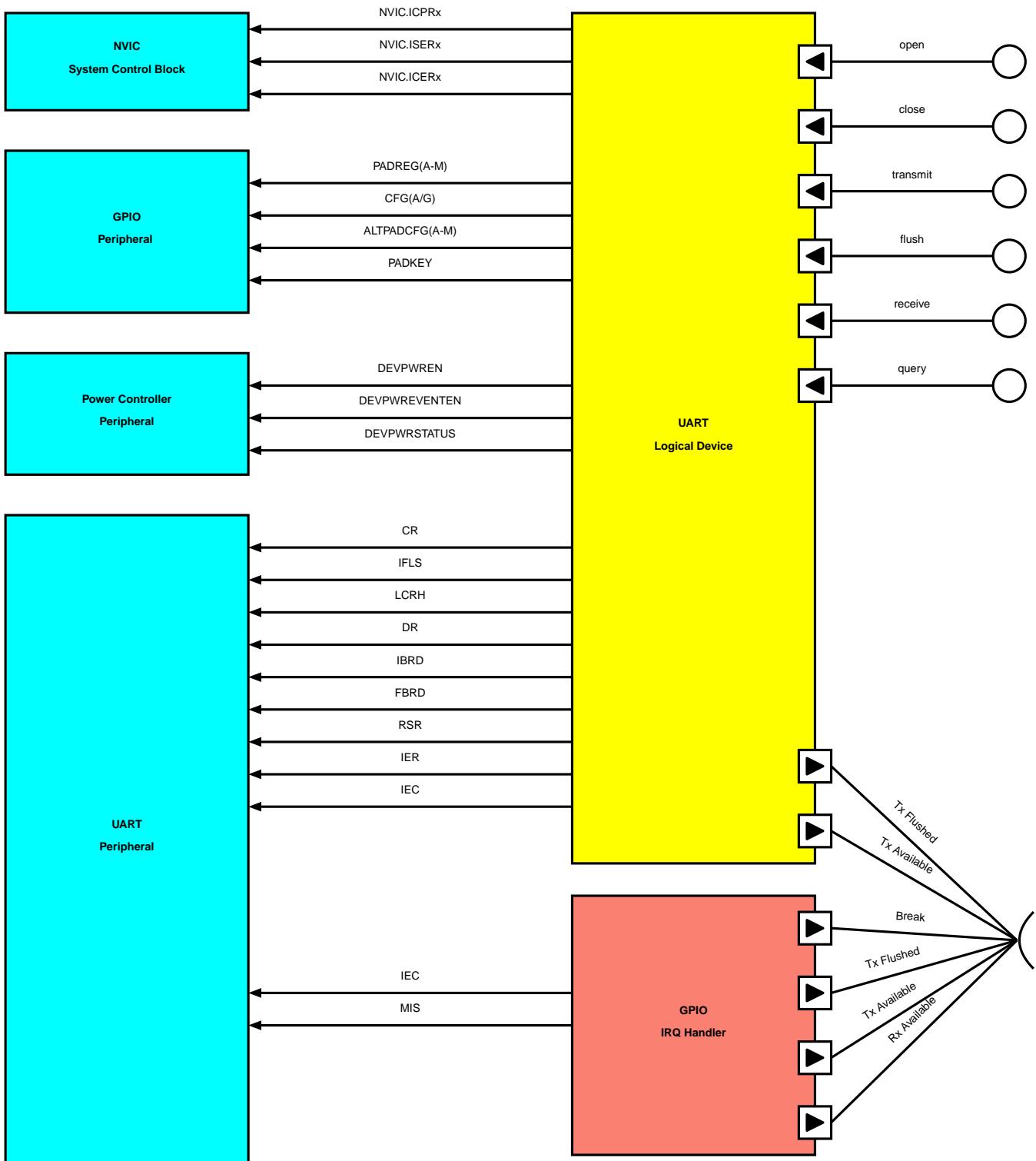
UART Peripheral

The UART peripheral is represented in the usual memory mapped manner.

Power Controller Peripheral

The Apollo 3 SOC has separate power enables for the UART peripherals which by default are turned off.

The following diagram shows the combination of peripheral devices used to implement the logical UART device. Notice that in addition to the expected interactions with the NVIC and the UART peripheral, the UART device also uses the GPIO and Power Controller peripherals. The GPIO peripheral is used to set up the pins which connect the UART to the environment. The Power Controller controls the power domain associated with the UART peripheral. There are more than usual peripheral block interactions required for the UART device.



The following data structure contains the elements needed to control communications using the UART peripheral and implement the buffer queuing.

```
<<uart proxies>>
/// The core data structure used to control the UART device.
const UartDevice = struct {
    /// The instance number of this UART.
```

```

instance: DevSvcInstance,
/// Control data and processing for handling NVIC interactions.
irq_control: IrqControl,
/// Control data and processing for handling I/O pin assignments.
pin_control: PinControl,
/// Data and processing for controlling power to the UART peripheral.
power_control: PowerControl,
/// Control data and processing for interacting with the UART peripheral device.
uart_control: UartControl,
/// The type of framing to be used for transmitting and receiving.
framing: FramingType,
/// Data required to control the transmit side of the UART.
tx_control: TxControl,
/// Data required to control the receive side of the UART.
rx_control: RxControl,
/// State information indicating whether a transmitter flush request is pending.
flush_pending: bool,
/// Background proxy notification function.
proxy: DevSvcNotifyProxy,
/// A data value supplied by the user of the UART device and returned in the
/// background notifications from the UART.
closure: usize,

<<uart control functions>>
};

```

```

<<uart proxies>>=
/// The type of the structure which overlays the memory mapped
/// control registers for the UART controller.
const UartController = apollo3.UartController;

```

The following sections discuss the components of the `UartDevice` data structure.

UART Peripheral Control

The data sheet for the Apollo 3 UART module does not contain a sufficient description to know all the necessary rules for how the UART peripheral operates. However, from a close examination of the register descriptions it appears that the UART peripheral is an [ARM PrimeCell part](#). The technical reference manual for the PL011 UART has been used to supplement the information in the Ambiq data sheet.

```

<<uart proxies>>=
/// The `UartControl` data and method provide for direct control of the
/// UART peripheral itself.
const UartControl = struct {
    uart: *volatile UartController,

    <<uart peripheral control functions>>
};

```

```
<<uart peripheral control functions>>=
/// Enable the interrupt necessary to support transmission.
fn enableTxInterrupt(
    self: UartControl,
) void {
    self.uart.ier.updateField(.tx, 1);
}
```

```
<<uart peripheral control functions>>=
/// Disable the transmit interrupt at the peripheral chip.
fn disableTxInterrupt(
    self: UartControl,
) void {
    self.uart.ier.updateField(.tx, 0);
}
```

```
<<uart peripheral control functions>>=
/// Enable the interrupts necessary to support reception.
fn enableRxInterrupt(
    self: UartControl,
) void {
    self.uart.iec.writeBitField(.rx);
    self.uart.iec.writeBitField(.rt);
    self.uart.ier.updateFields(.{
        .rx = 1,
        .rt = 1,
    });
}

/// Disable the interrupts necessary used to support reception.
fn disableRxInterrupt(
    self: UartControl,
) void {
    self.uart.ier.updateFields(.{
        .rx = 0,
        .rt = 0,
    });
}
```

```
<<uart peripheral control functions>>=
/// Check if the transmit FIFO is full.
fn txFifoFull(
    self: UartControl,
) bool {
    return self.uart.fr.readField(.tx_ff) == 1;
}
```

```
<<uart peripheral control functions>>=
/// Check if the transmit FIFO is empty.
```

```
fn txFifoEmpty(
    self: UartControl,
) bool {
    return self.uart.fr.readField(.tx_fe) == 1;
}
```

```
<<uart peripheral control functions>>=
/// Check if the receive FIFO is empty.
fn rxFifoEmpty(
    self: UartControl,
) bool {
    return self.uart.fr.readField(.rx_fe) == 1;
}
```

```
<<uart peripheral control functions>>=
/// Transfer a character to the UART transmitter.
fn loadTxChar(
    self: UartControl,
    tx_char: u8,
) void {
    self.uart.dr.writeField(.tx_data, tx_char);
}
```

```
<<uart peripheral control functions>>=
/// Obtain a character from the UART receive.
fn unloadRxChar(
    self: UartControl,
) RxChar {
    return self.uart.dr.readField(.rx_data);
}
```

```
<<uart peripheral control functions>>=
/// Check if the transmitter is currently busy transmitting characters.
fn txBusy(
    self: UartControl,
) bool {
    return self.uart.fr.readField(.tx_busy) == 1;
}
```

```
<<uart peripheral control functions>>=
/// Check if the transmitter is currently busy transmitting characters.
fn clearToSend(
    self: UartControl,
) bool {
    return self.uart.fr.readField(.cts) == 1;
}
```

```
<<uart peripheral control functions>>
/// Disable the UART as well as any interrupt sources within the UART.
fn clear(
    self: UartControl,
) void {
    self.uart.cr.writeDirect(0);
    self.uart.ier.writeDirect(0);
    self.uart.iec.writeDirect(~@as(mmio.WordRegister, 0));
}
```

```
<<uart peripheral control functions>>
/// Enable the UART. This is typically done after it has been
/// otherwise initialized.
fn enable(
    self: UartControl,
) void {
    self.uart.cr.setBitField(.uart_en);
}
```

```
<<uart peripheral control functions>>
/// Configure the levels at which the UART FIFO's cause interrupts.
/// Note there are no choices here.
fn configFifos(
    self: UartControl,
) void {
    self.uart.ifls.writeFields(.{
        .rx_ifl_sel = .@"3/4", // 3/4 full for receiving
        .tx_ifl_sel = .@"1/4", // 1/4 remaining for transmit
    });
}
```

```
<<uart peripheral control functions>>
/// Configure the length of a character for the UART.
/// Note there are no choices here -- the setting is 8 bits, no parity.
fn configCharLength(
    self: UartControl,
) void {
    self.uart.lchr.writeFields(.{
        .wlen = .@"8bits",
        .fen = 1,
    });
}
```

```
<<uart peripheral control functions>>
/// Configure clocking, rx / tx, and possibly rts / cts.
fn configControls(
    self: UartControl,
    hdwr_flow_ctrl: bool,
) void {
```

```

    self.uart.cr.updateFields(.{
        .clk_en = 1,
        .clk_sel = .@"24MHz",
        .rxe = 1,
        .txe = 1,
    });
    if (hdwr_flow_ctrl) {
        self.uart.cr.updateFields(.{
            .cts_en = 1,
            .rts_en = 1,
        });
    }
}

```

```

<<uart peripheral control functions>>=
/// Place a character into the transmitter, waiting if transmitter
/// is full.
fn putTxChar(
    self: UartControl,
    tx_char: u8,
) void {
    // Make sure we don't hang forever. This can happen when hardware flow
    // control is in effect and there is no connection on the other
    // end of the UART that asserts CTS and so the FIFO never drains.
    var limit: usize = 50;
    while (limit > 0) : (limit -= 1) {
        if (!self.txFifoFull()) {
            self.loadTxChar(tx_char);
            break;
        }
    } else {
        @panic("cannot load character into Tx FIFO");
    }
}

```

```

<<uart peripheral control functions>>=
/// Place a buffer of characters into the transmitter, waiting as
/// necessary to transmit the entire buffer.
fn putTxBuffer(
    self: UartControl,
    buffer: []const u8,
) void {
    for (buffer) |char| {
        putTxChar(self, char);
    }
}

```

Baud rate divisor computations are often quite fussy since asynchronous communications requires both sides to have reasonably precise clocks which have to be generated from the clock sources available at each end of the connection. There are a number of schemes used by different UART

peripheral designs. In the Apollo 3 case, the baud rate divisor has an integer portion of 16 bits and a fractional portion of 6 bits. In other words, the baud rate divisor is an unsigned UQ16.6 fixed binary point number. That number is stored in the hardware as two separate registers, one register for the integer portion and one for the fractional portion. The formula for computing the baud rate divisor is given in the data sheet as, $F_{UART}/(16 \times BR) = IBRD + FBRD$, where F_{UART} is the UART clock frequency and BR is the desired baud rate. The term on the right hand side of the equals sign is interpreted to mean the fixed binary point result in its separated component form. This seems to be born out by the HAL code in the SDK, albeit the calculation there is done rather strangely, indicating the author didn't have much experience in fixed radix point arithmetic. The ARM manual for the PL011 UART has a better description of the calculation on p. 3-10. The usual concern here is to avoid overflow during the calculation. The fixed binary point division is performed as a 64-bit value since multiplying the maximum UART clock frequency of 24 MHz by 64 (6 fraction bits) overflows 32 bits. Otherwise, the usual fixed binary point calculation in 64-bit variable is used and then the integer and fractional parts are separated out so they may be written to registers.

```
<<uart peripheral control functions>>=
fn configBaudRate(
    self: UartControl,
    baud_rate: u32,
) void {
    const ureg = self.uart;
    const UartClockFrequency = std.enums.EnumArray(UartController.ClockFreq, u32);
    const clock_freq_map = comptime UartClockFrequency.init({
        .nclk = 0,
        @"24MHz" = 24_000_000,
        @"12MHz" = 12_000_000,
        @"6MHz" = 6_000_000,
        @"3MHz" = 3_000_000,
    });

    const clk_sel = ureg.cr.readField(.clk_sel); ①
    const clock_freq = clock_freq_map.get(clk_sel);
    std.debug.assert(clock_freq != 0);

    const br_frac_bits =
        @bitSizeOf(@FieldType(UartController.FbrdRegister, @tagName(.div_frac)));
    const max_idiv =
        math.maxInt(@FieldType(UartController.IbrdRegister, @tagName(.div_int)));
    const max_fdiv =
        math.maxInt(@FieldType(UartController.FbrdRegister, @tagName(.div_frac)));

    const br_div_numerator =
        (@as(u64, clock_freq) << br_frac_bits) + @as(u64, max_fdiv / 2); ②
    const br_div_denom = @as(u64, baud_rate) * @as(u64, 16);
    const br_divisor: u32 = @truncate(br_div_numerator / br_div_denom);

    const i_divisor = br_divisor >> br_frac_bits;
    const f_divisor = br_divisor & max_fdiv;

    std.debug.assert(i_divisor != 0); ③
    std.debug.assert(i_divisor <= max_idiv);
    std.debug.assert(i_divisor != max_idiv or f_divisor == 0);
}
```

```

const lchr = ureg.lchr.read();
ureg.ibrd.writeField(.div_int, @truncate(i_divisor));
ureg.fbrd.writeField(.div_frac, @truncate(f_divisor));
ureg.lchr.write(lchr); ④
}

```

- ❶ Note that the UART clock rate frequency must be set before invoking this function.
- ❷ For the numerator of the baud rate calculation, the clock frequency is scaled by the desired number of fraction bits. The second term is a rounding term. This rounds up values greater than half the fraction amount and rounds down those half or less.
- ❸ Check the bounds for the integer and fractional parts. Not all combinations of baud rates and UART clock frequencies yield a valid divisor.
- ❹ According to the [ARM documentation](#), the LCRH, IBRD, and FBRD registers are internally a single register which has been split out into three registers at the memory interface. The internal register is only updated when the LCRH register is written (p. 3-14). To update only the IBRD and FBRD registers, a write to LCRH must follow the updates to the IBRD and FBRD registers. Hence the pirouette of a register read followed by three register writes. The LCRH register is read to get its current value, update the IBRD and FBRD registers and follow those updates with a write to the LCRH register of its current value. Also note that as of V 2.5.1 of the Ambiq SDK, the HAL code for configuring the baud rate has a “TODO” comment about the sequence used to write the baud rate divisor registers. The code is not correct in that it does not write to LCRH, but the baud rate does get set properly since the LCRH register is written later as part of the HAL code’s UART configuration.

Power Control

Like many SOC’s, the Apollo 3 has a power controller peripheral to enable the fine control needed in low power systems. To conserve power, peripherals that are not used need not be powered up.

```

<<device access>>=
const PowerControl = struct {
    /// The bit field indicating the particular peripheral device for
    /// which the power is controlled. These bit fields help when dealing
    /// when multiple peripheral instances show up in other control
    /// registers as discretely named fields. For example, although
    /// UART 0 and 1 are distinct instances, the power controller has a
    /// control register with fields named `pwr_uart_0` and `pwr_uart_1`.
    /// This makes it impossible to use comptime enumeration literals to
    /// identify the control register bit fields. Hence, we have to handle
    /// runtime bit fields in order to separate power controls for the
    /// two UART peripheral instances.
    device: @TypeOf(apollo3.pwrctrl.dev_pwr_en).BitField,
    /// The power domain in which the controlled device resides.
    domain: @TypeOf(apollo3.pwrctrl.dev_pwr_event_en).BitField,

    fn init(
        comptime device_enable: @Type(.enum_literal),
        comptime power_domain: @Type(.enum_literal),
    ) PowerControl {
        return .{
            .device = .init(device_enable),
            .domain = .init(power_domain),
        }
    }
}

```

```

    };
}

fn power_up(
    self: *const PowerControl,
) void {
    // When power to a peripheral device is applied it takes
    // some time for the circuitry to stabilize and be ready.
    // The Apollo3 Power Controller supports sending an MCU event
    // to the processor when the power to a domain is turned on
    // and ready. We use that capability here to prevent a
    // polling loop. It's a small power consumption gain,
    // so we use what is available.
    apollo3.pwrctrl.dev_pwr_event_en.insertFieldValue(self.domain, 1);
    apollo3.pwrctrl.dev_pwr_en.insertFieldValue(self.device, 1);
    while (apollo3.pwrctrl.dev_pwr_status.extractFieldValue(self.domain) == 0) {
        v7m.wfe();
    }
    apollo3.pwrctrl.dev_pwr_event_en.insertFieldValue(self.domain, 0);
}

fn power_down(
    self: *const PowerControl,
) void {
    apollo3.pwrctrl.dev_pwr_en.insertFieldValue(self.device, 0);
}
};

```

GPIO Pin Control

Since the UART must be wired to the external environment, GPIO pins must be configured with the proper function and electrical characteristics. The only complication here is handling hardware flow of control, which requires two extra pins to be managed. Also, not all UART peripherals can be used with hardware flow of control. This is discussed [below](#) when the UART configuration on the Artemis MicroMod is presented.

```

<<uart proxies>>=
/// The `PinControl` type holds that information for handling the
/// I/O pin interactions required to attach the UART to external pins.
const PinControl = struct {
    hdwr_flow_support: bool,
    tx_pin: UartIoPin,
    rx_pin: UartIoPin,
    rts_pin: UartIoPin,
    cts_pin: UartIoPin,

    /// Allocate all the GPIO pins required for a UART. Note this
    /// is an all or nothing operations. One either gets all the
    /// required pins or no pins are allocated.
    fn allocate(
        self: *const PinControl,
        enable_hdwr_flow: bool,

```

```

) !void {
    try self.tx_pin.allocate();
    errdefer self.tx_pin.free();

    try self.rx_pin.allocate();
    errdefer self.rx_pin.free();

    if (enable_hdwr_flow and self.hdwr_flow_support) {
        try self.rts_pin.allocate();
        errdefer self.rts_pin.free();

        try self.cts_pin.allocate();
    }
}

fn free(
    self: *const PinControl,
) void {
    self.tx_pin.free();
    self.rx_pin.free();
    if (self.hdwr_flow_support) {
        self.rts_pin.free();
        self.cts_pin.free();
    }
}

fn configureTxRx(
    self: *const PinControl,
) void {
    self.tx_pin.configure();
    self.rx_pin.configure();
}

fn configureHdwrFlowControl(
    self: *const PinControl,
) void {
    self.rts_pin.configure();
    self.cts_pin.configure();
}

fn disable(
    self: *const PinControl,
) void {
    self.tx_pin.disable();
    self.rx_pin.disable();
    if (self.hdwr_flow_support) {
        self.rts_pin.disable();
        self.cts_pin.disable();
    }
}
};


```

```

<<uart proxies>>=
const Gpio = apollo3.Gpio;

/// The `UartIoPin` type collects together the information to control
/// the pins and pads required for a UART. Note the use of functions
/// provided by the `gpio` proxy that perform the physical I/O for
/// configuring GPIO pins and pads.
const UartIoPin = struct {
    const PadConfigType = gpio.PadConfigType;
    const GpioConfigType = gpio.GpioConfigType;
    const AltPadConfigType = gpio.AltPadConfigType;

    pin: u8,
    pad_cfg: PadConfigType,
    pin_cfg: GpioConfigType,
    alt_pad_cfg: AltPadConfigType,

    fn allocate(
        self: UartIoPin,
    ) !void {
        try gpio_allocator.allocate(self.pin);
    }

    fn free(
        self: UartIoPin,
    ) void {
        gpio_allocator.free(self.pin);
    }

    fn configure(
        self: UartIoPin,
    ) void {
        gpio.pin_control.configure(self.pin, self.pad_cfg, self.pin_cfg, self.alt_pad_cfg);
    }

    fn disable(
        self: UartIoPin,
    ) void {
        gpio.pin_control.disable(self.pin);
    }
};

```

When the Apollo 3 is incorporated into the Sparkfun MicroMod Artemis processor board, UART0 is attached to the USB-to-serial converter which is part of the MicroMod processor board. The only UART pins connected are those for receive and transmit lines. However, the RTS pin of the USB-to-serial converter is connected to the processor reset line. This means that when the device is opened on the host computer and the RTS line is asserted, the Apollo 3 processor is reset. This is convenient for using a boot loader to download an executable which is a primary use case supported by the Sparkfun environment. But in our circumstances the processor reset disrupts a JLink debugger and we need another method to obtain output from the system during JLink debugger sessions. UART0 is serviceable at other times as long as you remember that opening UART0 causes a system reset.

The UART1 module is wired to the MicroMod connector as TX1, RX1, RTS1, and CTS1. This allows hardware flow control via RTS and CTS and it appears as the primary UART. The MicroMod connector also defines a RX2 and TX2 connection. These are not connected when the Artemis processor serves as in the MicroMod system processor.

This particular hardware design implies that hardware flow control may only be configured on UART1. UART0 may be configured and used for those cases where the implied reset is acceptable. The most frequently anticipated use case is for UART1 carrying output to a connected host via the USB-to-serial converter. This will provide a convenient and speedy output mechanism which is standalone and always available.

The following table shows the pin labeling and gives the pin function select value which encodes the pin for use with the UART peripheral in a given role.

Artemis Pin Label	Apollo 3 Pin Number	Function Select Value
TXO-0	48	0
RXI-0	49	0
TX1	12	7
RX1	25	0
RTS1	10	5
CTS1	17	7

Table 5. UART Pin Mappings

Using the previous table, the hardware configuration information is specified as an initialized variable. Note the variable is constant as this information does not change at run time. TX and RTS pins must be configured to disable input and with no pull up resistor. RX and CTS pins must be configured to enable input, also with no pull up resistor.

UART TX and RX Controls

The last few section described the control of the UART peripheral and its ancillary supporting peripherals. That covers the hardware aspects of the logical UART. In this section, we describe the internal data and code used to support the buffering and framing of UART data.

Since the UART device is capable of *full duplex* communications, the transmit and receive sides are treated separately. The data required to handle both sides is similar, but not identical. For transmission, the following data structure is used.

```
<<uart proxies>>=
/// The `TxControl` type holds the data required to manage transmission on a UART.
const TxControl = struct {
    const TxChar = u8; // We only transmit bytes.

    /// A FIFO queue to support buffering for the transmit side.
    queue: BipBuffer(TxChar),
    /// Storage for buffer Tx characters.
    storage: [tx_queue_size]TxChar = undefined,
    /// Character storage available to the transmit framing function for any
    /// holding state history of the previously seen characters.
```

```

set_aside: [4]u8,
/// An index variable used as an iterator in the TX queue. This member is
/// available to those framing functions which require tracking individual
/// characters in the TX queue.
cursor: usize,
/// A boolean status indicating whether the last transmit request was able to
/// queue all the requested bytes.
short_write: bool,

fn reset(
    self: *TxControl,
) void {
    self.queue = BipBuffer(TxChar).init(&self.storage);
    @memset(&self.set_aside, 0);
    self.cursor = 0;
    self.short_write = false;
}
};


```

For receiving, there are differences in the receive queue and receive framing functions. The following data type records the information for controlling the receive side.

```

<<uart proxies>>=
/// The `RxControl` type holds the data required to manage reception on a UART.
const RxControl = struct {
    /// A FIFO queue to support buffering for the receive side.
    queue: BipBuffer(RxChar),
    /// Storage for buffer Rx characters.
    storage: [rx_queue_size]RxChar = undefined,
    /// Character storage available to the receive framing function for any holding
    /// state history of the previously seen characters.
    set_aside: [4]u8,
    /// An index variable used as an iterator in the RX queue. This member is
    /// available to those framing functions which require tracking individual
    /// characters in the RX queue.
    cursor: usize,
    /// A boolean status indicating whether fewer than the number of bytes
    /// requested in the last receive request were transferred.
    short_read: bool,

    fn reset(
        self: *RxControl,
    ) void {
        self.queue = BipBuffer(RxChar).init(&self.storage);
        @memset(&self.set_aside, 0);
        self.cursor = 0;
        self.short_read = false;
    }
};


```

We use a [Bipartite Buffer](#) to implement a data queue for both receive and transmit. This type of

buffer is noteworthy because all blocks reserved from the buffer are contiguous in memory. This characteristic is most convenient for some of the framing operations that are performed in Rx and Tx data.

```
<<uart proxies>>=
const BipBuffer = @import("BipBuffer.zig").BipBuffer;
```

The RX queue is slightly different from the Tx one. The UART peripheral device augments each received byte with 4 bits of error status. The error status is tracked through the RX queue and error information is handed back whenever the corresponding received byte is read by background processing. So, 12-bit quantities are required to hold the RX data when extended by its status information. The register definitions for the UART peripheral have a suitable data type.

```
<<uart proxies>>=
/// The type of a received character when it is obtained directly
/// from the UART receiver. Additional bits of error information
/// are included.
const RxChar = UartController.RxCharWithError;
```

The queue storage is sized generously enough for common usage. Much more transmitted data is expected than received data.

```
<<system_config: constants>>=
// The number of characters in the Tx queue
pub const tx_queue_size: usize = 2048;
// The number of characters in the Rx queue
pub const rx_queue_size: usize = 256;
```

```
<<uart proxies>>=
const tx_queue_size = system_config.tx_queue_size;
const rx_queue_size = system_config.rx_queue_size;
```

Framing Transformations

When transmit data is copied into the TX queue as part of the `transmit` service and when received data is copied out of the RX queue as part of the `receive` service, the data stream is framed by adding (in the TX case) or subtracting (in the RX case) bytes to either encapsulate or decode the data. This design supports a general mechanism to invoke framing functions and supplies framing functions for three common situations.

Identity

All data is passed through unmodified in both directions.

Terminal

When receiving, carriage return (CR), linefeed (LF) or CR/LF sequences are converted to LF and the frame is returned after any LF character. When transmitting, LF is converted to CR/LF. This allows input records to be LF separated internally, but converted to CR/LF sequences on output.

Interactive

When transmitting, interactive framing is the same as terminal framing. When receiving,

interactive framing echos input characters and responds to line editing characters (e.g. backspace (BS)). Interactive framing is suitable for use as a console directly interfacing to a human.

```
<<uart services>>=
/// An enumeration to specify the type of framing to be done on characters
/// passing through the UART.
pub const FramingType = enum {
    identity,
    terminal,
    interactive,
};
```

```
<<uart proxies>>=
const FramingType = svc_services.uart.FramingType;
```

The framing mechanism can support an arbitrary number of framing transformers. The three provided are for common usage, but framing functions for better defined link frames such as Serial Line Internet Protocol (SLIP) or common modem communications frames can be integrated into the mechanism.

Because the TX and RX framing functions transfer data from and to unprivileged memory, these functions are required to perform that transfer in an unprivileged manner.

There are separate framing functions for transmit and receive.

```
<<uart control functions>>=
fn txFramer(
    udev: *UartDevice,
    src: []const u8,
) usize {
    return switch (udev.framing) {
        .identity => identityTxFramer(udev, src),
        .terminal => terminalTxFramer(udev, src),
        .interactive => interactiveTxFramer(udev, src),
    };
}
```

```
<<uart control functions>>=
fn rxFramer(
    udev: *UartDevice,
    dest: []u8,
) struct {usize, RxCharError} {
    return switch (udev.framing) {
        .identity => identityRxFramer(udev, dest),
        .terminal => terminalRxFramer(udev, dest),
        .interactive => interactiveRxFramer(udev, dest),
    };
}
```

Since error information is tracked through the receiving mechanism, additional typing is required to handle them.

```
<<uart proxies>>=
const RxCharError = UartController.RxCharError;
const RxCharInt = meta.Int(.unsigned, @bitSizeOf(RxCharError));
const rx_status_success: RxCharError = .{};

fn isRxSuccess(
    rx_status: RxCharError,
) bool {
    return @as(RxCharInt, @bitCast(rx_status)) == 0;
}
```

Identity Framing Transformation

The identity framer performs no transformation on characters and passes through all characters received or transmitted. This type of framing can be used by background processing for application specific framing.

For transmitting, the identify framer just copies the data from the caller's buffer into the TX buffer. The copy continues until there is either no more caller data or the TX buffer is full

```
<<uart control functions>>=
fn identityTxFramer(
    udev: *UartDevice,
    src: []const u8,
) usize {
    const queue = &udev.tx_control.queue;
    // Tx framing involves inserting characters into the Tx queue.
    // The Bip Buffer operates with a reserve / write / commit protocol.
    const dest = queue.reserve(src.len);

    const write_count = @min(dest.len, src.len);
    memcpyFromUnpriv(dest[0..write_count], src[0..write_count]);

    queue.commit(write_count);
    return write_count;
}
```

```
<<uart control functions>>=
fn identityRxFramer(
    udev: *UartDevice,
    dest: []u8,
) struct {usize, RxCharError} {
    const queue = &udev.rx_control.queue;
    // Rx framing involves transferring received data to the caller.
    // Here, the Bip Buffer operations with a get block / copy out data / decommit block
    // protocol.
    const src = queue.getContiguousBlock();
    const read_count = @min(src.len, dest.len);
```

```

const count: usize, const recv_status: RxCharError =
    copyOutRxChars(dest[0..read_count], src[0..read_count]);

queue.decommitBlock(count);
return { count, recv_status };
}

```

There are a couple of cases that arise when transferring Rx data to the caller. In one situation, we want to check for any received errors as the data is passed to the caller. Any errors will cause the transfer to stop and report the error.

```

<<uart control functions>>=
fn copyOutRxChars(
    dest: []u8,
    src: []RxChar,
) struct {usize, RxCharError} {
    var count: usize = 0;
    var recv_status = rx_status_success;
    while (count < dest.len and isRxSuccess(recv_status)) : (count += 1) {
        const char_with_err = src[count];
        recv_status = char_with_err.data_err;
        v7m.strbt(char_with_err.data, &dest[count]);
    }
    return { count, recv_status } ;
}

```

The other situation arises when an Rx framer holds back portions of the received data waiting for the entire frame to arrive. In this case, any received error is checked as data is placed in the Rx queue. When a frame is detected, we do not have to check for received error again.

```

<<uart control functions>>=
fn copyOutRxCharsUnchecked(
    dest: []u8,
    src: []RxChar,
) usize {
    var count: usize = 0;
    while (count < dest.len) : (count += 1) {
        v7m.strbt(src[count].data, &dest[count]);
    }
    return count;
}

```

Terminal I/O Framing Transformation

Terminal I/O framing is intended for use with conventional CR/LF terminated records. The framing is flexible to allow the usual permutations of CR and LF as either solitary record terminators or as a sequence to indicate frame termination.

The implementation of `terminalTxFramer` uses the `set_aside` buffer to hold the last character seen in the stream. This gives us the necessary state information to handle inserting a CR/LF

sequence into the transmit buffer when either a single LF character is transmitted or a CR/LF sequence is present in the data stream. Note that isolated CR characters are simply passed through.



There appears to be a compiler error that shows up in this function as of Zig version 0.15.1. It appears that the incrementation of `src_index` happens in the wrong place. This causes an out of bounds panic condition when the `src` slice consists of only a NL character. The patch in the `for` loop appears to correct things. Very strange indeed.

```
<<uart control functions>>=
fn terminalTxFramer(
    udev: *UartDevice,
    src: []const u8,
) usize {
    if (src.len == 0) return 0;

    const control = &udev.tx_control;
    const queue = &control.queue;

    var nl_count: usize = 0;
    for (src) |*src_slot| {
        const src_char = v7m.ldrbt(src_slot);
        if (src_char == '\n') nl_count += 1; ①
    }

    const resv_len = src.len + nl_count;
    const dest = queue.reserve(resv_len);
    const xfer_count = @min(resv_len, dest.len);
    var src_index: usize = 0;
    nl_count = 0;

    for (dest[0..xfer_count]) |*dest_slot| {
        if (src.len == 1 and src_index == 1) src_index = 0; // NOTE: miscompilation
workaround
        const tx_char = v7m.ldrbt(&src[src_index]);
        if (tx_char == '\n' and control.set_aside[0] != '\r') {
            nl_count += 1;
            control.set_aside[0] = '\r';
            dest_slot.* = '\r'; ②
        } else {
            control.set_aside[0] = tx_char;
            dest_slot.* = tx_char;
            src_index += 1;
        }
    }

    queue.commit(xfer_count);

    return xfer_count - nl_count;
}
```

- With terminal framing, any NL character requires a preceding CR characters. So we must count the number of NL characters in the output string to obtain enough space for the corresponding CR characters.
- Note that by **not** incrementing `src_count`, we force the NL character to be transferred from the caller supplied buffer twice. However, this scheme works should the I/O queue become full after inserting the CR character, *i.e.* in the loop body, at least one character can be inserted, and the `continue` forces a re-evaluation of the loop control to insure there is a place for the NL.



The following version of the framer is a rewrite and is intended to implement the same logic. The advantage it that it does not have to scan the input twice as the previous implementation did. However, it does not work correctly. It drops characters and occasionally outputs a NL with no preceding CR character. At this point it remains a mystery and we move on.

```
<<uart control functions>>=
fn terminalTxFramer(
    udev: *UartDevice,
    src: []const u8,
) usize {
    if (src.len == 0) return 0;

    const control = &udev.tx_control;
    const queue = &control.queue;

    const resv_len = src.len + 2; // guess extra space for CR characters
    const dest = queue.reserve(resv_len); // may return less than the requested.

    var xfer_count: usize = 0;
    var dest_index: usize = 0;

    for (src) |*src_slot| {
        const src_char = v7m.ldrbt(src_slot);

        if (src_char == '\n') {
            if (control.set_aside[0] == '\r') { // NL after having seen CR
                control.set_aside[0] = src_char;
                dest[dest_index] = src_char;
                xfer_count += 1;
                dest_index += 1;
                if (dest_index >= dest.len) break;
            } else { // NL without having seen CR -- try to insert both
                control.set_aside[0] = '\r';
                dest[dest_index] = '\r';
                dest_index += 1; // don't count the CR in the xfer_count, it was "added"
                if (dest_index >= dest.len) break;

                dest[dest_index] = '\n';
                xfer_count += 1;
                dest_index += 1;
                if (dest_index >= dest.len) break;
            }
        } else { // ordinary character, not a line terminator
    }
```

```

        control.set_aside[0] = src_char;
        dest[dest_index] = src_char;
        xfer_count += 1;
        dest_index += 1;
        if (dest_index >= dest.len) break;
    }
}

queue.commit(xfer_count);

return xfer_count;
}

```

Similar to the TX framer, the terminal Rx framer uses the `set_aside` buffer to handle CR/LF sequences. The logic is to replace all CR characters with LF and to drop all LF characters immediately preceded by a CR.

```

<<uart control functions>>=
fn terminalRxFramer(
    udev: *UartDevice,
    dest: []u8,
) struct {usize, RxCharError} {
    const control = &udev.rx_control;
    const queue = &control.queue;

    const src = queue.getContiguousBlock();
    const xfer_count = @min(src.len, dest.len);

    var dest_count: usize = 0;
    var src_count: usize = 0;
    var recv_status = rx_status_success;

    while (isRxSuccess(recv_status) and src_count < xfer_count) {
        const rx_char_with_err = src[src_count];
        var rx_char = rx_char_with_err.data;
        recv_status = rx_char_with_err.data_err;

        if (rx_char == '\n' and control.set_aside[0] == '\r') {
            control.set_aside[0] = '\n';
            src_count += 1;
            continue;
        } else if (rx_char == '\r') {
            control.set_aside[0] = '\r';
            rx_char = '\n';
        } else {
            control.set_aside[0] = rx_char;
        }

        v7m.strbt(rx_char, &dest[dest_count]);
        dest_count += 1;
        src_count += 1;
    }
}

```

```

queue.decommitBlock(src_count);
return { dest_count, recv_status };
}

```

Interactive Framing Transformation

The interactive framing functions are intended for using the UART to directly interact with a human. Human console interactions are record oriented, but need to support echoing of input characters and line editing. It operates by holding a line in reserve in the buffer so that backspace and other line editing can occur. Once the line is terminated with a CR, it is made available to the background. There is nothing special about interactive output: it is the same as terminal output.

```

<<uart control functions>>=
inline fn interactiveTxFramer(
    udev: *UartDevice,
    src: []const u8,
) usize {
    return terminalTxFramer(udev, src);
}

```

For interactive receive framing, an entire line of input is held in the RX buffer so that line editing commands can be performed. This framer uses the `cursor` member of the receive controls to store the position in the RX buffer where it has last examined the input. This design only supports backspace (and delete) characters to remove the last input character and the `^U` character (kill) to discard the entire frame. Otherwise, frame detection is as expected, with CR (or a CR/LF sequence) ending the frame. Received characters are also echoed. This includes the necessary echo sequence to handle backspace and line kill, *i.e.* a sequence of backspaces and spaces are transmitted to erase the character from the interactive terminal display. Note that since the RX framer only executes when there is an ongoing attempt to read UART input, echoing typed characters only occurs when background processing is executing a receive request.

```

<<uart control functions>>=
// There are two states of the framer:
// 1. filling -- characters are brought in and held in the Rx queue to allow for
//      line editing. When the end of frame is detected, a transition is made to
//      emptying.
// 2. emptying -- accumulated characters are handed to the caller until the frame
//      is read.
// Current state is held in set_aside[1].
fn interactiveRxFramer(
    udev: *UartDevice,
    dest: []u8,
) struct {usize, RxCharError} {
    const FrameState = enum(u8) {
        filling,
        emptying,
    };
    const bs_echo = [_]u8{'\x08', ' ', '\x08'};
    const crlf_echo = [_]u8{'\r', '\n'};

    const control = &udev.rx_control;
}

```

```

const queue = &control.queue;

var xfer_count: usize = 0;
var recv_status = rx_status_success;

var current_state: FrameState = @enumFromInt(control.set_aside[1]);
sm: switch (current_state) {
    .filling => {
        current_state = .filling;
        var src = queue.getContiguousBlock();
        if (control.cursor < src.len) {
            const rx_char_with_err = src[control.cursor];
            const rx_char = rx_char_with_err.data;
            recv_status = rx_char_with_err.data_err;

            if (!isRxSuccess(recv_status)) {
                continue :sm .emptying;
            } else if (rx_char == '\r') {
                udev.uart_control.putTxBuffer(&crlf_echo);
                control.set_aside[0] = '\r';
                src[control.cursor].data = '\n';
                control.cursor += 1;
                continue :sm .emptying;
            } else if (rx_char == '\n' and control.set_aside[0] == '\r') {
                control.set_aside[0] = '\n';
                mem.copyForwards(
                    RxChar,
                    src[control.cursor .. src.len - 1],
                    src[control.cursor + 1 .. src.len],
                );
            } else if (rx_char == '\x08' or rx_char == '\x7f') { // BS or DEL
                if (control.cursor == 0) {
                    queue.decommitBlock(1);
                    src = queue.getContiguousBlock();
                } else {
                    udev.uart_control.putTxBuffer(&bs_echo);
                    control.cursor -= 1;
                    queue.sza -= 2;
                    mem.copyForwards(
                        RxChar,
                        src[control.cursor .. src.len - 2],
                        src[control.cursor + 2 .. src.len],
                    );
                }
                continue :sm .filling;
            } else if (rx_char == '\x15') { // ^U
                var kill_count = control.cursor;
                while (kill_count != 0) : (kill_count -= 1) {
                    udev.uart_control.putTxBuffer(&bs_echo);
                }
                queue.decommitBlock(control.cursor + 1); // +1 to remove ^U character
                control.cursor = 0;
            }
        }
    }
}

```

```

        continue :sm .filling;
    } else {
        udev.uart_control.putTxChar(rx_char);
        control.cursor += 1;
        continue :sm .filling;
    }
} else {
    break :sm;
}
},
.emptying => {
    current_state = .emptying;
    var src = queue.getContiguousBlock();
    xfer_count = @min(control.cursor, dest.len);
    const count = copyOutRxCharsUnchecked(dest[0..xfer_count], src[0..xfer_count]);
    queue.decommitBlock(count);
    if (count >= control.cursor) {
        control.cursor = 0;
        continue :sm .filling;
    } else {
        control.cursor -= count;
        break :sm;
    }
},
control.set_aside[1] = @intFromEnum(current_state);
return { xfer_count, recv_status };
}
}

```

Apollo 3 UART Instantiation

With all the various controls in place, we can create an initialized variable of type, UartDevice, to hold the information required to operate the UART as a logical entity.

The Apollo 3 has two instances of the UART peripheral block.

```

<<uart proxies>>=
var uart_devices = [svc_services.uart.instances]UartDevice{
    .{ // UART 0
        .instance = 0,
        .uart_control = .{
            .uart = apollo3.uart[0],
        },
        .irq_control = .{
            .irq_number = UartController.uart0_irq_number,
        },
        .power_control = .init(.pwr_uart_0, .hcpa),
        .pin_control = .{
            .hdwr_flow_support = false,
            .tx_pin = .{
                .pin = 48,
            }
        }
    }
}

```

```

        .pad_cfg = .{
            .fnc_sel = Gpio.padFunctionSelector(48, .uart0tx) catch unreachable,
        },
        .pin_cfg = .{},
        .alt_pad_cfg = .{},
    },
    .rx_pin = .{
        .pin = 49,
        .pad_cfg = .{
            .fnc_sel = Gpio.padFunctionSelector(49, .uart0rx) catch unreachable,
            .inp_en = 1,
        },
        .pin_cfg = .{},
        .alt_pad_cfg = .{},
    },
    .rts_pin = .{
        .pin = math.maxInt(u8),
        .pad_cfg = .{},
        .pin_cfg = .{},
        .alt_pad_cfg = .{},
    },
    .cts_pin = .{
        .pin = math.maxInt(u8),
        .pad_cfg = .{},
        .pin_cfg = .{},
        .alt_pad_cfg = .{},
    },
},
.framing = .identity,
.tx_control = .{
    .queue = undefined,
    .storage = undefined,
    .set_aside = [_]u8{0} ** 4,
    .cursor = 0,
    .short_write = false,
},
.rx_control = .{
    .queue = undefined,
    .storage = undefined,
    .set_aside = [_]u8{0} ** 4,
    .cursor = 0,
    .short_read = false,
},
.flush_pending = false,
.proxy = undefined,
.closure = 0,
},
.{ // UART 1
    .instance = 1,
    .uart_control = .{
        .uart = apollo3.uart[1],
    },
}

```

```

.irq_control = .{
    .irq_number = UartController.uart1_irq_number,
},
.power_control = .init(.pwr_uart_1, .hcpa),
.pin_control = .{
    .hdwr_flow_support = true,
    .tx_pin = .{
        .pin = 12,
        .pad_cfg = .{
            .fnc_sel = Gpio.padFunctionSelector(12, .uart1tx) catch unreachable,
        },
        .pin_cfg = .{},
        .alt_pad_cfg = .{},
    },
    .rx_pin = .{
        .pin = 25,
        .pad_cfg = .{
            .fnc_sel = Gpio.padFunctionSelector(25, .uart1rx) catch unreachable,
            .inp_en = 1,
        },
        .pin_cfg = .{},
        .alt_pad_cfg = .{},
    },
    .rts_pin = .{
        .pin = 10,
        .pad_cfg = .{
            .fnc_sel = Gpio.padFunctionSelector(10, .ualrts) catch unreachable,
        },
        .pin_cfg = .{},
        .alt_pad_cfg = .{},
    },
    .cts_pin = .{
        .pin = 17,
        .pad_cfg = .{
            .fnc_sel = Gpio.padFunctionSelector(17, .ualcts) catch unreachable,
            .inp_en = 1,
        },
        .pin_cfg = .{},
        .alt_pad_cfg = .{},
    },
},
.framing = .identity,
.tx_control = .{
    .queue = undefined,
    .storage = undefined,
    .set_aside = [_]u8{0} ** 4,
    .cursor = 0,
    .short_write = false,
},
.rx_control = .{
    .queue = undefined,
    .storage = undefined,
}

```

```

.set_aside = [_]u8{0} ** 4,
.cursor = 0,
.short_read = false,
},
.flush_pending = false,
.proxy = undefined,
.closure = 0,
),
};

}

```

We track the allocation of UART devices by the background execution. Although multiple portions of a program can use the same UART, some one part must be responsible for opening and closing the UART device.

```

<<device access>>=
const uart_allocator = DeviceAllocator(svc_services.uart.instances, &.{});

```

UART Services

The interface to the UART device for background processing is provided by five functions:

- Open the UART
- Close the UART
- Write data to transmit
- Flush queued transmit data
- Read received data
- Query status

```

<<system_services: device service definitions>>=
/// UART services
pub const uart = struct {
    <<uart services>>
};

```

```

<<uart services>>=
pub const instances: DevSvcInstance = apollo3.UartController.uart_instance_count;

```

```

<<uart services>>=
pub const UartOperations = DefineDeviceOperations(
    .uart,
    &.{ "open", "close", "transmit", "flush", "receive", "query" },
);

```

```

<<device proxies>>=
/// UART service proxies

```

```

const uart = struct {
    <<uart proxies>>
};

}

```

Open a UART

The background functions which marshal the request follow the same pattern used previously, fill in the parameter structures and make a device realm SVC call.

```

<<uart services>>=
/// The `open` function makes ready the UART given by, `uart`.
/// This function must be invoked before any other operations on the UART.
pub fn open(
    /// The instance number which identifies which UART device is to be opened.
    inst: DevSvcInstance,
    /// The desired baud rate of the UART in units of bits per second.
    /// Note not all possible baud rates can be reproduced in the UART hardware.
    baud_rate: u32,
    /// A boolean to indicate if hardware flow control is to be used.
    hdwr_flow_ctrl: bool,
    /// The `framing` argument selects the type of line framing to be used.
    framing: FramingType,
    /// A pointer to a notification proxy function which is to be invoked during
    /// background processing to handle UART notifications.
    proxy: DevSvcNotifyProxy,
    /// A closure data value which is included in any background notifications.
    closure: usize,
) SvcError!void {
    const input_params = OpenInputParam.marshal(.{
        .baud_rate = baud_rate,
        .hdwr_flow_ctrl = hdwr_flow_ctrl,
        .framing = framing,
        .proxy = proxy,
        .closure = closure,
    });
    const request = UartOperations.makeRequest(.open, inst);
    return devRealmSvcCall(request, @ptrCast(&input_params), null, null);
}

```

An appropriate data structure is needed to transfer input parameters across the SVC interface.

```

<<uart services>>=
pub const OpenInputParam = DefineSvcRequestParam(struct {
    baud_rate: u32,
    hdwr_flow_ctrl: bool,
    framing: FramingType,
    proxy: DevSvcNotifyProxy,
    closure: usize,
});

```

Open UART Proxy

The foreground proxy function for opening the UART is long because there are multiple peripherals involved. It must handle:

- Allocation and configuration of the I/O pins used by the UART.
- Enabling power to the UART from the power controller.
- Configuring the UART itself.
- Initializing the control block data used to manage the UART operations.
- Enabling the UART and its interrupt.

The following function is the foreground proxy for open.

```
<<uart proxies>>=
pub fn open(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    _: ?* anyopaque,
    _: ?* anyopaque,
) SvcResult {
    const inst = req.instance;
    if (inst >= svc_services.uart.instances) return .unknown_instance;
    uart_allocator.allocate(inst) catch return .operation_failed;

    const udev = &uart_devices[inst];

    <<uart open proxy: receive arguments>>
    <<uart open proxy: enable power>>
    <<uart open proxy: configure I/O pins>>
    <<uart open proxy: configure uart>>
    <<uart open proxy: init control block>>
    <<uart open proxy: enable uart>>

    return .success;
}
```

The input arguments must be read in from unprivileged memory. Some validation is necessary since a request for hardware flow control may be made only for a UART that supports it and a proxy function must be specified to handle the buffer notifications.

```
<<uart open proxy: receive arguments>>
const OpenInputParam = svc_services.uart.OpenInputParam;
const open_params, const result = OpenInputParam.receive(input.?);
if (result != .success) return result;

if (open_params.hdwr_flow_ctrl and
    !udev.pin_control.hdwr_flow_support)
{
    return .invalid_param;
}
```

```
udev.framing = open_params.framing;
```

The Apollo 3 SOC has power controls for all the peripherals. To use the UART, it must be powered up and execution flow must wait for the power up status to show.

```
<<uart open proxy: enable power>>=
udev.power_control.power_up();
```

Note also that if hardware flow of control is requested, but no clear to send (CTS) signal is indicated, then ultimately and transmission will force a failure because the Tx FIFO becomes full with no receiver on the other end to assert CTS. This situation would cause a panic.

```
<<uart open proxy: configure I/O pins>>=
udev.pin_control.configureTxRx();
if (open_params.hdwr_flow_ctrl) {
    udev.pin_control.configureHdwrFlowControl();
}
```

Much of the configuration of the UART is fixed by the manner in which the peripheral is operated. The FIFO's are set to trigger at 3/4 full for receiving and 1/4 remaining for transmit.

```
<<uart open proxy: configure uart>>=
udev uart_control.clear();
udev uart_control.configFifos();
udev uart_control.configCharLength();
udev uart_control.configControls(open_params.hdwr_flow_ctrl);
udev uart_control.configBaudRate(open_params.baud_rate) ;
```

The I/O queues for both receiving and transmitting are cleared out. The remainder of the control block members are also initialized.

```
<<uart open proxy: init control block>>=
udev.tx_control.reset();
udev.rx_control.reset();
udev.proxy = open_params.proxy ;
udev.closure = open_params.closure ;
```

Finally, the UART is enabled and its interrupts are configured.

```
<<uart open proxy: enable uart>>=
udev uart_control.enableTxInterrupt();
udev uart_control.enableRxInterrupt();

udev.irq_control.clearPending();
udev.irq_control.enable();

udev uart_control.enable();
```

Close a UART

```
<<uart services>>=
/// The `close` function shuts down the UART given by, `uart`.
pub fn close(
    /// The instance number which identifies which UART device is to be closed.
    inst: DevSvcInstance,
) SvcError!void {
    const request = UartOperations.makeRequest(.close, inst);
    return devRealmSvcCall(request, null, null, null);
}
```

Close UART Proxy

Closing a UART is simpler than opening one. The following function is the foreground proxy for the `close` service.

```
<<uart proxies>>=
pub fn close(
    req: DevSvcRequest,
    _: ?*const anyopaque,
    _: ?* anyopaque,
    _: ?* anyopaque,
) SvcResult {
    const inst = req.instance;
    if (inst >= svc_services.uart.instances) return .invalid_param;
    if (!uart_allocator.isAllocated(inst)) return .success; ①

    const udev = &uart_devices[inst];
    udev.irq_control.disable();
    udev.uart_control.clear();
    udev.pin_control.disable();
    udev.pin_control.free();
    udev.power_control.power_down();
    uart_allocator.free(inst);

    return .success;
}
```

① Closing an unallocated UART is not an error. Closed is closed.

UART Transmit

```
<<uart services>>=
/// The `transmit` function requests that the data pointed to by the `bytes` slice
/// be transmitted on the UART given by, `uart`. The return value of the function
/// is the number of data bytes actually queued for transmission. If the return
/// number of queue bytes is less than the length of the `bytes` slice, then the
/// caller must retry to transmit the remaining bytes. The caller can know when to
/// retry _short_ writes by waiting for the `tx_available` reason of the UART
```

```

/// notification.

pub fn transmit(
    /// The instance number which identifies which UART device on which
    /// the data is transmitted.
    inst: DevSvcInstance,
    /// A pointer to the data to be transmitted.
    bytes: []const u8,
) SvcError!usize {
    const input_params = TransmitInputParam.marshal(.{
        .tx_chars = bytes,
    });
    var output_params: TransmitOutputParam = .initial;

    const request = UartOperations.makeRequest(.transmit, inst);
    try devRealmSvcCall(request, @ptrCast(&input_params), @ptrCast(&output_params), null);

    return output_params.unmarshal().tx_count;
}

```

```

<<uart services>>=
pub const TransmitInputParam = DefineSvcRequestParam(struct {
    tx_chars: []const u8,
});

pub const TransmitOutputParam = DefineSvcRequestParam(struct {
    tx_count: usize,
});

```

UART Transmit Proxy

The following function is the foreground proxy for `transmit`.

```

<<uart proxies>>=
pub fn transmit(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    output: ?* anyopaque,
    _: ?* anyopaque,
) SvcResult {
    const inst = req.instance;
    if (inst >= svc_services.uart.instances) return .invalid_param;
    if (!uart_allocator.isAllocated(inst)) return .operation_failed;

    const TransmitInputParam = svc_services.uart.TransmitInputParam;
    const input_params, const result = TransmitInputParam.receive(input.?);
    if (result != .success) return result;

    const udev = &uart_devices[inst];

    var tx_count: usize = undefined;

```

```

// INTERRUPT SECTION BEGIN
{
    udev.uart_control.disableTxInterrupt();
    defer udev.uart_control.enableTxInterrupt();

    tx_count = udev.txFramer(input_params.tx_chars);
    _ = udev.fillTxFifo();
}
// INTERRUPT SECTION END

udev.tx_control.short_write = tx_count < input_params.tx_chars.len;

const TransmitOutputParam = svc_services.uart.TransmitOutputParam;
return TransmitOutputParam.send({.tx_count = tx_count}, output.?);
}

```

UART Flush

```

<<uart services>>=
/// The `flush` function requests that the background be notified when the
/// TX queue goes empty.
pub fn flush(
    /// The instance number which identifies which UART device which
    /// is to be flushed.
    inst: DevSvcInstance,
) SvcError!void {
    const request = UartOperations.makeRequest(.flush, inst);
    try devRealmSvcCall(request, null, null, null);
}

```

UART Flush Proxy

```

<<uart proxies>>=
pub fn flush(
    req: DevSvcRequest,
    _: ?*const anyopaque,
    _: ?* anyopaque,
    _: ?* anyopaque,
) SvcResult {
    const inst = req.instance;
    if (inst >= svc_services.uart.instances) return .invalid_param;
    if (!uart_allocator.isAllocated(inst)) return .operation_failed;

    const udev = &uart_devices[inst];
    var result: SvcResult = .success;

// INTERRUPT SECTION BEGIN
{
    udev.uart_control.disableTxInterrupt();
    defer udev.uart_control.enableTxInterrupt();
}

```

```

    if (udev.tx_control.queue.getCommittedSize() == 0) {
        while (!udev uart_control.txFifoEmpty()) {
            // empty
        }
        if (!udev.notifyToBackground(.tx_flushed)) {
            result = .operation_failed;
        }
        udev.flush_pending = false;
    } else {
        udev.flush_pending = true;
    }
}
// INTERRUPT SECTION END

return result;
}

```

UART Receive

```

<<uart services>>=
/// The `receive` function attempts to transfer the number of bytes given by
/// `buffer.len` from the receive queue of `uart` and places any bytes read into
/// the memory pointed to by `buffer`. It returns the number of bytes actually
/// placed in `buffer`. If the returned number of bytes is less than `buffer.len`,
/// then the caller must invoke `receive` again to obtain any additional data. The
/// caller can know when to retry _short_ reads by waiting for the `rx_available`
/// reason for UART notification. After returning from the function, the memory
/// pointed to by `status` contains the error status of the read. If the status is
/// `.success`, then all the transferred bytes were received without error. If the
/// status is otherwise, then it indicates the cause of the receive error and the
/// last byte placed in buffer is the one in error. The value of the byte in error
/// is returned, _i.e._ `buffer[count - 1]`, where `count` is the return value of
/// the function, is the byte as it was received by the UART peripheral. Receive
/// requests are ended, possibly short, when the first byte encountered is found to
/// be in error.
pub fn receive(
    /// The instance number that identifies from which UART device data is
    /// to be received.
    inst: DevSvcInstance,
    /// A pointer to the memory where the received data is placed.
    buffer: []u8,
    /// A pointer to a memory object where receive error status is placed.
    status: *RxErrStatus,
) SvcError!usize {
    const input_params = ReceiveInputParam.marshal(.{
        .rx_chars = buffer,
    });
    var output_params: ReceiveOutputParam = .initial;

    const request = UartOperations.makeRequest(.receive, inst);

```

```

try devRealmSvcCall(request, @ptrCast(&input_params), @ptrCast(&output_params), null);

const output = output_params.unmarshal();
status.* = output.status;
return output.rx_count;
}

```

```

<<uart services>>=
pub const ReceiveInputParam = DefineSvcRequestParam(struct {
    rx_chars: []u8
});

pub const ReceiveOutputParam = DefineSvcRequestParam(struct {
    rx_count: usize,
    status: RxErrStatus,
});

```

UART Receive Proxy

The following function is the foreground proxy for receive.

```

<<uart proxies>>=
pub fn receive(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    output: ?* anyopaque,
    _: ?* anyopaque,
) SvcResult {
    const inst = req.instance;
    if (inst >= svc_services.uart.instances) return .invalid_param;
    if (!uart_allocator.isAllocated(inst)) return .operation_failed;

    const ReceiveInputParam = svc_services.uart.ReceiveInputParam;
    const input_params, const result = ReceiveInputParam.receive(input.?);
    if (result != .success) return result;

    const udev = &uart_devices[inst];

    var rx_count: usize = undefined;
    var char_err: RxCharError = .{};

// BEGIN INTERRUPT SECTION
{
    udev.uart_control.disableRxInterrupt();
    defer udev.uart_control.enableRxInterrupt();

    rx_count, char_err = udev.rxFramer(input_params.rx_chars);
}
// END INTERRUPT SECTION

    udev.rx_control.short_read = rx_count < input_params.rx_chars.len;

```

```

const ReceiveOutputParam = svc_services.uart.ReceiveOutputParam;
return ReceiveOutputParam.send(//{
    .rx_count = rx_count,
    .status = mapRxErrors(char_err),
}, output.?);
}

```

```

<<uart proxies>>=
pub fn mapRxErrors(
    char_err: RxCharError,
) svc_services.uart.RxErrStatus {
    if (char_err.fe_err == 1) return .frame_err;
    if (char_err.pe_err == 1) return .parity_err;
    if (char_err.oe_err == 1) return .overrun_err;

    return .success;
}

```

UART Query

```

<<uart services>>=
/// The `query` function returns the number of bytes in the RX and TX queues and
/// the length of the RX and TX buffers for the UART given by, `uart`.
/// The counts are returned as a four element anonymous
/// struct in the order receive count, transmit count, receive buffer length, and
/// transmit buffer length.
pub fn query(
    /// The instance number which identifies the UART device whose RX and TX queue
    /// counts are returned.
    inst: DevSvcInstance,
) SvcError!struct {usize, usize, usize, usize} {
    var output_params: QueryOutputParam = .initial;

    const request = UartOperations.makeRequest(.query, inst);
    try devRealmSvcCall(request, null, @ptrCast(&output_params), null);

    return output_params.unmarshal();
}

```

```

<<uart services>>=
pub const QueryOutputParam = DefineSvcRequestParam(struct {
    rx_count: usize,
    tx_count: usize,
    rx_buf_len: usize,
    tx_buf_len: usize,
});

```

UART Query Proxy

```
<<uart proxies>>=
pub fn query(
    req: DevSvcRequest,
    _: ?*const anyopaque,
    output: ?* anyopaque,
    _: ?* anyopaque,
) SvcResult {
    const inst = req.instance;
    if (inst >= svc_services.uart.instances) return .invalid_param;
    if (!uart_allocator.isAllocated(inst)) return .operation_failed;

    const udev = &uart_devices[inst];

    var rx_count: usize = undefined;
    var tx_count: usize = undefined;

// BEGIN PRIORITY SECTION
{
    const section = v7m.basepri.PrioritySection.enter(system_config
.irq_encoded_priority); ①
    defer section.leave();

    rx_count = udev.rx_control.queue.getCommittedSize();
    tx_count = udev.tx_control.queue.getCommittedSize();
}
// END PRIORITY SECTION

    const QueryOutputParam = svc_services.uart.QueryOutputParam;
    return QueryOutputParam.send(.{
        .rx_count = rx_count,
        .tx_count = tx_count,
        .rx_buf_len = rx_queue_size,
        .tx_buf_len = tx_queue_size,
    }, output.?);
}
```

- ① Both Rx and Tx interrupts need to be disabled to insure accurate counts. So, we just raise execution priority to cover all interrupts.

Dispatching UART Requests

In keeping with the device foreground proxy design, a class level handler is used to dispatch background requests to the appropriate proxy that handles the request.

```
<<svc device classes>>=
uart,
```

```
<<dev svc service request prongs>>=
.uart => &uart.devClassHandler,
```

Finally, the UART class handler performs the dispatch to the operation functions of requests.

```
<<uart proxies>>
/// The `devClassHandler` function dispatches UART operations based
/// on the value of the `req` argument. This function is invoked as a
/// class level dispatcher for the UART device.
fn devClassHandler(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    output: ?*anyopaque,
    err: ?*anyopaque,
) SvcResult {
    const Operation = svc_services.uart.UartOperations.Operation;
    const uart_operation: Operation = @enumFromInt(req.operation);
    const proxy = switch (uart_operation) {
        .open => &open,
        .close => &close,
        .transmit => &transmit,
        .flush => &flush,
        .receive => &receive,
        .query => &query,
    };
    return proxy(req, input, output, err);
}
```

Servicing the UART

This section discusses the foreground processing required to service the UART. Because there is no DMA associated with the UART on the Apollo 3, the processor executes code to move data to and from the UART FIFO's. The necessary code starts with the IRQ handler.

UART IRQ Handling

Each UART peripheral is tied to a single IRQ in the processor core. So, the IRQ handler must deal with both receiving and transmitting. All the common code has been factored into a single function, so the IRQ handlers themselves are simple one-liners.

```
<<uart control functions>>
export fn uart0IrqHandler() void {
    irqHandler(0);
}

export fn uart1IrqHandler() void {
    irqHandler(1);
}
```

This design uses only three interrupt sources in the UART. The receive and transmit FIFO interrupts indicate when the FIFO needs service. The receive timeout interrupt indicates that there are bytes in the receive FIFO and there has been a time gap where no additional byte has arrived. Note neither interrupts for error conditions nor for modem control lines status changes are enabled. Since

each received character carries with it 4 bits of error information and since the error information is tracked through the RX queue, errors are processed as each byte is read out of the RX queue. Since only simplified RTS/CTS hardware flow control is supported (and not complete modem control which might require RI, DCD and DSR status), the peripheral itself handles the necessary changes in the I/O pins associated with the flow control.

```
<<uart control functions>>=
fn irqHandler(
    inst: DevSvcInstance,
) void {
    const udev = &uart_devices[inst];
    const ureg = udev.uart_control.ureg;
    const irq_sources = ureg.mis.read();

    if (irq_sources.rx == 1 or irq_sources.rt == 1) udev.receiveRxData();
    if (irq_sources.tx == 1) udev.transmitTxData();

    ureg.iec.write(irq_sources);
}
```

The processing of interrupts is simplified somewhat by the fact that the peripheral has a masked interrupt status which indicates directly which of the enabled sources caused the interrupt. Note that the RX FIFO interrupt and RX timeout interrupt sources execute the same code, *i.e.* receive bytes from the RX FIFO into the RX queue.

UART Transmit Operations

This section shows the operations used for transmitting. There is one function used by the IRQ handler for the transmit side. This function is the core of what must happen to transfer bytes from the transmit buffering scheme to the UART transmit FIFO. The transmit FIFO is filled to capacity as long as there is sufficient data in the buffer.

```
<<uart control functions>>=
/// The `transmitTxData` function reads character data from the Tx queue and writes
/// it in the TX FIFO in the device. Data is transmitted until either the FIFO is
/// full or there are no remaining characters in the TX queue.
fn transmitTxData(
    /// A pointer to a UART control block corresponding to the UART on which
    /// characters are to be transmitted.
    udev: *UartDevice,
) void {
    const tx_count = udev.fillTxFifo();
    if (udev.tx_control.short_write and
        tx_count != 0 and
        udev.notifyToBackground(.tx_available))
    {
        udev.tx_control.short_write = false;
    }
}
```

There are only two steps:

- Fill the TX FIFO from the TX queue, recording the number of bytes placed in the FIFO.
- Queue a background notification based on:
 - Past history as to whether the last transmit request was fulfilled.
 - The number of bytes just placed in the TX FIFO.

Note that the logic allows for the background notification to fail, *i.e.* there was no room in the background notification queue. In that case, the `tx_short_write` status is maintained as `true` so another attempt at the notification can be made later.

The semantics of the notifications are discussed [below](#).

Filling the TX FIFO is a simple loop, but is complicated by the need to check if we have a pending flush that needs to be notified.

```
<<uart control functions>>=
/// The `fillTxFifo` function copies bytes from the TX queue into the
/// transmitter FIFO of the UART. Copied bytes are then elided from
/// Tx queue.
fn fillTxFifo(
    udev: *UartDevice,
) usize {
    const queue = &udev.tx_control.queue;

    const tx_chars = queue.getContiguousBlock();
    var tx_count: usize = 0;
    while (tx_count < tx_chars.len and !udev.uart_control.tx_fifoFull()) : (tx_count += 1) {
        udev.uart_control.loadTxChar(tx_chars[tx_count]);
    }
    queue.decommitBlock(tx_count);

    // If there is no more data in the TX queue and there is a pending request
    // for flush notification, then we poll until the transmitter FIFO is empty
    // to ensure all the characters go out. After all the characters are transmitted,
    // the flush is notified. Note this makes flushing the UART transmitter
    // somewhat expensive, especially at lower baud rates. So, don't flush unnecessarily.
    if (queue.getCommittedSize() == 0 and udev.flush_pending) {
        while (!udev.uart_control.tx_fifoEmpty()) {
            // empty
        }
        if (udev.notifyToBackground(.tx_flushed)) udev.flush_pending = false;
    }

    return tx_count;
}
```

UART Receive Operations

One wishes that receiving was just like transmitting only with some conceptual flow switched in the opposite direction. Unfortunately, that is almost never the case. For UART communications, receiving is more complicated because:

- The receiver can detect errors in the received data, e.g. framing errors. The transmit side simply dumps bits onto a wire and hopes for the best.
- Receiving means data is being *pushed* at you and it is necessary to find the space to put it. Hardware flow of control can help this issue, but in its absence, the only option is to supply some buffering for incoming data and be prepared for overflow if the transmitting peer is inconsistent.
- The receiver has an internal FIFO which holds 32 bytes of input before it overflows.
- The UART has a built-in timeout that is triggered when the receive FIFO contains data and 32-bit times have elapsed. This is useful for handling received characters where there have not been a sufficient number to trigger the FIFO threshold interrupt. As shown in the IRQ handler, the receive timeout is handled in the same manner as the receive FIFO interrupt, i.e. move bytes from the receive FIFO and into the RX queue.

Despite these differences from the transmit side, reception is conceived of as requesting characters from the RX queue which is filled by the IRQ handler from the UART FIFO.

Receive UART Operations

```
<<uart control functions>>=
/// The `receiveRxData` function reads character data from the UART FIFO and places
/// it in the RX queue. Data is received until either the FIFO is empty or there
/// is no remaining space in the RX queue. If in the process of receiving data a
/// serial line _break_ condition is detected, then a notification for the break is
/// placed in the Background Notification Queue. If a character is received in
/// error, the received data is placed in the RX queue, and the error is handled
/// when the data is read out of the RX queue.

fn receiveRxData(
    /// A pointer to a UART control block corresponding to the UART where
    /// characters are to be received.
    udev: *UartDevice,
) void {
    const queue = &udev.rx_control.queue;
    const rx_buf = queue.reserve(32); // there are 32 bytes max in the Rx FIFO

    var rx_count: usize = 0;
    while (rx_count < rx_buf.len and !udev.uart_control.rx_fifoEmpty()) : (rx_count += 1) {
        const rx_char = udev.uart_control.unloadRxChar();
        if (rx_char.data_err.be_err == 1) {
            _ = udev.notifyToBackground(.line_break);
            break;
        // When doing interactive framing, we must leave one slot in the buffer open
        // to handle a line editing character (e.g. CR, or BS). So if there is only
        // one slot left, we insist that the input character be a line editing
character.
        } else if (rx_buf.len - rx_count == 1 and
            udev.framing == .interactive and
            mem.count(u8, "\r\n\x08\x7f\x15", &{rx_char.data}) == 0)
        {
            break;
        } else {
            rx_buf[rx_count] = rx_char;
        }
    }
}
```

```

        }
    }

queue.commit(rx_count);

if (udev.rx_control.short_read and
    rx_count != 0 and
    udev.notifyToBackground(.rx_available))
{
    udev.rx_control.short_read = false;
}
}

```

UART Notifications

The UART device issues notifications for three reasons.

```

<<uart services>>=
pub const NotifyReason = enum {
    /// The `tx_available` notification is issued to the background when the last
    /// transmit request filled the TX queue and there is now more space available
    /// in the queue. Transmit requests which are not completely transferred to
    /// the TX queue receive a TX available notification to know when to transmit
    /// additional data.
    tx_available,
    /// The `tx_flushed` notification is issued to the background in response to
    /// flush operation. The transmitter is considered flushed when the TX queue
    /// becomes empty after a preceding flush operation request.
    tx_flushed,
    /// The `rx_available` notification is issued to the background when the
    /// last receive request was not fulfilled because the RX queue was emptied.
    /// The notification is an indication to background processing that additional
    /// bytes have been received.
    rx_available,
    /// The `line_break` notification is issued when the UART peripheral detects a
    /// *break* condition on the serial line. This notification is queued to the
    /// background as out of band data, i.e. the notification is sent when the
    /// break is read off of the UART peripheral and is not sent through the RX
    /// queue.
    line_break,
};

```

As mentioned previously, each received character has an associated error status which is tracked along with the character in the RX queue. Error status is returned when the byte received in error is transferred out of the RX queue. The value of the received byte is passed to the background. Reading a byte in error ends the request and passes back the error status. The error status is represented as a bit encoded enumeration that is part of the notification for the UART device.

```

<<uart services>>=
pub const RxErrStatus = enum {
    success,
    frame_err,

```

```

    parity_err,
    overrun_err,
};
```

The notification data structure for the UART also includes the number of bytes in both the RX and TX queues.

```

<<dev notification specs>>=
.{ "uart", struct {
    reason: uart.NotifyReason,
    tx_count: usize,
    rx_count: usize,
},
},
```

All the above considerations are factored into a function that queues a UART notification to the Background Notification Queue.

```

<<uart control functions>>=
/// The `notifyToBackground` function queues a UART notification to the background
/// notification queue. The return value is `true` if the notification was
/// successfully queued and `false` otherwise. _N.B._ failure to queue a UART
/// notification to the Background Notification Queue is _not_ considered a
/// "panic" condition. This recognizes the _interactive_ nature of
/// computer-to-computer communications. Missing a notification gives the two
/// communicating peers a chance to catch up with each other. Furthermore, an
/// overly exuberant transmitter must _not_ to be able to force a panic condition
/// on the system.
const NotifyReason = svc_services.uart.NotifyReason;

fn notifyToBackground(
    /// A pointer to a UART control block.
    udev: *const UartDevice,
    /// The reason for the UART notification.
    reason: NotifyReason,
) bool {
//BEGIN PRIORITY SECTION
    const section = v7m.basepri.PrioritySection.enter(system_config.irq_encoded_priority);
①    defer section.leave();

    const result: SvcResult, const notification: *DevNotification =
        dev_notify_queue.reserveNotification();
    if (result != .success) return false;

    notification.* = DevNotification.init(
        .uart, // operation
        udev.instance, // instance,
        udev.proxy, // proxy
        udev.closure, // closure
        .{
```

```

    .uart = .{
        .reason = reason,
        .tx_count = udev.tx_control.queue.getCommittedSize(),
        .rx_count = udev.rx_control.queue.getCommittedSize(),
    }
}, // UART specific notification information
);

dev_notify_queue.commitNotification();
return true;
//END PRIORITY SECTION
}

```

- ① The `notifyToBackground` function is invoked both from SVC proxies and from IRQ handlers.

Console I/O

As discussed [previously](#), UART 0 in the Sparkfun MicroMod design has the side effect of causing a system reset when the port is first connected. That leaves us with UART 1 as the only UART that does not cause a dramatic side effect. UART 1 also supports hardware flow control which is needed to run at high baud rates with minimal buffering.

In this section, we show the implementation of a “system” console and a “log” console. We assign UART0 to the system console and UART1 to the logging console. Both behave the same with the following exceptions:

- When first connected to, UART0 causes a system reset.
- UART0 has no hardware flow of control.

The main task here is to implement the `std.Io.Reader` and `std.Io.Writer` interfaces on top of the `uart` device. These interfaces appeared in Zig version 0.15.1 and are part of the newly extended I/O concepts as embodied by `std.Io` of the standard library.

Design Overview

The primary design goal is integrate console I/O via a UART into the reader / writer scheme of the Zig standard library. These interfaces are somewhat complicated and contain both buffering in the interface itself as well a virtual function table to handle actual device interactions. The overall structure is to define a specific `Console` entity that can satisfy the I/O interface of the Zig standard library. A `Console` instance contains UART specific fields and functions. The `reader` and `writer` functions return `Console.Reader` and `Console.Writer` instances that are to be stored in a variable. The `reader` and `writer` functions can be thought of as variable initialization functions. The `Reader` and `Writer` typed variables contain a pointer to the `Console` instance and two fields called, `ior_interface` and `iow_interface` which conform to the `std.Io.Reader` and `std.Io.Writer` interfaces, respectively.

A primary complication is to deal with the asynchronous nature of the UART device. Like all the logical devices we have discussed, UART interactions are asynchronous. We embed synchronization with the UART device in the implementation when no characters are received from the UART. The number of bytes transferred by the reading and writing functions on the interfaces can be “short” of the requested amount, but no received or transmitted count is returned from the I/O interface as zero. If no data is available to receive or no space is available to send data, the device operations synchronize with the UART peripheral code to be notified when either more characters can be

received or more characters can be sent.

Since the UART peripheral control code detects a serial *line break* condition and sends a notification for that condition, the line break is mapped to the `StreamError.EndOfStream` error return.

Defining a Console

A Console serves as a container for the UART device specific data and the inner `Console.Reader` and `Console.Writer` structures.

```
<<system_services: device service definitions>>=
/// A Console is a structure suitable for use for input and output
/// that conforms to the Zig standard library (as of 0.15.1) interfaces
/// for I/O. There are two UART channels that can be used as a Console.
/// Since UARTs are often connected to terminals, the UART peripheral
/// device provides three types of UART framing. The `framing` argument may be one of: `.identity`, `.terminal` or `.interactive`.
/// The `framing` controls whether line editing is available and whether
/// I/O transfers should be flushed when a newline is encountered.
/// Initialized instances of Console that correspond to using a UART
/// channel as a "system" console with `.interactive` framing and as a
/// "logging" console with `.terminal` framing are provided. Typical
/// usage is for the system console to be dedicated to interactive I/O
/// with a human and for the logging console to be used for output for
/// std.log operations.
pub const Console = struct {
    uart_instance: DevSvcInstance,
    framing: uart.FramingType,
    rx_notified: bool = false, // synchronization fields
    break_notified: bool = false,
    tx_notified: bool = false,
    flush_notified: bool = false,
    config: Io.tty.Config = .escape_codes, // Assume we are connected to a tty.

    pub var sys_console: Console = .{
        .uart_instance = 0,
        .framing = .interactive,
    };
    pub var log_console: Console = .{
        .uart_instance = 1,
        .framing = .terminal,
    };

    const default_console_baud_rate: u32 = 921_600;
    const Io = std.Io;

    <<console services>>
};
```

Most applications will want to simply set up the two available consoles with little fuss and early in the program execution. Note however that active UARTs consume power and power-conscious applications may not wish to open any consoles or to open and close them as needed.

```

<<console services>>=
/// The `openAllConsoles` function opens logical consoles for the available
/// UART device channels. The consoles are opened in their default configuration
/// where UART 0 is used for interactive I/O with a human and UART 1 is configured
/// for terminal interactions with presumably ASCII characters. Any failure
/// to open either console creates a panic condition.
pub fn openAllConsoles() void {
    sys_console.open(null, false) catch @panic("failed to open system console");
    log_console.open(null, true) catch @panic("failed to open logging console");
}

```

```

<<console services>>=
/// The `open` function opens the UART device associated with the
/// `console` argument. Typically, the `console` argument is one of the
/// two provided by the `Console` namespace. The UART is configured
/// according to the given `baud_rate`, `hdwr_flow_control`, and
/// `framing` arguments. If `baud_rate` is given as `null`, the it is
/// set to 921,600 baud, which is the highest rate that can be achieved
/// by the UART device. The UART device is configured to the usual 8
/// bits per character, one stop bit, and no parity.
pub fn open(
    console: *Console,
    baud_rate: ?u32,
    hdwr_flow_control: bool,
) !void {
    try uart.open(
        console.uart_instance,
        baud_rate orelse default_console_baud_rate,
        hdwr_flow_control,
        console.framing,
        Console.consoleNotifier,
        @intFromPtr(console),
    );
}

```

```

<<console services>>=
/// The `close` function closes the underlying UART for the `console`.
/// Note the buffers internal to the UART peripheral device are
/// flushed and the function synchronized with the UART device code to
/// ensure all characters buffered by the peripheral device have been
/// transmitted. Note, closing does not flush any console writer data
/// that has not already been transferred to the UART control code. This
/// implies that the usual sequence is to flush any IO.Writer interface
/// first before closing the console itself.
pub fn close(
    console: *Console,
) void {
    console.flushSync();
    uart.close(console.uart_instance) catch {};
}

```

Console Reader

Each Console has a `Console.Reader` structure that holds the context information that enables using the Console for input.

```
<<console services>>
/// The `Reader` structure is defined internally to the `Console`
/// namespace and contains a `std.Io.Reader` field that conforms to the
/// Zig standard library conventions. User code should assign the `Reader`
/// value, as returned from the `reader` function, into a variable.
pub const Reader = struct {
    console: *Console,
    rx_status: uart.RxErrStatus = .success,
    ior_interface: Io.Reader,

    const StreamError = Io.Reader.StreamError;

    <<console reader interface>>
};
```

```
<<console services>>
/// The `reader` function returns a `Console.Reader` value that holds the
/// data required to use `console` for input. Typically, a `buffer`
/// slice is provided as a space to hold transient data received from
/// the UART device but not yet taken by user code. Note that the UART
/// device control code contains an internal buffer that is used for
/// framing typical terminal interactions, e.g. to allow interactive
/// line editing of input.
pub fn reader(
    console: *Console,
    buffer: []u8,
) Reader {
    return .{
        .console = console,
        .ior_interface = Reader.initInterface(buffer),
    };
}
```

The `std.Io.Reader` interface in the Console provides a `vtable` field with only the `stream` field set to a non-default value. It is the `stream` function that does the heavy lifting to obtain input. The defaults provided for the other interface functions suffice for this usage.

```
<<console reader interface>>
fn initInterface(
    buffer: []u8,
) Io.Reader {
    return .{
        .vtable = &.{},
        .stream = stream,
    },
    .buffer = buffer,
```

```

    .seek = 0,
    .end = 0,
};

}

```

An explicit function is provided to obtain a reference to the `std.Io.Reader` interface embedded in the `Console.Reader`. Its use is encouraged.

```

<<console reader interface>>=
/// The `interface` function returns a reference to the `std.Io.Reader` field
/// that is contained in a `Console.Reader` structure. User code is encouraged to
/// use this function to obtain a reference to the `Io.Reader` interface.
/// N.B. it is important not to make a copy of the of the `Io.Reader`
/// value. Since the methods of `Io.Reader` use the `@fieldParentPtr` function,
/// all usage of the `Io.Reader` must be by pointer reference to the
/// value of the `Io.Reader` that is contained within a `Console.Reader` value.
pub fn interface(
    r: *Reader,
) *Io.Reader {
    return &r.iор_interface;
}

```

The virtual function contained in the `stream` field of the `Io.Reader` interface performs the interactions with the UART to obtain data.

```

<<console reader interface>>=
/// The `stream` function performs the requested transfer from
/// `io_r` to `io_w` for the number of bytes given by `limit`. This
/// function conforms to the `std.Io.Reader` VTable interface. See the
/// documentation of `std.Io.Reader` for a description of the interface
/// and requirements of this function.
fn stream(
    io_r: *Io.Reader,
    io_w: *Io.Writer,
    limit: Io.Limit,
) StreamError!usize {
    const r: *Reader = @alignCast(@fieldParentPtr("ior_interface", io_r));

    const dest = limit.slice(try io_w.writableSliceGreedy(1));
    var recv_count: usize = 0;

    while (true) : (r.console.rxSync()) { ①
        if (r.console.break_notified) { ②
            r.console.break_notified = false;
            return StreamError.EndOfStream;
        }
        const rx_status = &r.rx_status;
        rx_status.* = .success;
        recv_count = uart.receive(r.console.uart_instance, dest, rx_status)
            catch return StreamError.ReadFailed;
        if (recv_count == 0) continue;
    }
}

```

```

        if (rx_status.* != .success) return StreamError.ReadFailed;
        io_w.advance(recv_count);

        break;
    }

    return recv_count;
}

```

- ➊ If no characters are received, we synchronize with the UART peripheral code to await input. The peripheral code sends a rx_available notification after characters arrive and the previous read was "short", i.e. less was returned than requested. Note this is not a blocking call. The synchronization happens in conjunction with the "run wait" event loop so that other notifications are dispatched and the system continues to execute normally.
- ➋ The line break condition is considered transitory. It is possible to continue reading from the console after the EndOfStream error is returned without any actions to "clear" the condition.

Console Writer

Each Console has a `Console.Writer` structure that holds the context information that enables using the `Console` for output.

```

<<console services>>=
/// The `Writer` structure is defined internally to the `Console`
/// namespace and contains a `std.Io.Writer` field that conforms to the
/// Zig standard library conventions. User code should place a `Writer`
/// value, as returned from the `writer` function, into a variable.
pub const Writer = struct {
    console: *Console,
    iow_interface: Io.Writer,

    const Error = Io.Writer.Error;

    <<console writer interface>>
};

```

```

<<console services>>=
/// The `writer` function returns a `Console.Writer` value that holds the
/// data required to use `console` for output. Typically, a `buffer`
/// slice is provided as a space to hold transient data buffer for
/// the UART device but not yet flushed by user code. Note that the UART
/// device control code contains an internal buffer that is used for
/// offload the number of interactions required for transmitting data.
pub fn writer(
    console: *Console,
    buffer: []u8,
) Writer {
    return .{
        .console = console,
        .iow_interface = Writer.initInterface(buffer),
    };
}

```

```
}
```

```
<<console writer interface>>=
fn initInterface(
    buffer: []u8,
) Io.Writer {
    return .{
        .vtable = &.{
            .drain = drain,
        },
        .buffer = buffer,
    };
}
```

```
<<console writer interface>>=
/// The `interface` function returns a reference to the `std.Io.Writer` field
/// that is contained in a `Console.Writer` structure. User code is encouraged to
/// use this function to obtain the reference to the `Io.Writer` interface.
/// N.B. it is important not to make a copy of the of the `Io.Writer`
/// value. Since the methods of `Io.Writer` use the `@fieldParentPtr` function,
/// all usage of the `Io.Writer` must be by pointer reference to the
/// value of the `Io.Writer` that is contained within a `Console.Writer` value.
pub fn interface(
    w: *Writer,
) *Io.Writer {
    return &w.iow_interface;
}
```

The requirements on the `Io.Writer` `drain` function are somewhat complex and oriented to “scattered” write operations. The sequence is to write buffered data first, write all but the last buffer of data, and then write the last data buffer a given number of times. In the case where there is no available space in the UART peripheral control code output buffer, the `drain` function synchronizes with the UART to wait for space to become available.

```
<<console writer interface>>=
/// The `drain` function outputs the data contained in the `data`
/// argument to the `Io.Writer` given by the `io_w` argument.
/// This function conforms to the `std.Io.Writer` VTable interface.
/// See the documentation of `std.Io.Writer` for a description of the
/// interface and requirements of this function. Note this implementation
/// follows the pattern in std.fs.File.zig for Windows file writing.
fn drain(
    io_w: *Io.Writer,
    data: []const []const u8,
    splat: usize,
) Error!usize {
    std.debug.assert(data.len != 0);

    const w: *Writer = @alignCast(@fieldParentPtr("iow_interface", io_w));
```

```

var xmit_total: usize = 0;
const buffered = io_w.buffered();
if (buffered.len != 0) {
    const xmit_count = try transmitWithSync(w.console, buffered);
    if (io_w.consume(xmit_count) == 0) return 0;
}

for (data[0 .. data.len - 1]) |buf| {
    if (buf.len == 0) continue;
    const xmit_count = try transmitWithSync(w.console, buf);
    xmit_total += io_w.consume(xmit_count);
}

const pattern = data[data.len - 1];
if (pattern.len != 0) {
    var splat_counter = splat;
    while (splat_counter != 0) : (splat_counter -= 1) {
        const xmit_count = try transmitWithSync(w.console, pattern);
        xmit_total += io_w.consume(xmit_count);
    }
}

return xmit_total;
}

```

```

<<console writer interface>>=
/// The `transmitWithSync` function attempts to transit the data in `buffer`
/// to the UART associated with `console`. If no data was transmitted
/// on the first attempt, then the function synchronizes with the UART.
/// After the synchronization, there will be some space in the UART
/// peripheral device buffer and another transmit attempt is made.
fn transmitWithSync(
    console: *Console,
    buffer: []const u8,
) Error!usize {
    var xmit_count = uart.transmit(console.uart_instance, buffer) catch
        return Error.WriteFailed;
    if (xmit_count == 0) {
        console.txSync();
        xmit_count = uart.transmit(console.uart_instance, buffer) catch
            return Error.WriteFailed;
    }
    return xmit_count;
}

```

Console I/O Background Notification Proxy

The UART peripheral device control code issues notifications when semantically significant events occur. These notifications form the basis of the synchronization between the I/O code and the UART device. When the UART is opened for use as a console, the `consoleNotifier` is used as a callback function with pointer to a `Console` as the notification closure value. This notifier is used by the

`Io.Reader` and `Io.Writer` virtual functions to handle cases where no data was read or written, a line break was detected or the transmitter buffer needs to be flushed.

```
<<console services>>=
fn consoleNotifier(
    notification: *const anyopaque,
) void {
    const notify: *const DevNotification = @ptrCast(@alignCast(notification));
    const console: *Console = @ptrFromInt(notify.closure);

    switch (notify.params uart.reason) {
        .tx_available => console.tx_notified = true,
        .tx_flushed => console.flush_notified = true,
        .rx_available => console.rx_notified = true,
        .line_break => {
            console.break_notified = true;
            console.rx_notified = true;
        },
    }
}
```

Separate functions are provided each UART notification.

```
<<console services>>=
pub fn txSync(
    console: *Console,
) void {
    console.tx_notified = false;
    exec_control.runWait(&console.tx_notified) catch unreachable;
}
```

```
<<console services>>=
pub fn flushSync(
    console: *Console,
) void {
    uart.flush(console.uart_instance) catch return;
    console.flush_notified = false;
    exec_control.runWait(&console.flush_notified) catch unreachable;
}
```

```
<<console services>>=
pub fn rxSync(
    console: *Console,
) void {
    console.rx_notified = false;
    console.break_notified = false;
    exec_control.runWait(&console.rx_notified) catch unreachable;
}
```

With the use of the synchronization functions, what appears as a synchronous blocking I/O operations is in fact asynchronous and the remainder of the system execution proceeds as expected because the event loop continues to dispatch device notifications.

Logging to the Console

The Zig standard library provides a logging module. That module allows an application to supply its own log output function. In this section, we show code to integrate the Console I/O from the last chapter to the Zig logging available from the standard library.

```
<<system_services: device service definitions>>=
/// The `logToConsole` function is a suitable replacement for the default
/// logging function of the `std.log` module.
pub fn logToConsole(
    comptime message_level: std.log.Level,
    comptime scope: @Type(.enum_literal),
    comptime format: []const u8,
    args: anytype,
) void {
    var log_buf: [256]u8 = undefined;
    var log_writer = Console.log_console.writer(&log_buf);
    const log_out = log_writer.interface();

    const message_color: std.Io.tty.Color = switch (message_level) { ❶
        .err => .bright_red,
        .warn => .bright_yellow,
        .info => .bright_green,
        .debug => .reset,
    };
    Console.log_console.config.setColor(log_out, message_color) catch {};
    log_out.writeAll(comptime message_level.asText()) catch {};
    Console.log_console.config.setColor(log_out, .reset) catch {};

    const scope_text = if (scope == .default) ":" else "(" ++ tagName(scope) ++ ")";
    log_out.print(scope_text ++ format ++ "\n", args) catch {};

    log_out.flush() catch {};
}
```

❶ Coloring the output based on level is as fancy as we get.

Privileged Transmission

There are times during privileged execution when UART output needs to be generated. For example, the `panic` function should print the panic message if the UART is being used for console output. Privileged execution cannot invoke the `transmit` service since that would cause a Hard Fault. This section presents a “back door” for privileged code to get immediate access to UART output. Since output by privileged code is usually on the path to a system reset, these functions wait for all the output to be transmitted.

```
<<device proxies>>=
```

```

/// Privileged write service
const PrivConsole = struct {
    uart_instance: DevSvcInstance,
    config: std.Io.tty.Config = .{escape_codes, ...};

    pub fn init() PrivConsole {
        return .{
            .uart_instance = svc_services.Console.log_console.uart_instance,
        };
    }

    pub fn writer(
        console: *PrivConsole,
        buffer: []u8,
    ) Writer {
        return .init(console, buffer);
    }
};

pub const Writer = struct {
    console: *PrivConsole,
    iow_interface: Io.Writer,

    const Error = Io.Writer.Error;

    pub fn interface(
        w: *Writer,
    ) *Io.Writer {
        return &w.iow_interface;
    }

    pub fn init(
        console: *PrivConsole,
        buffer: []u8,
    ) Writer {
        return .{
            .console = console,
            .iow_interface = initInterface(buffer),
        };
    }

    fn initInterface(
        buffer: []u8,
    ) Io.Writer {
        return .{
            .vtable = &.{},
            .drain = drain,
        },
        .buffer = buffer,
    };
};

fn drain(
    io_w: *Io.Writer,

```

```

    data: []const []const u8,
    splat: usize,
) Error!usize {
    const w: *Writer = @alignCast(@fieldParentPtr("iow_interface", io_w));

    const console_uart = w.console uart_instance;
    if (!uart_allocator.isAllocated(console_uart))
        return std.Io.Writer.Error.WriteFailed;

    const uart_control = uart.uart_devices[console_uart].uart_control;
    uart_control.disableTxInterrupt();

    var write_count: usize = 0;

    const buffered = io_w.buffered();
    if (buffered.len != 0) {
        var buf_count: usize = 0;
        for (buffered) |tx_char| {
            if (tx_char == '\n') {
                uart_control.putTxChar('\r');
            }
            uart_control.putTxChar(tx_char);
            buf_count +=1;
        }
        _ = io_w.consume(buf_count);
    }

    for (data[0 .. data.len - 1]) |buf| {
        if (buf.len == 0) continue;

        var buf_count: usize = 0;
        for (buf) |tx_char| {
            if (tx_char == '\n') {
                uart_control.putTxChar('\r');
            }
            uart_control.putTxChar(tx_char);
            buf_count +=1;
        }
        write_count += buf_count;
    }

    const pattern = data[data.len - 1];
    if (pattern.len != 0) {
        var splat_counter: usize = splat;
        while (splat_counter != 0) : (splat_counter -= 1) {
            var buf_count: usize = 0;
            for (pattern) |tx_char| {
                if (tx_char == '\n') {
                    uart_control.putTxChar('\r');
                }
                uart_control.putTxChar(tx_char);
                buf_count +=1;
            }
        }
    }
}

```

```

        }
        write_count += buf_count;
    }

    return write_count;
}
};

};


```

Console I/O Example

This example demonstrates an echo server using standard library functions and configuring the UART for interactive framing. All input records are immediately echoed. The log console is used in the example.

```

<<speak-to-me-console-test.zig>>=
//! This file contains a simple test application demonstrating the use
//! of console I/O functions through a UART.
<<edit warning>>
<<copyright info>>

comptime {
    _ = @import("start_main");
}

const std = @import("std");
const log = std.log;
const Io = std.Io;
const svc_services = @import("svc_services");

pub const panic = svc_services.panic;
pub const std_options: std.Options = .{
    .logFn = svc_services.logToConsole,
};

pub fn main() !void {
    svc_services.Console.openAllConsoles();
    const sys_console = &svc_services.Console.sys_console;
    var sys_writer = sys_console.writer(&.{});
    const sysOut = sys_writer.interface();

    var reader_buf: [256]u8 = undefined;
    var sys_reader = sys_console.reader(&reader_buf);
    const sysIn = sys_reader.interface();

    while (true) {
        _ = sysOut.writeAll("All your codebase are belong to us.\n# ") catch |err|
            std.debug.panic("failed to write prompt: {t}", .{err});

        const taken = sysIn.takeDelimiterInclusive('\n');
    }
}

```

```

    if (taken) |input| {
        _ = sysOut.writeAll(input) catch |err|
            std.debug.panic("failed to write output: {t}\n", .{err});
    } else |err| {
        switch (err) {
            Io.Reader.DelimiterError.EndOfStream =>
                try sysOut.print("**** end of stream\n\n", .{}),
            else => std.debug.panic("failed to read input: {t}", .{err}),
        }
    }
}
}

```

Echo Example

This example shows an echo server working with raw input. Any input received is immediately transmitted. This example demonstrates reliable communications with only a relatively small buffer and hardware flow of control.

Also in this example, we run and service the watchdog timer. Note that watchdog acknowledgment proceeds without any interference from the I/O happening for the echo service.

```

<<echo-server-test.zig>>=
//! This file contains a application demonstrating the use the UART
//! as an echo server.
<<edit warning>>
<<copyright info>>

comptime {
    _ = @import("start_main");
}

const std = @import("std");
const config = @import("config");
const svc_services = @import("svc_services");
const wdog = svc_services.wdog;

pub const panic = svc_services.panic;
pub const std_options: std.Options = .{
    .logFn = svc_services.logToConsole,
};

pub fn main() !void {
    // To demonstrate that synchronization with the UART through
    // the event loop does not "block" the execution, run the
    // Watchdog timer. Any significant execution blockage will
    // cause a system reset. Here the Watchdog is set to interrupt
    // every 3500 ms and must be acknowledged at least every 4000 ms.
    try wdog.init(0, 4000, 3500, wdogDevNotifier, 0);
    try wdog.start(0, config.allow_wdog_reset);
}

```

```

svc_services.Console.openAllConsoles();

const log_console = &svc_services.Console.log_console;
var writer_buf: [256]u8 = undefined;
var log_writer = log_console.writer(&writer_buf);
const log_out = log_writer.interface();

var reader_buf: [256]u8 = undefined;
var log_reader = log_console.reader(&reader_buf);
const logIn = log_reader.interface();

while (true) {
    const streamed = try logIn.stream(log_out, .limited(writer_buf.len));
    if (streamed < writer_buf.len) try log_out.flush();
}
}

fn wdogDevNotifier(
    notification: *const anyopaque,
) void {
    const wdogNotification: *const svc_services.DevNotification =
        @ptrCast(@alignCast(notification));
    wdog.restart(wdogNotification.instance) catch
        @panic("Failed to restart watchdog timer");
}

```

Console Transmit Stress Test

```

<<console-stress-test.zig>>=
<<edit warning>>
<<copyright info>>

comptime {
    _ = @import("start_main");
}

const std = @import("std");
const config = @import("config");
const svc_services = @import("svc_services");

pub const panic = svc_services.panic;
pub const std_options: std.Options = .{
    .logFn = svc_services.logToConsole,
};

const buf_size: usize = 512;
var buffer: [buf_size]u8 = @splat('A');

pub fn main() !void {
    var buf_index: usize = 0;
    outer: while (true) {

```

```

    for ("ABCDEFGHIJKLMNPQRSTUVWXYZ") |cap_char| {
        buffer[buf_index] = cap_char;
        buf_index += 1;
        if (buf_index >= buffer.len) break :outer;
    }

const log_console = &svc_services.Console.log_console;
try log_console.open(null, true);

var writer_buf: [256]u8 = undefined;
var log_writer = log_console.writer(&writer_buf);
const log_out = log_writer.interface();

var len: usize = 1;
while (len < buf_size) : (len += 1) {
    try log_out.print("{d}: {s}\n", .{len, buffer[0..len]});
}

try log_out.flush();
log_console.flushSync();
try svc_services.halt.sysHalt();
}

```

[1] It is sometimes possible to do automatic baud rate determination

Just a Matter of Time

Since we experience the world around us in a Newtonian fashion, time can be considered separate from space and is an important independent coordinate in the way a system reacts to its environment. Timing services required by reactive systems tend to fall into the following broad areas.

Generating waveforms

Generating a waveform, either analog or digital, to control external peripherals is a major function of many systems. Motor control and audio output are just a couple of examples. Many external peripheral devices use specifically patterned digital waveforms for communications, e.g. I2C, SPI, and UART. In the last chapter we saw how UART peripherals may be used to generate the serial waveform for terminal communications. In a subsequent chapter, we show how the I2C peripheral accomplishes chip to chip communications. Complex waveform generation, especially to implement a defined protocol, usually requires specialized microcontroller peripherals. It is difficult to generate intricate waveforms using “bit banging” by the MCU, especially if the required speed is close to the clock rate of the processor or the response time of the GPIO peripheral. Modern SOC’s have specialized peripheral blocks which which can operate precisely at high speeds.

Measuring waveforms

Measuring analog and digital waveforms are important tasks that arise when interacting with the environment of the system. Precise measurement of pulse widths or precise sampling of an analog signal requires accurate timing, usually achieved with timer peripherals.

Internal software timing

When software deals with the external world, frequently, time is used to determine if a desired interaction will ever happen or needs to be retried. Communications protocol timeouts are an archetypical example, but waiting on external peripherals to accomplish some task is also common. This type of timing does not usually have the strict precision requirements as waveform timing.

Tracking human time

Humans keep time in an intricate way that is associated with astronomical phenomenon such as the rotational period of the Earth around the Sun. Specialized hardware for tracking human time is also usually available in the form of a so-called Real Time Clock (RTC). Computer oriented solutions to tracking time for humans is usually based on simple counting of periodic clock signals and then calculating the time in human terms based of fixing a specific time, i.e. the epoch, as time zero.

Tracking computer time

Having a computer monitor its own performance is a special case of time measurement. For this case, a specific set of base time services is required that support characterization of the performance of the software execution.

This chapter presents designs for the later three cases. We have previously seen and will see later designs for using peripherals to generate digital waveforms. We do not take up the weighty topic of measuring waveforms.

Driving Software with Time

For most microcontroller software purposes, initiating processing with approximately millisecond resolution works well. The frequency is not too high to require extraordinary considerations. The

type of timing considered here is useful for timeouts, for example, in handling communications, low speed sensor sampling, or button debouncing. Since there are many distinct timing service requests from application software, a major consideration is to make best use of the available timing peripherals. The number of timing requests by application software is not generally predictable but one can anticipate that there are many more software timing requests than physical timers. Multiplexing physical timer peripherals is one solution. To avoid unproductive computation, the design for multiplexing a timer must *not* involve periodic execution which accomplishes little more than decrementing a counter. Hardware is available for that purpose. The intent is to have hardware peripherals precisely inform the software when some action is to be taken.

In this chapter two topics are covered.

1. First, a design concept for multiplexing a single physical timer to produce multiple time expiration notifications is presented.
2. Second, the code is shown for the device background requests and corresponding foreground proxies which, together, provide timing service to background software.

Timer Queue Concepts

Using a single timer peripheral to drive a timing queue stored in time-relative order is a particularly handy technique to multiplex timer use. A timer peripheral is used to signal the elapsed time for the queue element at the head of the queue. The subsequent elements in the queue record the amount of additional time, after the expiration of the previous entry, which must elapse before the element expires. When the element at the head of the timer queue expires, it is removed from the queue and the next element's expiration time is loaded into the timer. Thus, the total time elapsed before the second element expires is the expiration time for the first element plus the expiration time for the second one. This logic applies to all the subsequent elements in the queue.

The following diagram illustrates these ideas by showing a schematic snapshot of a timer queue.

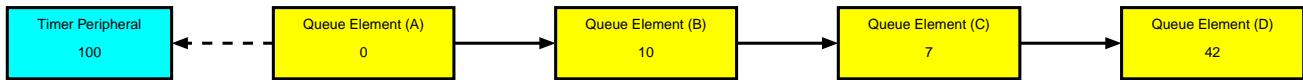


Figure 17. Snapshot of Timer Queue Operations

In this diagram, the yellow boxes represent timer queue elements and contain their relative counts. The cyan box represents the timer peripheral which is currently timing a 100 tick delay. This is the offset from the current time to when **Queue Element (A)** expires (note the count for **(A)** is zero indicating that no additional time beyond that in the timer peripheral is required). **Queue Element (B)** is to expire 10 ticks after **Queue Element (A)**. **(C)** is to expire 7 ticks after **(B)** and **(D)** is to expire 42 ticks after **(C)**. So, **Queue Element (D)** expires $100 + 0 + 10 + 7 + 42 = 159$ ticks after the moment shown in the diagram.

Specifically, the design concept is:

- A queue of relative expiration times is maintained.
- A timer peripheral is devoted to counting down the expiration time of the element at the head of the queue.
- When the head element expires, a background notification is sent for the that element. A separate notification is also sent for all elements following the head element which have an additional expiration time of zero (i.e. those elements which were intended to expire at the same time as the head element).
- Each queue element also carries a *reload* time to support periodic expiration without having to process additional requests. A reload value of zero, indicates that the element produces only a

single expiration notification.

The fundamental operations on the timer queue data structure are:

- a. *insert* an element
- b. *remove* an element
- c. determine the *remaining* time before an element expires

It is also necessary to supply a request to “open” and “close” a timer queue device so that a background notification proxy function may be specified. The background notification proxy implements the actions taken by background processing when a timer queue element expires.

Keeping the expiration times in the queue ordered by relative amounts of additional time that must elapse before the element expires minimizes the computation required to load the next time value into the timer peripheral. The next value to place into the timer peripheral is *pre-computed* and it is only necessary to walk the timer queue to expire elements. Upon finding the first non-zero expiration time, the timer is loaded with the value that counts the additional time before, what is now the head of the timer queue, expires. This is an attempt to minimize the timing *jitter* which is a consequence of using a single timer peripheral and requiring software to reload and control the peripheral.

Minimizing the computation required when a queue element expires comes at a cost. This scheme trades off lower costs at expiration for additional costs when an element is inserted or removed from the timer queue. In the insert case, the queue must be searched, in order, to find the appropriate place to insert the element accounting for the time that will have elapsed before the newly inserted element progresses to the head of the queue. When an element is removed, its relative elapsed time must be added to the next element in the queue to keep the timing correct for those elements which expire after the removed one. An update operation is also supported. Updating an element is equivalent to removal and re-insertion of the same element but with a new expiration time.

The next section shows the implementation of a timer queue and the basic operations that act upon it. Later, the timer queue operations are packaged into the background requests and foreground proxies according to the established pattern.

Timer Queue Elements

A timer queue element is the data object held in the timer queue. There are a couple of design decisions to be made with respect to the structure of the queue elements:

- A doubly-linked list, which is treated as a queue, is used to hold the timer queue. Two link pointers make inserting and removing at arbitrary places in the queue much simpler. The Zig standard library module, `DoublyLinkedList`, serves this use case well.
- The linked list provided by `std.DoublyLinkedList` is *intrusive*, i.e. the structure of each queue element contains the storage necessary for the linked list pointers.

Since the timer queue is a list of queue elements, storage for the list nodes that are held in the queue must be allocated. This is one of the cases where it seems advisable to provide storage space and allocation as part of the foreground processing. You could design the interface such that queue element storage was managed by the background requests, but this would complicate the background request interface considerably and simply move the consideration of how to size the storage pool for queue elements to another part of the system. Allocating queue elements in the background would also expose essential control values to unprivileged processing.

Sizing the storage pool for queue elements is problematic at this stage. As usual, a best guess is

used with the understanding that it may need to be resized when a specific application is deployed.

```
<<system_config: constants>>=
// The number of timer queue elements for a timer queue device.
pub const timq_element_count: usize = 32;
```

```
<<timer queue proxies>>=
pub const element_count: DevSvcInstance = system_config.timq_element_count; ①
```

- ① This is a slightly different usage of the DevSvcInstance type. Requests to timer queue elements use the instance field of the request to identify the element on which the operation is performed.

Storage is contained in an array, and the allocation just uses std.StaticBitSet. Separate storage pools are kept for each timer queue instance. You could consider having a single element pool which is shared among all timer queue instances. Unfortunately, that design would suffer from non-deterministic behavior, *i.e.* the number of elements available to be queued would depend upon how the pool of queue elements is used by other parts of the system. Deterministic behavior is preferred even at the cost of poorer utilization for some memory. Also, a single storage pool would mean that operations on the pool would need to disable multiple interrupts. The system timer peripheral has distinct IRQ vectors for each of the compare registers used to expire timer elements.

Two instances of the timer queue logical device are supported. One instance is dedicated to specific timing that is required in Part II of this book. The other instance is available for application uses such as time sequenced processing.

```
<<timer queue services>>=
pub const instances: DevSvcInstance = 2; // need config parameter for this
```

Since the timer queues are a system resource, we setup the means to allocate them.

```
<<device access>>=
const timq_allocator = DeviceAllocator(svc_services.Timq.instances, &{});
```

System Timer Comparators

The following figure is taken from the Apollo 3 data sheet and shows a block diagram of the system timer peripheral.

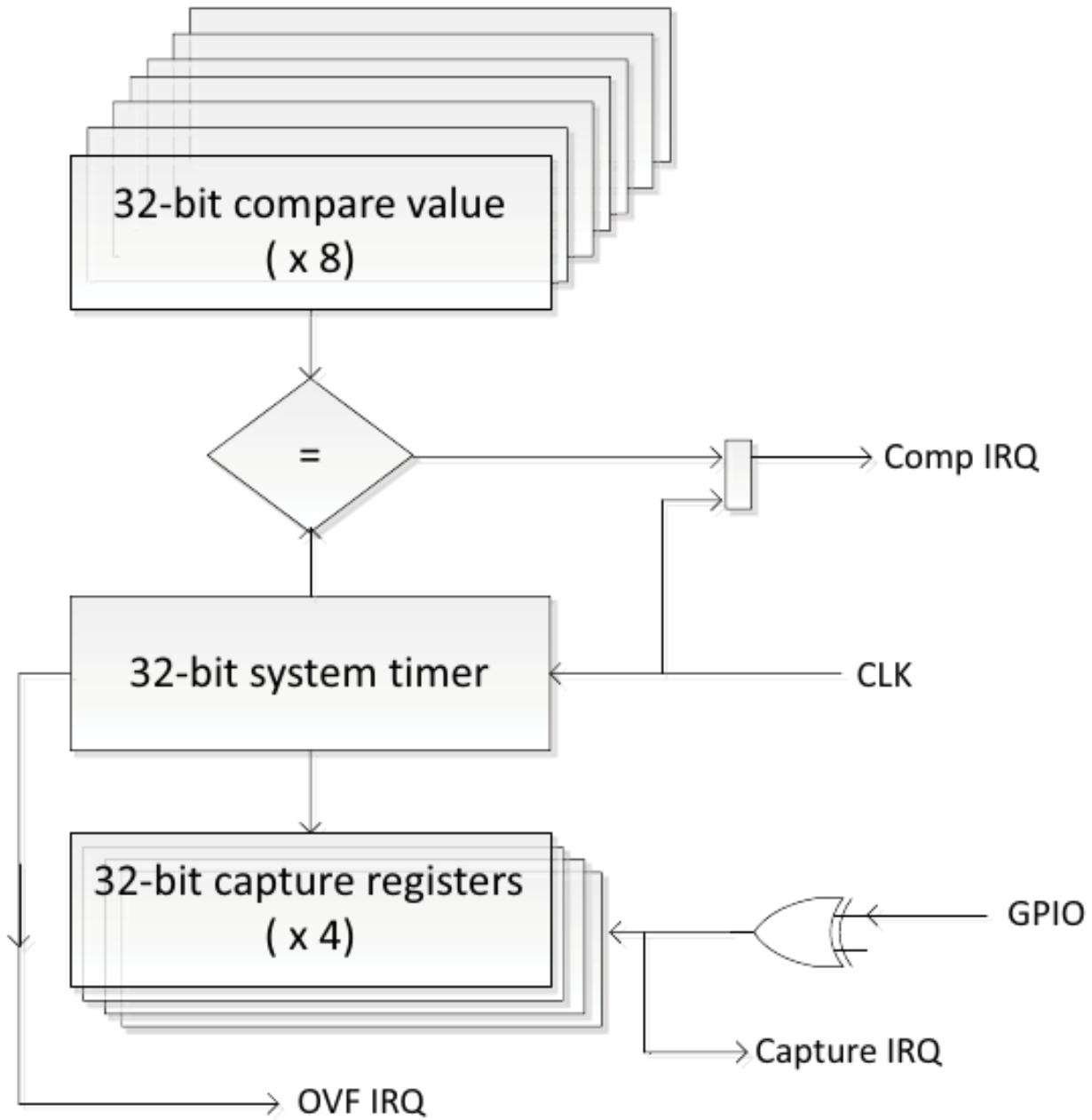


Figure 18. Block Diagram of the Apollo 3 System Timer

The Apollo 3 SOC supplies a system timer peripheral which consists of a register which counts at a programmable frequency and has eight compare registers which indicate when there is a match between the counter and the compare register. This [peripheral](#) was discussed in Chapter 2 where it was initialized.

A system timer compare register is the hardware basis for operating a timer queue instance. There are eight timer compare registers^[1]. The operating rules for the compare registers are slightly unusual.

- The value **written** to the compare registers is the offset, in timer ticks, from *now* to when the comparator is to match. The addition of the offset to the current value of the timer counter, yielding the compare register match value, is done in hardware. This is quite a nice feature and

essential given the high frequency the counter clock can support.

- The values **read** from the compare register is the counter value at which the compare register will match, i.e. after writing an offset, you read by the sum computed in hardware.
- Each compare register is wired to a separate NVIC interrupt line. This is somewhat unusual as most SOC designs wire peripherals to only a single NVIC interrupt line and then use an internal register to indicate the source of the interrupt. This arrangement gives software a separate IRQ vector directly connected to the compare register interrupt. This is also a nice feature for faster response to compare register matches.

Two compare registers of the system timer peripheral are allocated to run two instances of timer queues. For one timer queue instance, there is a definite plan for its use which is considered in Part II of the book. The other instance is available for any application purpose which needs to perform time ordered sequencing of actions.

Since the system timer compare registers are a system wide resource, we set up our usual allocation scheme to access them.

```
<<device access>>=
const systimer_allocator =
    DeviceAllocator(apollo3.SystemTimer.compare_reg_count, &.{0, 1}); ①
```

- ① We pre-allocate comparators 0 and 1. It would be preferable to be able to use an arbitrary comparator, but that is not possible since each comparator register has its own vector in the vector table. Recall, that the vector table resides in read-only memory and so the code is not able to adapt to use an arbitrary comparator register since it cannot write the correct IRQ handler pointer into the vector table.

Timer Queue Device Components

The following figure shows the Apollo 3 peripheral components that are used to implement the timer queue logical device.

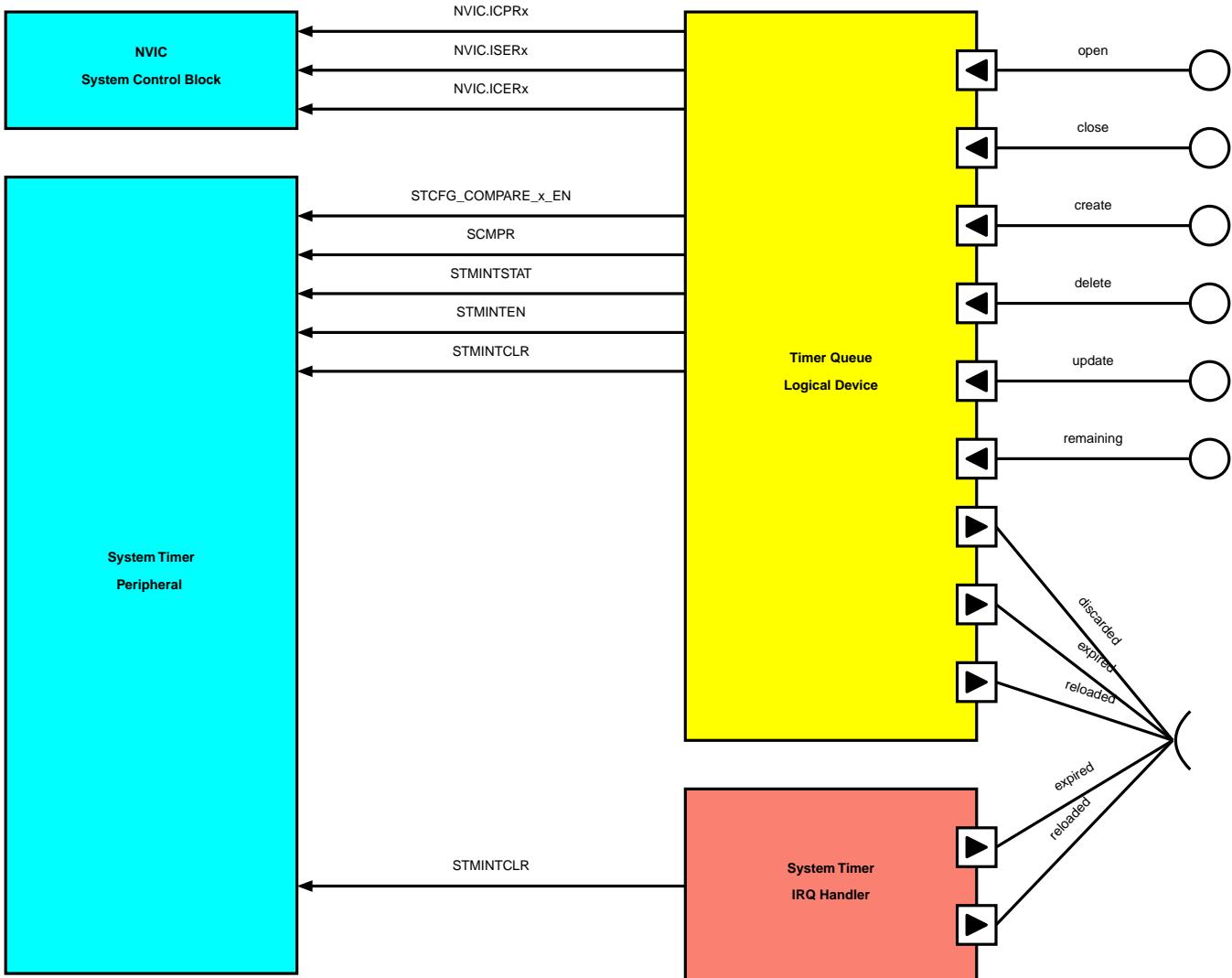


Figure 19. Timer Queue Logical Device Components

Given the rules of operation for the compare registers, the necessary hardware controls are few.

```
<<timer queue proxies>>=
/// The collection of hardware I/O controls required to handle the
/// system timer comparator.
const StimerControl = struct {
    /// A pointer to the memory mapped I/O for the compare register.
    cmp_reg: *volatile mmio.WordRegister,
    /// A bit field with a single set bit which corresponds to the interrupt
    /// control registers for the compare register. The interrupt status, set, and
    /// clear registers all have the same bit-field arrangement.
    intr_bit_field: @TypeOf(apollo3.stimer.stim_int_en).BitField,
    /// A bit field with a single set bit which corresponds to the enable control
    /// for the compare register used for the timer queue.
    cmp_enable_bit_field: @TypeOf(apollo3.stimer.st_cfg).BitField,

    <<timer device control operations>>
};
```

The following function show the direct control over the compare register in the system timer.

```
<<timer device control operations>>=
/// The `getCompareReg` function returns the current value of the given system
/// timer comparator register.
fn getCompareReg(
    self: *const StimerControl,
) mmio.WordRegister {
    return self.cmp_reg.*;
}
```

```
<<timer device control operations>>=
/// The `getInterruptStatus` function returns the interrupt status for the
/// given timer comparator register.
fn getInterruptStatus(
    self: *const StimerControl,
) bool {
    return apollo3.stimer.stim_int_stat.extractFieldValue(self.intr_bit_field) == 1;
}
```

```
<<timer device control operations>>=
/// The `disableCompareInterrupt` function disables the interrupt from the
/// given timer comparator register.
fn disableCompareInterrupt(
    self: *const StimerControl,
) void {
    apollo3.stimer.stim_int_en.insertFieldValue(self.intr_bit_field, 0);
}
```

```
<<timer device control operations>>=
/// The `enableCompareInterrupt` function enables the interrupt from the
/// given timer comparator register.
fn enableCompareInterrupt(
    self: *const StimerControl,
) void {
    apollo3.stimer.stim_int_clr.writeValue(self.intr_bit_field, 1);
    apollo3.stimer.stim_int_en.insertFieldValue(self.intr_bit_field, 1);
}
```

```
<<timer device control operations>>=
/// The `disableCompareRegister` function disables the given comparator register
/// along with its interrupt in the system timer peripheral.
fn disableCompareRegister(
    self: *const StimerControl,
) void {
    self.disableCompareInterrupt();
    apollo3.stimer.st_cfg.insertFieldValue(self.cmp_enable_bit_field, 0);
}
```

```

<<timer device control operations>>=
/// The `enableCompareRegister` function enables the given comparator register
/// along with its interrupt in the system timer peripheral.
fn enableCompareRegister(
    self: *const StimerControl,
) void {
    self.enableCompareInterrupt();
    apollo3.stimer.st_cfg.insertFieldValue(self.cmp_enable_bit_field, 1);
}

```

It is common for SOC's to have errors in their circuitry. These so called, "errata", are typically cataloged by the chip manufacturer along with any available workarounds. In this case the Ambiq Apollo 3 erratum **ERR014** [2] details the problem and a workaround. The workaround is in the HAL code of the Ambiq SDK and the basic processing from the HAL code (which follows the erratum prescriptions) has been used in the following function.

```

<<timer device control operations>>=
/// The `setExpirationTime` function sets the value of the `expire_time` argument
/// into the given system timer compare register. The `expire_time` value is
/// the number of clock ticks from now that the compare register will match.
/// This function follows the Ambiq recommendation to work around an _erratum_
/// associated with compare registers.
/// Note if setting the compare register value is successful, the compare
/// register and its interrupt are left enabled.
fn setExpirationTime(
    dev_control: *const StimerControl,
    expire_time: timq_time.Ticks,
) void {
    var tries: usize = 0;
    const timer_access_latency: usize = 2; ①

// BEGIN CRITICAL SECTION
{
    v7m.disableIrq(); defer v7m.enableIrq(); ②

    dev_control.disableCompareRegister();

    while (tries < 4) : (tries += 1) {
        const expected_lower = apollo3.stimer.st_tmr + expire_time;
        const expected_upper = expected_lower + timer_access_latency;
        dev_control.cmp_reg.* = expire_time;
        const compare = dev_control.cmp_reg.*;
        if (compare >= expected_lower and compare < expected_upper){
            dev_control.enableCompareRegister();
            return;
        }
    }
}

// END CRITICAL SECTION

panic.privPanic(null, "unable to set system timer expiration time");

```

```
}
```

- ❶ The HAL code uses a latency value of 10 ticks. No explanation is given for why that particular value was chosen. For the clock frequency at which the system timer is run, 10 ticks is greater than 300 μ s. This seems excessive since our clock tick is about 30.5 μ s. We choose 2 which is approximately 61 μ s.
- ❷ *N.B.* this uses a critical section with all interrupts disabled. Since the workaround for the race condition in the hardware requires repeated iterations in software, the workaround code must execute to completion without interruption.

Timer Queue Device Control

There are three aspects of the timer queue design which must be managed:

1. Storage, allocation, and operations for the timer elements.
2. Sending background notifications when the status of a timer element changes.
3. Programming the actions of a compare register in the system timer peripheral.

This leads to the following members for a data structure used to control the timer queue device operations.

```
<<timer queue proxies>>
const TimqDevice = struct {
    /// Data and methods to access and control the timer queue.
    queue: QueueControl,
    /// The storage for the timer queue elements,
    storage: ElementStorage,
    /// Control data and processing for handling NVIC interactions.
    irq_control: IrqControl,
    /// The parameters needed to control the system timer hardware.
    dev_control: StimerControl,
    /// The background notification proxy.
    proxy: DevSvcNotifyProxy,

    <<timer queue operations>>
};
```

In addition to the expire and reload times, each timer queue element contains a notification closure field. There is only a single background notification function for the queue, so the notification closure value is needed to determine the actions of the background proxy function. We also keep the element identifier in each element. It is convenient for generating the background notification.

```
<<timer queue operations>>
/// The data held in each timer queue element.
const QueueElement = struct {
    /// An identifier for the timer queue element.
    element_id: ElementId,
    /// The minimum number of timer ticks which are to elapse before a notification
    /// is sent.
    expire_time: timq_time.Ticks = 0,
    /// The number of timer ticks for repeated timer expiration. A `reload` value
```

```

    /// of 0, implies the timer element is used as a one-shot notification.
    reload_time: timq_time.Ticks = 0,
    /// A value returned in any background notification associated with the timer
    /// element.
    notify_closure: usize = 0,
    /// List linkage for each element.
    node: std.DoublyLinkedList.Node = .{},
};


```

Control of the timer queue is then just the list of timer queue elements associated with the operations on the queue. Those operations are discussed in the next section.

```

<<timer queue operations>>=
/// The timer queue elements are an intrusive doubly linked list.
const QueueControl = struct {
    element_queue: std.DoublyLinkedList = .{},

    <<queue control operations>>
};


```

Finally, the storage mechanism follows similar patterns we have established. The timer queue elements are held in an array and a static bit set is used to track which elements are allocated.

```

<<timer queue operations>>=
const ElementStorage = struct {
    const Allocator = std.StaticBitSet(element_count);
    storage: [element_count]QueueElement = undefined,
    allocation: Allocator = Allocator.initEmpty(),

    /// The `init` function initializes the timer queue element storage.
    /// All timer elements are discarded and the storage is made to
    /// appear empty.
    fn init(
        self: *ElementStorage,
    ) void {
        self.allocation = Allocator.initEmpty();
        for (&self.storage, 0..) |*element, element_index| {
            element.element_id = @truncate(element_index);
        }
    }

    /// The `allocate` function finds a free timer queue element node and
    /// marks it as allocated, returning either a pointer to the allocated
    /// queue element or an error indicated no queue elements are available.
    fn allocate(
        self: *ElementStorage,
    ) !*QueueElement {
        var element_iter = self.allocation.iterator(.{ .kind = .unset });
        while (element_iter.next()) |element_index| {
            self.allocation.set(element_index);
            self.storage[element_index] = .{

```

```

        .element_id = @truncate(element_index),
    };
    return &self.storage[element_index];
}

return error.OutOfMemory;
}

/// The `free` function marks the timer queue element given by `element`
/// as free so that it may be reallocated in the future.
fn free(
    self: *ElementStorage,
    element: *QueueElement,
) void {
    const element_id = element.element_id;
    self.allocation.unset(element_id);
    self.storage[element_id] = undefined;
}

fn isAllocated(
    self: *ElementStorage,
    element_id: ElementId,
) bool {
    return self.allocation.isSet(element_id);
}

fn elementFromId(
    self: *ElementStorage,
    element_id: ElementId,
) !*QueueElement {
    return if (self.isAllocated(element_id))
        &self.storage[element_id]
    else
        error.UnknownElement;
}
};


```

The values of the control blocks are initialized at compile time to save ourselves yet another initialization function. The first two compare registers in the system timer peripheral are allocated to supply the timing for the queues.

```

<<timer queue proxies>>=
var timer_queues = [instances]TimqDevice{
    .{ // 0
        .queue = .{},
        .storage = .{},
        .irq_control = .{
            .irq_number = apollo3.SystemTimer.compare_a_irq_number,
        },
        .dev_control = .{
            .cmp_reg = &apollo3.stimer.s_cmr[0],
            .intr_bit_field = @TypeOf(apollo3.stimer.stim_int_en).BitField.init(),
        }
    }
};

```

```

compare_a),
    .cmp_enable_bit_field = @TypeOf(apollo3.stimer.st_cfg).BitField.init
(.compare_a_en),
},
.proxy = undefined,
},
.{ // 1
.queue = .{},
.storage = .{},
.irq_control = .{
    .irq_number = apollo3.SystemTimer.compare_b_irq_number,
},
.dev_control = .{
    .cmp_reg = &apollo3.stimer.s_cmpr[1],
    .intr_bit_field = @TypeOf(apollo3.stimer.stim_int_en).BitField.init(
compare_b),
    .cmp_enable_bit_field = @TypeOf(apollo3.stimer.st_cfg).BitField.init
(compare_b_en),
},
.proxy = undefined,
},
};


```

Timer Queue Operations

This section defines the set of functions which act as operators on the timer queue control blocks.



All the timer queue operations assume that the IRQ Handler for the corresponding timer compare register does not execute during the operation. This happens either because the IRQ handler itself is currently executing (recall all IRQ handlers run at the same priority in this design) or that the foreground proxy functions (associated with timer queue requests coming from the background via the SVC exception) has explicitly disabled the interrupt. The operations share data with the IRQ handler and must implement a exclusive section with respect to the compare register interrupt associated with the queue. This is made visibly distinct in the code below.

A timer queue operates in one of two states.

1. The queue is running. The hardware timing peripheral is counting down the expiration time of the element at the head of the queue. In this case, the `expire_time` member of the element at the head of the timer queue is zero as its expiration time was moved into the hardware. The head element requires no additional expiration time past that loaded into the system timer compare register.
2. The queue is stopped, implying that the timer compare register is not being used. For this case, any residual time remaining in the timer register is placed back into the `expire_time` member of the element at the head of the queue. Stopping the timer queue should not be confused with stopping the timer peripheral. The timer peripheral counter is never stopped.

The timer queue must be stopped for any operations on the timer queue except for expiring queue elements. This is required in order to correctly insert or remove an element keeping the ordering relationship among the elements correct. For example, when inserting a new element, the requested expiration time might be shorter than the residual time of the element at the head of the queue.

Since starting and stopping the timer queue involve interactions with the system timer peripheral, the coding requires more than the usual care. The hardware counters run asynchronously to the processor. This means that there is a race condition between what the timer is doing compared to what the processor sees. The difference arises because the processor must execute instructions to view the actions of the peripheral through its memory mapped registers. This can be obscured, especially in the case of Zig language statements, since there is not necessarily a one-to-one correspondence from language statements to machine instructions which are accessing the timer peripheral.

stop

To stop the timer queue, the residual time remaining in the hardware must be determined. The residual time is the difference between the compare register value and the timer counter. After reading and computing that difference, a check of the interrupt status is made to determine if the compare register matched during the time it took to obtain the residual time. If the interrupt status is set (*N.B.* since the interrupt is disabled at this point, execution has not been preempted), then the race was “lost” and simply declare the queue element as expired. The expiration is indicated by a `expire_time` element value of 0.

```
<<timer queue operations>>
/// The `stop` function halts timer queue operations for the queue given by
/// `tdev` and places the residual unexpired time of the element at the head of the
/// queue back into that element's `expire_time` member.
fn stop(
    /// A pointer to the timer queue control block to be stopped.
    tdev: *TimqDevice,
) void {
    const queue = &tdev.queue.element_queue;
    if (queue.first) |head| {
        const counter = apollo3.stimer.st_tmr;
        const expire = tdev.dev_control.getCompareReg();
        const q_elem: *QueueElement = @fieldParentPtr("node", head);
        q_elem.expire_time = expire -% counter; ①

        const already_expired = tdev.dev_control.getInterruptStatus();
        if (already_expired) {
            q_elem.expire_time = 0; ②
        }
    }
}
```

① Because of the wrapping subtraction, this expression works even if the expiration time wrapped around when it was loaded into the compare register.

② If the interrupt status indicates a match, then there is no residual time.

start

Starting the timer queue is easier. The hardware interactions must be ordered such that the compare register interrupt is cleared and ready to react before loading the compare register. This ensures that the interrupt can be latched before installing the time value into the compare registers. For small expire times, this is important. After loading the compare register, the interrupt is enabled.

```

<<timer queue operations>>=
/// The `start` function starts the queue given by `tdev` by placing the
/// `expire_time` value of the element at the head of the timer queue into the
/// hardware peripheral which is used for the timer queue.
/// Note, this function must be invoked in an interrupt section where the
/// timer queue comparator register interrupt is disabled or when we
/// are running at IRQ priority.
fn start(
    /// A pointer to the timer queue control block to be started.
    tdev: *TimqDevice,
) void {
    tdev.expireElements(); ①

    const queue = &tdev.queue.element_queue;
    if (queue.first) |head| {
        const q_elem: *QueueElement = @fieldParentPtr("node", head);
        std.debug.assert(q_elem.expire_time != 0);
        tdev.dev_control.setExpirationTime(q_elem.expire_time); ②
        q_elem.expire_time = 0;
    } else {
        tdev.dev_control.disableCompareRegister(); ③
    }
}

```

- ① Before starting the timer queue, any elements which have an `expire_time` of zero must be expired. Expired elements could come either when the timing was stopped (see `timqStop` above) or if a timer element with a zero expire time was inserted or updated. In either case, the timer queue must be placed into a state where it has a non-zero value as the `expire_time` for the element at the head of the queue. Since expiring timer queue elements involves the notification queue, we must make sure no other interrupt runs.
- ② Setting the time value into the compare register was factored into a separate function because of a chip erratum. See below.
- ③ Note that the compare register interrupt is left disabled when the timer queue is empty.

```

<<timer queue operations>>=
/// The `delete` function removes the timer queue element given by `element_id`
/// from the timer queue given by `tdev`.
fn delete(
    /// A pointer to the timer queue control block from which the element is removed.
    tdev: *TimqDevice,
    /// The identifier of the element to remove.
    element_id: ElementId,
) !void {
    const element = try tdev.storage.elementFromId(element_id);
    tdev.queue.remove(element);
    tdev.storage.free(element);
}

```

init

When the timer queue is opened, the element queue is initialized by the open function.

```
<<queue control operations>>=
fn init() QueueControl {
    return {.element_queue = .{}};
}
```

insert

Where the start and stop operations interacted with the hardware, the insert, remove, and remaining operations are only concerned about searching the timer queue and maintaining the relative expiration times in proper order. Insertion means walking the queue, subtracting the expiration times from the requested time until the place in the queue is found where an element with the requested time belongs.

```
<<queue control operations>>=
/// The `insert` function inserts the timer queue element given by
/// `element` into the timer queue pointed to by `tdev`. It is assumed that the
/// `expire_time` and `reload_time` members of the `element` structure have been
/// set to the requested expiration and reload times, respectively.
fn insert(
    /// A pointer to the timer queue control block to which the insertion is made.
    qcb: *QueueControl,
    /// A pointer to a timer queue element which is inserted into queue given by `qcb`.
    element: *QueueElement,
) void {
    var queue_iter = qcb.element_queue.first;
    while (queue_iter) |node| : (queue_iter = node.next) {
        const q_elem: *QueueElement = @fieldParentPtr("node", node);
        if (element.expire_time < q_elem.expire_time) { ①
            q_elem.expire_time -= element.expire_time; ②
            break;
        } else {
            element.expire_time -= q_elem.expire_time;
        }
    }
    if (queue_iter) |insert_node| {
        qcb.element_queue.insertBefore(insert_node, &element.node);
    } else {
        qcb.element_queue.append(&element.node);
    }
}
```

- ① By making the comparison strictly less than, the ordering of the expiration of the elements that, coincidentally, are meant to expire at the same time is maintained to be the same as the order of insertion.
- ② This is the right spot, but the expiration time of the new entry must be subtracted from those which follow in order not to add time to the trailing entries.

Note particularly what happens in the above code if the inserted element's `expire_time` is zero, i.e.

if `element.expire_time` is zero. It is inserted into the queue as expected, but behind any other elements that might have already expired. This maintains an ordering where elements expired by virtue of the timer precede those expired by virtue of a supplied request. This is the desired order. The implication is that requesting an expiration time of zero expires the element immediately, but in the correct order if there are other elements that have expired coincidentally to the request.

Removing a queue element is similar, but is easier to compute since we have direct access to the storage pool for the timer queue elements. In the removal case, any expiration time of the removed element must be added to the next element in the list. Of course, the boundary condition of removing the last element in the queue requires a conditional test. Note that removing an element from the queue does *not* automatically return the element to the storage pool. An update request uses this function to remove the queue element, but then inserts the queue element with a new expiration time.

```
<<queue control operations>>=
/// The `remove` removes the queue element pointed to by `element` from
/// the timer queue referenced by `tdev`, preserving the time ordering of the queue.
fn remove(
    /// A pointer to the queue control block from which the removal is made.
    qcb: *QueueControl,
    /// A pointer to a timer queue element which is removed from queue given by `qcb`.
    element: *QueueElement,
) void {
    if (element.node.next) |next_element| {
        if (next_element == qcb.element_queue.last) {
            const q_elem: *QueueElement = @fieldParentPtr("node", next_element);
            q_elem.expire_time += element.expire_time; ①
        }
    }
    qcb.element_queue.remove(&element.node);
}
```

- ① If the removed element is not the last one in the queue, any expiration time associated with it must be added to its next neighbor to preserve the time ordering.

Note that if it were *not* required to handle any residual time in the removed element, the iteration across the timer queue would be unnecessary since an element of a doubly-linked list can be removed by direct pointer manipulation.

Computing the remaining time for an element is similar to the inverse of what happens at insertion time. Where insertion subtracted off the preceding expiration times, here the queue is walked adding together all the expiration times until the matching element is found.

```
<<queue control operations>>=
fn remainingTime(
    /// A pointer to the queue control block on which the remaining time is computed.
    qcb: *QueueControl,
    /// The identifier of the element for which the remaining time is determined.
    element_id: ElementId,
) timq_time.Ticks {
    var remaining_time: timq_time.Ticks = 0;

    var queue_iter = qcb.element_queue.first;
```

```

while (queue_iter) |node| : (queue_iter = node.next) {
    const q_elem: *QueueElement = @fieldParentPtr("node", node);
    remaining_time += q_elem.expire_time;
    if (q_elem.element_id == element_id) {
        break;
    }
}

return remaining_time;
}

```

There are two operations associated with sending background notifications. When an element expires, the queue is traversed finding all the elements with zero expiration times and a background notification is posted. Once a notification has been sent for an element, it may be returned to the storage pool if it is a one-shot expiration or re-inserted into the timer queue if it has periodic expiration.

```

<<timer queue operations>>=
/// The `expireElements` function sends background notifications to all timer queue
/// elements which have expired, i.e. those elements in the queue that have a zero
/// `expire_time`.
fn expireElements(
    /// A pointer to the timer queue control block which contains the elements to expire.
    tdev: *TimqDevice,
) void {
    const element_list = &tdev.queue.element_queue;
    var queue_iter = element_list.first;
    while (queue_iter) |element| {
        var current: *QueueElement = @fieldParentPtr("node", element);
        queue_iter = current.node.next; ①

        if (current.expire_time == 0) {
            tdev.queue.element_queue.remove(element);
            const element_id = current.element_id;
            const notify_closure = current.notify_closure;
            const reason: NotifyReason = if (current.reload_time == 0) eblk: {
                tdev.storage.free(current);
                break :eblk .expired;
            } else rblk: {
                current.expire_time = current.reload_time;
                tdev.queue.insert(current);
                break :rblk .reloaded;
            };
        }
    }

    //BEGIN PRIORITY SECTION
    const section =
        v7m.basepri.PrioritySection.enter(system_config.irq_encoded_priority); ②
    defer section.leave();

    const result: SvcResult, const notification: *DevNotification =
        dev_notify_queue.reserveNotification();
    if (result != .success) {

```

```

        panic.privPanic(null, "timer queue notification failed");
    }
    notification.* = DevNotification.init(
        .timq,
        element_id,
        tdev.proxy,
        notify_closure,
        .{
            .timq = reason,
        },
    );
    dev_notify_queue.commitNotification();
//END PRIORITY SECTION
} else {
    break; ③
}
}
}
}

```

- ① Since an element may be removed during the queue traversal and element removal invalidates the list iterator, the iterator must be advanced_before_ removing any element.
- ② Since events are expired both at SVC priority level and IRQ priority level, we insert a priority section to ensure protecting the queue, despite the fact that it is superfluous when expireElements is called from the IRQ handler.
- ③ The first non-zero expiration time marks the end of those elements which require notifications.

If the timer queue is “closed” and there are still elements in the queue, a background notification is posted for all elements. The notification has a status indicating that the queue element has been discarded. This informs background processing that no notifications for those elements will ever come.

```

<<timer queue operations>>=
/// The `closeElements` function sends background notifications to all elements in
/// the timer queue indicating the status of the element as `discarded`.
fn closeElements(
    /// A pointer to the timer queue control block which contains elements to discard.
    tdev: *TimqDevice,
) void {
    const element_list = &tdev.queue.element_queue;
    var queue_iter = element_list.first;
    while (queue_iter) |element| {
        const current: *QueueElement = @fieldParentPtr("node", element);
        queue_iter = current.node.next; ①

//BEGIN PRIORITY SECTION
{
    const section =
        v7m.basepri.PrioritySection.enter(system_config.irq_encoded_priority); ②
    defer section.leave();

    const result: SvcResult, const notification: *DevNotification =
        dev_notify_queue.reserveNotification();
}
}
}
}

```

```

        if (result != .success) {
            panic.privPanic(null, "timer queue discarded notification failed");
        }
        notification.* = DevNotification.init(
            .timq,
            current.element_id,
            tdev.proxy,
            current.notify_closure,
            .{
                .timq = .discarded,
            },
        );
        dev_notify_queue.commitNotification();
    }
    //END PRIORITY SECTION

    tdev.queue.element_queue.remove(element);
    tdev.storage.free(current);
}
}

```

- ① Again, linked list removal invalidates the iterator and it must be advanced *before* removal.
- ② We are sending notifications from SVC priority level. We must raise the base priority to be irq priority to avoid contention with the device notification queue.

Background Timing Queue Requests

All the foundation is in place to run the timer queue. What remains is to provide an interface to background processing to make timer queue requests. This interface follows the pattern for device control already seen.

```

<<system_services: device service definitions>>=

/// Timer Queue services
pub const timer_queue = struct {
    <<timer queue services>>
};

```

Timer queue request encoding

First, the timer queue request operations are encoded as small integer numbers. Since the enumerator values are used for dispatching the corresponding function, they start at zero and the values are sequential.

```

<<timer queue services>>=
pub const TimqOperations = DefineDeviceOperations(
    .timq,
    &{ "open", "close", "create", "delete", "update", "remaining" },
);

```

Open Timer Queue

The background function which marshals the open request follows the same pattern used previously: fill in the parameter structures and make a device realm SVC call.

```
<<timer queue services>>=
/// The `open` function makes the timer queue given by, `timq`, ready for timing
/// queue operations. When a timing queue element expires, the background
/// notification contains the proxy function, `notify_proxy`, which is to be
/// invoked to process the element expiration in the background. This function
/// must be invoked before any other operations on the timer queue.
pub fn open(
    /// The instance number which identifies which timer queue is to be opened.
    timq: DevSvcInstance,
    /// A pointer to a notification proxy function which is to be invoked during
    /// background processing when a timer queue entry changes status.
    notify_proxy: DevSvcNotifyProxy,
) SvcError!void {
    if (timq >= instances) return SvcError.unknown_instance;
    const input_params = OpenInputParam.marshal(.{
        .notify_proxy = notify_proxy,
    });
    const request = TimqOperations.makeRequest(.open, timq);
    return devRealmSvcCall(request, @ptrCast(&input_params), null, null);
}
```

The input parameters must be transferred across the SVC interface and need an appropriate data structure to do so.

```
<<timer queue services>>=
pub const OpenInputParam = DefineSvcRequestParam(struct {
    notify_proxy: DevSvcNotifyProxy,
});
```

Open Timer Queue Proxy

```
<<device proxies>>=

/// Timer Queue service proxies
const timer_queue = struct {
    const ElementId = svc_services.timer_queue.ElementId;
    const ElementCount = svc_services.timer_queue.ElementId;
    const NotifyReason = svc_services.timer_queue.NotifyReason;
    const instances = svc_services.timer_queue.instances;

    <<timer queue proxies>>
};
```

The following function is the foreground proxy for open.

```

<<timer queue proxies>>=
fn open(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const inst = req.instance;
    if (!systimer_allocator.isAllocated(inst)) return .operation_failed;

    const OpenInputParam = svc_services.timer_queue.OpenInputParam;
    const open_params, const result = OpenInputParam.receive(input.?);
    if (result != .success) return result;

    const tdev = &timer_queues[inst];
    tdev.queue = TimqDevice.QueueControl.init();
    tdev.storage.init();
    tdev.proxy = open_params.notify_proxy;

    tdev.irq_control.clearPending();
    tdev.irq_control.enable();

    return .success;
}

```

Close Timer Queue

```

<<timer queue services>>=
/// The `close` function decommissions the timer queue instance given by `timq`.
/// After invoking `close` no further timer expiration notifications are made. Any
/// queued timer elements are notified with the status field in the notification
/// set to `.discarded`.
pub fn close(
    /// The instance number which identifies which timer queue is to be closed.
    timq: DevSvcInstance,
) SvcError!void {
    if (timq >= instances) return SvcError.unknown_instance;
    const request = TimqOperations.makeRequest(.close, timq);
    return devRealmSvcCall(request, null, null, null);
}

```

Close Timer Queue Proxy

The foreground proxy function returns any lingering queue elements and restores the state of the timer control block.

```

<<timer queue proxies>>=
fn close(
    req: DevSvcRequest,
    _: ?*const anyopaque,

```

```

    _ : ?*anyopaque,
    _ : ?*anyopaque,
) SvcResult {
    const inst = req.instance;
    if (!sysTimer_allocator.isAllocated(inst)) return .operation_failed;
    const tdev = &timer_queues[inst];
    tdev.irq_control.disable();
    tdev.dev_control.disableCompareRegister();
    tdev.closeElements();

    return .success;
}

```

Create Timer Element

```

<<timer queue services>>=
/// The `create` function allocates and inserts a new timer queue element into the timer
queue
/// given by, `timq`. The element is set to expire at, `expire_time`, milliseconds
/// from now. If the timer element is intended to be periodic, then `reload_time`
/// is supplied as non-zero and the second and subsequent expirations of the
/// element happen `reload_time` milliseconds after the first expiration. Note,
/// `expire_time` and `reload_time` need not be the same value for a periodic timer
/// queue element. It is possible to use `expire_time` to synchronize to some
/// ongoing period which has partially elapsed and subsequent notifications are
/// issued every `reload_time` milliseconds.
pub fn create(
    /// The instance number which identifies which timer queue into which a new
    /// timing element is inserted.
    timq: DevSvcInstance,
    /// The minimum number of milliseconds which must elapse before an expiration
    /// notification is sent. An `expire_time` value of 0, causes the newly
    /// inserted element to expire immediately.
    expire_time: u32,
    /// The minimum number of milliseconds which must elapse before an expiration
    /// notification is sent on the second and subsequent periods. A `reload_time`
    /// of zero indicates that the timer element issues only a single notification.
    /// A non-zero `reload_time` value indicates the inserted element is to issue
    /// repeated notifications every `reload_time` millisecond after the first expiration.
    reload_time: u32,
    /// A value returned in any background notification associated with the newly
    /// inserted timer element.
    notify_closure: usize,
) SvcError!ElementId {
    if (timq >= instances) return SvcError.unknown_instance;
    const input_params = CreateInputParam.marshal(.{
        .expire_time = expire_time,
        .reload_time = reload_time,
        .notify_closure = notify_closure,
    });
    var output_params: CreateOutputParam = .initial;

```

```

const request = TimqOperations.makeRequest(.create, timq);
try devRealmSvcCall(request, @ptrCast(&input_params), @ptrCast(&output_params), null);
return output_params.unmarshal().element_id;
}

```

The input parameter for the create function requires two tick times and a notification closure value.

```

<<timer queue services>>=
pub const CreateInputParam = DefineSvcRequestParam(struct {
    expire_time: u32,
    reload_time: u32,
    notify_closure: usize,
});

```

The scaling factor for timer ticks depends upon the frequency of the system timer peripheral clock. The clock ticks at 32786 Hz and is held in a 32 bit register. The queue element times are treated as an unsigned fixed binary point UQ17.15 number in units of seconds. Scaling functions are provided [below](#).

The output parameter returns the queue element identifier.

```

<<timer queue services>>=
pub const CreateOutputParam = DefineSvcRequestParam(struct {
    element_id: ElementId,
});

```

Create Timer Element Proxy

The device realm foreground proxy function is a wrapper around the queue operation, `create`. It is necessary to allocate a timer queue element and fill in the parameters. With an element in hand, the timer queue operation sequence is stop, insert, start. As shown below and in subsequent proxy functions, the core of the logic of the queue element proxy functions is contained in the timer queue operations defined previously.

```

<<timer queue proxies>>=
fn create(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    output: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const inst = req.instance;
    if (!systimer_allocator.isAllocated(inst)) return .operation_failed;

    const CreateInputParam = svc_services.timer_queue.CreateInputParam;
    const input_params, const result = CreateInputParam.receive(input.?);
    if (result != .success) return result;

    const expire_ticks = timq_time.ticksFromMs(input_params.expire_time);
    const reload_ticks = timq_time.ticksFromMs(input_params.reload_time);

```

```

const tdev = &timer_queues[inst];
var element_id: DevSvcInstance = undefined;

//BEGIN INTERRUPT SECTION
{
    tdev.dev_control.disableCompareInterrupt();
    defer tdev.dev_control.enableCompareRegister();

    var element = tdev.storage.allocate() catch return .operation_failed;
    element_id = element.element_id; ①
    element.expire_time = expire_ticks;
    element.reload_time = reload_ticks;
    element.notify_closure = input_params.notify_closure;

    tdev.stop();
    tdev.queue.insert(element);
    tdev.start();
}
//END INTERRUPT SECTION

const CreateOutputParam = svc_services.timer_queue.CreateOutputParam;
return CreateOutputParam.send(.{
    .element_id = element_id,
}, output.?);
}

```

- ① If the expire time for the element is zero, it will be expired when the queue is started with `tdev.start()`. The expiration causes the element to be deleted. We take the `element_id` value out now, before the element is potentially deleted, in which case access to the element is undefined.

Delete Timer Element

```

<<timer queue services>>=
/// The `delete` function removes the timer queue element given by `element_id` from
/// the timer queue given by `timq`.
pub fn delete(
    /// The instance number which identifies which timer queue from which a timing
    /// element is removed.
    timq: DevSvcInstance,
    /// An identifier of a timer queue element. This value must be the same as
    /// the return value from a successful invocation of `create`.
    element_id: ElementId,
) SvcError!void {
    if (timq >= instances) return SvcError.unknown_instance;
    const input_params = DeleteInputParam.marshal(.{
        .element_id = element_id,
    });

    const request = TimqOperations.makeRequest(.delete, timq);
    return devRealmSvcCall(request, @ptrCast(&input_params), null, null);
}

```

The input to the remove request requires passing the element ID for the element to be removed.

```
<<timer queue services>>
pub const DeleteInputParam = DefineSvcRequestParam(struct {
    element_id: ElementId,
});
```

Delete Timer Element Proxy

Similar to the create operation, the foreground proxy function is a wrapper around the delete operation. Here the sequence of timer queue operations is stop, delete, start.

```
<<timer queue proxies>>
fn delete(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const inst = req.instance;
    if (!systimer_allocator.isAllocated(inst)) return .operation_failed;

    const DeleteInputParam = svc_services.timer_queue.DeleteInputParam;
    const input_params, const result = DeleteInputParam.receive(input.?);
    if (result != .success) return result;

    const tdev = &timer_queues[inst];

//BEGIN INTERRUPT SECTION
{
    tdev.dev_control.disableCompareInterrupt();
    defer tdev.dev_control.enableCompareInterrupt();

    tdev.stop();
    defer tdev.start();

    tdev.delete(input_params.element_id) catch return .operation_failed;
}
//END INTERRUPT SECTION

    return .success;
}
```

Update Timer Element

```
<<timer queue services>>
/// The `update` function modifies the timing parameters of the timer
/// queue element given by, `element_id`, which resides in the timer queue given by
/// `timq`. This function performs the logical equivalent of removing the element
/// from the timer queue and re-inserting the same element with new `expire_time`
/// and `reload_time` values. Note, it is not allowed to update the `reload_time`
```

```

/// of a queue element to a non-zero value if its current value is zero,
/// i.e. the `update` function may *not* be used to convert a one-shot queue element
/// to a recurring queue element. The converse is also true. It is not allowed to
/// update the `reload_time` of a queue element to be zero if it is currently non-zero,
/// i.e. the `update` function may *not* be used to convert a recurring queue element
/// into a one-shot queue element. Both of these cases require removing the old element
/// and inserting a new element.
pub fn update(
    /// The instance number which identifies the timer queue in which the element
    /// to be updated resides.
    timq: DevSvcInstance,
    /// An identifier of a timer queue element. This value must be the same as
    /// some return value from a successful invocation of `insert`.
    element_id: ElementId,
    /// The minimum number of milliseconds which must elapse before an expiration
    /// notification is sent.
    expire_time: u32,
    /// The minimum number of milliseconds which must elapse on the second and
    /// subsequent periods before an expiration notification is sent.
    reload_time: u32,
) SvcError!void {
    if (timq >= instances) return SvcError.unknown_instance;
    const input_params = UpdateInputParam.marshal(.{
        .element_id = element_id,
        .expire_time = expire_time,
        .reload_time = reload_time,
    });

    const request = TimqOperations.makeRequest(.update, timq);
    return devRealmSvcCall(request, @ptrCast(&input_params), null, null);
}

```

The input to the remove request requires passing the element ID for the element to be removed.

```

<<timer queue services>>=
pub const UpdateInputParam = DefineSvcRequestParam(struct{
    element_id: ElementId,
    expire_time: u32,
    reload_time: u32,
});

```

Update Timer Element Proxy

Because the timer queue is held in time relative order, one cannot just modify the queue element *in situ*. To maintain the correct time order of the timer queue, the foreground proxy function for updating a timer element must treat the update as the sequence: stop, remove, insert, and start.

```

<<timer queue proxies>>=
fn update(
    req: DevSvcRequest,
    input: ?*const anyopaque,

```

```

    _ : ?*anyopaque,
    _ : ?*anyopaque,
) SvcResult {
    const inst = req.instance;
    if (!sysTimer_allocator.isAllocated(inst)) return .operation_failed;

    const UpdateInputParam = svc_services.timer_queue.UpdateInputParam;
    const input_params, const result = UpdateInputParam.receive(input.?);
    if (result != .success) return result;

    const expire_ticks = timq_time.ticksFromMs(input_params.expire_time);
    const reload_ticks = timq_time.ticksFromMs(input_params.reload_time);

    const tdev = &timer_queues[inst];

//BEGIN INTERRUPT SECTION
{
    tdev.dev_control.disableCompareInterrupt();
    defer tdev.dev_control.enableCompareInterrupt();

    const element_id = input_params.element_id;
    const element = tdev.storage.elementFromId(element_id) catch return
.operation_failed;
    if ((element.reload_time == 0 and reload_ticks != 0) or
        (element.reload_time != 0 and reload_ticks == 0))
    {
        return .operation_failed;
    }

    tdev.stop();
    tdev.queue.remove(element);
    element.expire_time = expire_ticks;
    element.reload_time = reload_ticks;
    tdev.queue.insert(element);
    tdev.start();
}
//END INTERRUPT SECTION

return .success;
}

```

Remaining Time of a Timer Element

```

<<timer queue services>>=
/// The `remaining` function returns the minimum number of milliseconds which must elapse
/// before a notification for `element_id` is sent. An error is returned if the
/// timer queue element does not reside in the timer queue.
pub fn remaining(
    /// The instance number which identifies which timer queue that contains the
    /// element whose remaining time is obtained.
    timq: DevSvcInstance,
    /// An identifier of a timer queue element. This value must be the same as

```

```

    /// some return value from a successful invocation of `insert`.
    element_id: ElementId,
) SvcError!u32 {
    if (timq >= instances) return SvcError.unknown_instance;
    const input_params = RemainingInputParam.marshal(.{
        .element_id = element_id,
    });
    var output_params: RemainingOutputParam = .initial;

    const request = TimqOperations.makeRequest(.remaining, timq);
    try devRealmSvcCall(request, @ptrCast(&input_params), @ptrCast(&output_params), null);

    const output = output_params.unmarshal();
    return output.remaining;
}

```

```

<<timer queue services>>=
pub const RemainingInputParam = DefineSvcRequestParam(struct {
    element_id: ElementId,
});

```

```

<<timer queue services>>=
pub const RemainingOutputParam = DefineSvcRequestParam(struct {
    remaining: u32,
});

```

Remaining Time of a Timer Element Proxy

Finally, the foreground proxy for determining the amount of remaining time wraps the queue operations of stop, remaining, and start.

```

<<timer queue proxies>>=
fn remaining(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    output: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const inst = req.instance;
    if (!systimer_allocator.isAllocated(inst)) return .operation_failed;

    const RemainingInputParam = svc_services.timer_queue.RemainingInputParam;
    const input_params, const result = RemainingInputParam.receive(input.?);
    if (result != .success) return result;

    const tdev = &timer_queues[inst];
    var remain: timq_time.Ticks = 0;

//BEGIN INTERRUPT SECTION
{

```

```

tdev.dev_control.disableCompareInterrupt();
defer tdev.dev_control.enableCompareInterrupt();

const element_id = input_params.element_id;
if (!tdev.storage.isAllocated(element_id)) return .operation_failed;

tdev.stop();
remain = tdev.queue.remainingTime(element_id);
tdev.start();
}
//END INTERRUPT SECTION

const RemainingOutputParam = svc_services.timer_queue.RemainingOutputParam;
return RemainingOutputParam.send(.{
    .remaining = timq_time.msFromTicks(remain),
}, output.?);
}

```

Time Conversion

The time units used for the timer queue requests are device units. In this case the system timer device is configured to count as an unsigned UQ17.15 binary point numbers in units of seconds. The counter is clocked at 32 KiHz so that each timer tick represents approximately 30.5 μ s. The maximum value for an UQ17.15 number is approximately 131,072 seconds or approximately 36.4 hours.

The following namespace contains functions are useful for converting between device units and more conventional time units.

```

<<timer queue proxies>>=
pub const timq_time = struct{
    pub const Ticks = u32;

    const tick_int_bits: u5 = 17;
    const tick_frac_bits: u4 = 15;

    pub fn ticksFromMs(
        ms: u32,
    ) Ticks {
        const ticks: u64 = (@as(u64, ms) << tick_frac_bits) +
            @as(u64, time.ms_per_s) / 2) / @as(u64, time.ms_per_s);
        return math.lossyCast(Ticks, ticks);
    }

    pub fn msFromTicks(
        ticks: Ticks,
    ) u32 {
        const ms: u64 = ((@as(u64, ticks) * @as(u64, 1000)) +
            (math.maxInt(meta.Int(.unsigned, tick_frac_bits - 1)))) >> tick_frac_bits;
        return math.lossyCast(u32, ms);
    }
};

```

Dispatching Timer Queue Requests

In keeping with the device foreground proxy design, a class level handler is used to dispatch background requests to the appropriate proxy that handles the request.

```
<<svc device classes>>=
timq,
```

```
<<dev svc service request prongs>>=
.timq => &timer_queue.devClassHandler,
```

Finally, the Timer Queue class handler performs the dispatch to the operation functions of requests.

```
<<timer queue proxies>>=
/// The `devClassHandler` function dispatches timer queue operations based
/// on the value of the `req` argument. This function is invoked as a
/// class level dispatcher for the timer queue device.
fn devClassHandler(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    output: ?*anyopaque,
    err: ?*anyopaque,
) SvcResult {
    const Operation = svc_services.timer_queue.TimqOperations.Operation;
    const timq_operation: Operation = @enumFromInt(req.operation);
    const proxy = switch (timq_operation) {
        .open => &open,
        .close => &close,
        .create => &create,
        .delete => &delete,
        .update => &update,
        .remaining => &remaining,
    };
    return proxy(req, input, output, err);
}
```

Background Notifications

Each timer queue element is treated as a device instance with a given timer queue. This gives each element a small, zero-based, sequential integer value that is an identifier for the element. The element identifier is only unique with in the context of a given timer queue instance. Background processing is expected to use these identifiers when requesting timer queue element operations.

```
<<timer queue services>>=
pub const ElementId = DevSvcInstance;
```

The values of type `ElementId` are used as array indices into the storage pool from which the element was allocated. This is a simple and computationally cheap scheme which avoids using a pointer value as an identifier (don't want to expose privileged memory addresses to unprivileged background processing) or having to construct some other mapping between the queue element

and an external identifier.

A separate notification is sent for each queue element. The notification denotes changes in status for the element. The status changes supported are:

- a. The timer element has expired, *i.e.* at least the amount of time requested when the element was created has elapsed.
- b. The timer element has expired and was reloaded to expire again in the future. This is the case when the timer element has a non-zero reload time.
- c. The timer element was removed as a side effect of closing the timer queue instance. This notification ensures that queue elements do not mysteriously disappear with no indication.

```
<<timer queue services>>=
pub const NotifyReason = enum {
    expired,
    reloaded,
    discarded,
};
```

The information carried in the background notification both identifies the timer queue element (as the instance field of the notification) and gives the reason for the notification on the element.

```
<<dev notification specs>>=
.{ "timq", timer_queue.NotifyReason },
```

Timer Compare Register IRQ Handling

The last part of timer queue operations is handling the interrupt requests from the system timer compare registers which time the queue elements. As stated previously, there is a separate IRQ vector for each compare register.

```
<<timer device control operations>>=
export fn stimerCmpr0IrqHandler () void {
    irqHandler(0);
}
```

```
<<timer device control operations>>=
export fn stimerCmpr1IrqHandler () void {
    irqHandler(1);
}
```

The code for handling the IRQ's is factored into a common function that is invoked with the timer queue instance number. In terms of basic queue operations, the IRQ handler is required to expire those elements which have timed out and restart the timer queue for any elements which remain in the queue.

```
<<timer device control operations>>=
fn irqHandler(
    inst: DevSvcInstance,
```

```

) void {
    const tdev = &timer_queues[inst];
    tdev.start(); ①
}

```

- ① When the compare register interrupt happens, the queue element at the head of the queue will be zero. Since starting the timer queue expires any elements with zero expiration time, that is all the required work. There is no possibility of preemption here by requests coming from the background since they run at the lower SVC exception priority. Given the flat IRQ priority scheme, there is no preemption from other interrupts. In other words, once here, running to completion is assured.

Testing

Test execution synchronization

Before describing the test cases, a particular issue arises when testing code that is fundamentally asynchronous using a framework that assumes sequential, synchronous execution of the test cases.

However, to complete a test may involve asynchronous execution. For example, if the test inserts an element in the timer queue, it must wait for the time to expire, the IRQ to run, the background notification to be posted, and the notification to be dispatched before the status of the test can be determined.

To accomplish the synchronization between the foreground and background, the `exec_control.runWait` function, seen previously, is used. To use the busy/wait loop, a convention is used that passes a pointer to a boolean as the *closure* data for a device request and the notification function writes to the *terminate* variable via the closure pointer. This causes the loop to exit and the test appears to have simply executed sequential code.

For the test cases below, the timer queue notification function writes to the *terminate* variable by using a pointer to the *terminate* variable that was passed in the `DevNotification` value in the `timer_queue.insert` request. Recall that closure parameter type is such that variables of that type can hold a pointer value.

First, the variable that is used in the synchronization must be defined.

```

<<timq-test: declarations>>
var expiration_done: bool = false;

```

The background notification proxy function both records the notification information and writes to `expiration_done`.

```

<<timq-test: declarations>>
const Expiration = struct {
    notify: DevNotification,
    time: i64,
};

var expiration: Expiration = undefined;

```

```

<<timq-test: declarations>>

```

```

fn expired_proxy(
    notification: *const anyopaque,
) void {
    const notify: *const DevNotification = @ptrCast(@alignCast(notification));
    expiration.notify = notify.*;
    expiration.time = epoch_time.microTimestamp();
    const done: *volatile bool = @ptrFromInt(notify.closure); ①
    done.* = true;

    switch(notify.params.timq) {
        .expired => log.info("timer element {d} expired", .{notify.instance}),
        .reloaded => log.info("timer element {d} reloaded", .{notify.instance}),
        .discarded => log.info("timer element {d} discarded", .{notify.instance}),
    }
}

```

- ① This is where the closure value is used as a pointer to a boolean in order to synchronize to the sequentially running test case.

It is worth reiterating the execution sequence which synchronizes the execution of test cases with the asynchronous actions of timer queue elements. This technique is used in several places.

1. The `timer_queue.open` function establishes `expired_proxy` as the background proxy function. A pointer to `expired_proxy` is placed with the notification when a timer element expires.
2. The `timer_queue.create` function takes as an argument an item of `closure` data which is placed into the background notification when the inserted element expires. The value of the closure is the address of `expiration_done`, the variable which is monitored as a termination request.
3. A call to `exec_control.runWait` enters an *event loop* where background notifications are dispatched. The `exec_control.runWait` function checks the `terminate` variable, `expiration_done` in this case, to determine if one of the dispatched notification function changed its value to `true`.
4. If there are no notifications, the processor is put to sleep.
5. When the system timer compare register matches, an IRQ is pended and the IRQ handler runs. The IRQ handler places notifications in the background notification queue for any expired timer elements. That notification contains the `expired_proxy` notification function pointer along with the closure data, which in this case is a pointer to `expiration_done`. The interrupt causes the processor core to wake up.
6. Then, `exec_control.runWait`, as part of its event loop, dispatches `expired_proxy`, which in turn sets `expiration_done` to `true`.
7. After `exec_control.runWait` returns from dispatching the background notifications, it determines that the `expiration_done` variable has been changed to `true` and returns to its caller.

This may seem quite involved just for running a test case. However, this mechanism is at the heart of how background processing operates and can be used in an interactive situation to give the appearance that the test case function executed synchronously.

Test cases

```

<<timer_queue_unit_tests.zig>>
<<edit warning>>

```

```

<<copyright info>>

const std = @import("std");
const log = std.log;
const time = std.time;
const testing = @import("resee_testing");

const svc_services = @import("svc_services");
const timer_queue = svc_services.timer_queue;
const epoch_time = svc_services.epoch_time;
const DevNotification = svc_services.DevNotification;
const exec_control = svc_services.exec_control;

<<timq-test: declarations>>
<<timq-test: tests>>

```

```

<<timq-test: tests>>
test "open timer queue" {
    try timer_queue.open(0, expired_proxy);
}

```

```

<<timq-test: tests>>
test "remove element from empty timer queue" {
    const status = timer_queue.delete(0, 0);
    try testing.expectError(error.operation_failed, status);
}

```

```

<<timq-test: tests>>
test "expire a timer element after 1 second" {
    testing.log_level = .info;

    const insert_time = epoch_time.microTimestamp();
    const elemId = try timer_queue.create(0, 1000, 0, @intFromPtr(&expiration_done));
    try testing.expect(elemId >= 0);

    try exec_control.runWait(&expiration_done);
    try testing.expectEqual(true, expiration_done);
    try testing.expectEqual(elemId, expiration.notify.instance);

    log.info("element {d} inserted: {d} us", .{ elemId, insert_time }) ;
    log.info("element {d} expired: {d} us", .{ elemId, expiration.time }) ;
    const elapsed_time = @as(f32, @floatFromInt(expiration.time - insert_time)) / 1000.0;
    log.info("elapsed time: {d:.3} ms", .{elapsed_time});
    try testing.expect(elapsed_time >= 1000.0 and elapsed_time <= 1000.3);
}

```

```

<<timq-test: tests>>
test "insert a zero length expiration" {
    testing.log_level = .info;
}

```

```

const insert_time = epoch_time.microTimestamp();
const elemId = try timer_queue.create(0, 0, 0, @intFromPtr(&expiration_done));
try testing.expect(elemId >= 0);

// Since the timer expiration is immediate, we handle fetching it without
// entering the event loop.
try exec_control.dispatchNotifications();
try testing.expectEqual(elemId, expiration.notify.instance);

log.info("element {d} inserted: {d} us", .{ elemId, insert_time }) ;
log.info("element {d} expired: {d} us", .{ elemId, expiration.time }) ;
const elapsed_time = @as(f32, @floatFromInt(expiration.time - insert_time)) / 1000.0;
log.info("elapsed time: {d:.3} ms", .{elapsed_time});
try testing.expect(elapsed_time >= 0.0 and elapsed_time <= 1.0);
}

```

```

<<timq-test: tests>>=
test "expire after 1 ms" {
    testing.log_level = .info;

    const insert_time = epoch_time.microTimestamp();
    const elemId = try timer_queue.create(0, 1, 0, @intFromPtr(&expiration_done));
    try testing.expect(elemId >= 0);

    try exec_control.runWait(&expiration_done);
    try testing.expectEqual(true, expiration_done);
    try testing.expectEqual(elemId, expiration.notify.instance);

    log.info("element {d} inserted: {d} us", .{ elemId, insert_time }) ;
    log.info("element {d} expired: {d} us", .{ elemId, expiration.time }) ;
    const elapsed_time = @as(f32, @floatFromInt(expiration.time - insert_time)) / 1000.0;
    log.info("elapsed time: {d:.3} ms", .{elapsed_time});
    try testing.expect(elapsed_time >= 1.0 and elapsed_time <= 1.3);
}

```

```

<<timq-test: tests>>=
test "insert larger time before smaller time" {
    testing.log_level = .info;

    const insert_time = epoch_time.microTimestamp();
    const tmr_1000 = try timer_queue.create(0, 1000, 0, @intFromPtr(&expiration_done));
    const tmr_500 = try timer_queue.create(0, 500, 0, @intFromPtr(&expiration_done));

    try testing.expect(tmr_1000 >= 0);
    try testing.expect(tmr_500 >= 0);

    try exec_control.runWait(&expiration_done);
    const tmr_500_expired = expiration.time;
    try testing.expectEqual(true, expiration_done);
    try testing.expectEqual(tmr_500, expiration.notify.instance);
}

```

```

try exec_control.runWait(&expiration_done);
const tmr_1000_expired = expiration.time;
try testing.expectEqual(true, expiration_done);
try testing.expectEqual(tmr_1000, expiration.notify.instance);

log.info("elements inserted: {d} us", .{ insert_time }) ;
log.info("element {d} expired: {d} us", .{ tmr_500, tmr_500_expired }) ;
log.info("element {d} expired: {d} us", .{ tmr_1000, tmr_1000_expired }) ;

const tmr_500_elapsed = @as(f32, @floatFromInt(tmr_500_expired - insert_time)) / 
1000.0;
log.info("500 ms elapsed time: {d:.3} ms", .{tmr_500_elapsed});
try testing.expect(tmr_500_elapsed >= 0.0 and tmr_500_elapsed <= 500.4);

const tmr_1000_elapsed = @as(f32, @floatFromInt(tmr_1000_expired - insert_time)) / 
1000.0;
log.info("1000 ms elapsed time: {d:.3} ms", .{tmr_1000_elapsed});
try testing.expect(tmr_1000_elapsed >= 0.0 and tmr_1000_elapsed <= 1000.4);
}

```

```

<<timq-test: tests>>=
test "remaining time after one element has expired" {
    testing.log_level = .info;

    const insert_time = epoch_time.microTimestamp();
    const tmr_1172 = try timer_queue.create(0, 1172, 0, @intFromPtr(&expiration_done));
    const tmr_250 = try timer_queue.create(0, 250, 0, @intFromPtr(&expiration_done));

    try testing.expect(tmr_1172 >= 0);
    try testing.expect(tmr_250 >= 0);

    try exec_control.runWait(&expiration_done);
    const tmr_250_expired = expiration.time;
    try testing.expectEqual(true, expiration_done);
    try testing.expectEqual(tmr_250, expiration.notify.instance);

    const remaining = try timer_queue.remaining(0, tmr_1172);

    try exec_control.runWait(&expiration_done);
    const tmr_1172_expired = expiration.time;
    try testing.expectEqual(true, expiration_done);
    try testing.expectEqual(tmr_1172, expiration.notify.instance);

    log.info("elements inserted: {d} us", .{insert_time}) ;
    log.info("element {d} expired: {d} us", .{ tmr_250, tmr_250_expired }) ;
    const tmr_250_elapsed = @as(f32, @floatFromInt(tmr_250_expired - insert_time)) / 
1000.0;
    log.info("element {d} elapsed time: {d:.3} ms", .{ tmr_250, tmr_250_elapsed }) ;
    try testing.expect(tmr_250_elapsed >= 0.0 and tmr_250_elapsed <= 250.5);

    log.info("element {d} expired: {d} us", .{ tmr_1172, tmr_1172_expired }) ;
    const tmr_1172_elapsed = @as(f32, @floatFromInt(tmr_1172_expired - insert_time)) /

```

```

1000.0;
log.info("element {d} elapsed time: {d:.3} ms", .{ tmr_1172, tmr_1172_elapsed });
try testing.expect(tmr_1172_elapsed >= 0.0 and tmr_1172_elapsed <= 1172.5);

log.info("remaining time after first expiration: {d} ms", .{remaining}) ;
try testing.expect(remaining >= 920 and remaining < 923);
}

```

```

<<timq-test: tests>>=
test "update a larger expiration to be smaller than an existing one" {
    testing.log_level = .info;

    const insert_time_1000 = epoch_time.microTimestamp();
    const tmr_1000 = try timer_queue.create(0, 1000, 0, @intFromPtr(&expiration_done));
    const insert_time_500 = epoch_time.microTimestamp();
    const tmr_500 = try timer_queue.create(0, 500, 0, @intFromPtr(&expiration_done));
    try testing.expect(tmr_1000 >= 0);
    try testing.expect(tmr_500 >= 0);

    try timer_queue.update(0, tmr_1000, 250, 0);

    try exec_control.runWait(&expiration_done);
    const tmr_1000_expired = expiration.time;
    try testing.expectEqual(true, expiration_done);
    try testing.expectEqual(tmr_1000, expiration.notify.instance);

    try exec_control.runWait(&expiration_done);
    const tmr_500_expired = expiration.time;
    try testing.expectEqual(true, expiration_done);
    try testing.expectEqual(tmr_500, expiration.notify.instance);

    log.info("element {d} expired: {d} us", .{ tmr_1000, tmr_1000_expired }) ;
    const tmr_1000_elapsed = @as(f32, @floatFromInt(tmr_1000_expired - insert_time_1000)) / 1000.0;
    log.info("element {d} elapsed time: {d:.3} ms", .{ tmr_1000, tmr_1000_elapsed }) ;
    try testing.expect(tmr_1000_elapsed >= 0.0 and tmr_1000_elapsed <= 250.6);

    log.info("element {d} expired: {d} us", .{ tmr_500, tmr_500_expired }) ;
    const tmr_500_elapsed = @as(f32, @floatFromInt(tmr_500_expired - insert_time_500)) / 1000.0;
    log.info("element {d} elapsed time: {d:.3} ms", .{ tmr_500, tmr_500_elapsed }) ;
    try testing.expect(tmr_500_elapsed >= 0.0 and tmr_500_elapsed <= 500.6);
}

```

```

<<timq-test: tests>>=
test "series of one second periodic expirations" {
    testing.log_level = .info;

    var insert_time = epoch_time.microTimestamp();
    const tmr_1000 = try timer_queue.create(0, 1000, 1000, @intFromPtr(&expiration_done));
    try testing.expect(tmr_1000 >= 0);

```

```

for (0..4) |_| {
    try exec_control.runWait(&expiration_done);
    try testing.expectEqual(true, expiration_done);
    try testing.expectEqual(tmr_1000, expiration.notify.instance);

    const expired_time = expiration.time;

    log.info("element {d} expired: {d} us", .{ tmr_1000, expired_time }) ;
    const tmr_1000_elapsed = @as(f32, @floatFromInt(expired_time - insert_time)) /
1000.0;
    log.info("element {d} elapsed time: {d:.3} ms", .{ tmr_1000, tmr_1000_elapsed });
    try testing.expect(tmr_1000_elapsed >= 0.0 and tmr_1000_elapsed <= 1000.4);

    insert_time = expired_time;
}

try timer_queue.delete(0, tmr_1000);
}

```

```

<<timq-test: tests>>=
test "close timer queue" {
    try timer_queue.close(0);
}

```

Blinky

In the microcontroller world, a program to blink an LED carries a similar status as the *Hello, world!* program in the conventional timeshare world. There is an LED on the MicroMod processor board. The built-in LED is attached to GPIO pin number 19.

The usual blinking LED program distributed in SDK's as an example uses blocking timing to alternate turning on and then off the LED. Since we have not blocking mechanisms, this implementation is quite different. In this design, the timer queue requests just developed [above](#) are used.

The main program is structured as a simple command interface that uses single letters as commands, possibly followed by parameters. The ? command gives a usage response. The interaction is via the system console presented in the last chapter. The commands allow using both the one-shot and recurring timer queue features. Both the GPIO driving the LED pin and a timer queue element are set up and managed by the program. Just to demonstrate the asynchronous nature of the environment, the watchdog timer is also set up and maintained without needing any user intervention.

Note the command interface is very unforgiving and will exit `main` causing a panic condition on any input it does not recognize.

```

<<blinky-test.zig>>=
//! This file contains a simple test application demonstrating the use
//! of the timer queue to blink an LED.
<<edit warning>>
<<copyright info>>

```

```

comptime {
    _ = @import("start_main");
}

const std = @import("std");
const mem = std.mem;
const log = std.log;

const config = @import("config");

const svc_services = @import("svc_services");
const gpio = svc_services.GPIO;
const timer_queue = svc_services.timer_queue;
const DevNotification = svc_services.DevNotification;

pub const panic = svc_services.panic;
pub const std_options: std.Options = .{
    .logFn = svc_services.logToConsole,
};

pub fn main() !void {
    // command line interaction is via the system console
    svc_services.Console.openAllConsoles();

    try app_wdog.start();
    try app_led.start();
    try app_timer.start();

    const sys_console = &svc_services.Console.sys_console;
    var sys_writer = sys_console.writer(&.{});
    const sysOut = sys_writer.interface();

    var reader_buf: [256]u8 = undefined;
    var sys_reader = sys_console.reader(&reader_buf);
    const sysIn = sys_reader.interface();

    const usage =
        \usage:
        \c <time>           // create one-shot timer with <time> (ms)
        \d                   // delete timer
        \r <time>           // create recurring timer with <time> (ms)
        \l <time>           // update timer reload with <time> (ms)
        \m                 // print the remaining time (ms) of the timer
        \x                 // exit main with an error
        \?
        ++ "\n";
}

while (true) {
    try sysOut.writeAll("All your codebase are belong to us.\n# ");

    const command = try sysIn.takeDelimiterExclusive('\n');
    if (command.len == 0) continue;
}

```

```

var cmd_iter = mem.tokenizeAny(u8, command, " \t");
const cmd_letter = cmd_iter.next().?[0];
const next_arg = cmd_iter.next();

switch (cmd_letter) {
    'c' => {
        if (next_arg |time_arg| {
            if (std.fmt.parseUnsigned(usize, time_arg, 0)) |time| {
                try app_timer.create(time);
            } else |err| {
                try sysOut.print("failed to parse '{s}' as an unsigned: '{s}'\n",
.
{
                time_arg,
                @errorName(err),
            });
        }
    } else {
        try sysOut.writeAll(usage);
    }
},
'd' => {
    try app_timer.delete();
    try app_led.off();
},
'r' => {
    if (next_arg |time_arg| {
        if (std.fmt.parseUnsigned(usize, time_arg, 0)) |time| {
            try app_timer.recurring(time);
        } else |err| {
            try sysOut.print("failed to parse '{s}' as an unsigned: '{s}'\n",
.
{
                time_arg,
                @errorName(err),
            });
        }
    } else {
        try sysOut.writeAll(usage);
    }
},
'l' => {
    if (next_arg |time_arg| {
        if (std.fmt.parseUnsigned(usize, time_arg, 0)) |time| {
            try app_timer.update(time);
        } else |err| {
            try sysOut.print("failed to parse '{s}' as an unsigned: '{s}'\n",
.
{
                time_arg,
                @errorName(err),
            });
        }
    } else {
        try sysOut.writeAll(usage);
    }
}
}

```

```

        }
    },
    'm' => {
        const remaining = try app_timer.remaining();
        try sysOut.print("{d}\n", .{remaining});
    },
    'x' => {
        log.warn("exiting main", .{});
        return error.exit;
    },
    '?' => try sysOut.writeAll(usage),
    else => {
        log.err("unrecognized command '{c}'", .{cmd_letter});
        try sysOut.print("unrecognized command '{c}'\n", .{cmd_letter});
        try sysOut.writeAll(usage);
    },
},
}

try app_led.off();
try app_timer.stop();
}

/// The following namespaces group together the required elements to
/// specialize the device interactions to the application requirements.
/// This includes both device interactions as well as synchronization functions.
const app_timer = struct {
    const timq_inst: svc_services.DevSvcInstance = 0;
    var element_id: timer_queue.ElementId = undefined;

    fn start() !void {
        try timer_queue.open(timq_inst, timerReloadProxy);
    }

    fn stop() !void {
        try timer_queue.delete(timq_inst, element_id);
        try timer_queue.close(timq_inst);
    }

    fn create(
        time: u32,
    ) !void {
        element_id = try timer_queue.create(timq_inst, time, 0, 0);
    }

    fn recurring(
        time: u32,
    ) !void {
        element_id = try timer_queue.create(timq_inst, time, time, 0);
    }

    fn delete() !void {

```

```

        try timer_queue.delete(timq_inst, element_id);
    }

fn remaining() !u32 {
    return try timer_queue.remaining(timq_inst, element_id);
}

fn update(
    reload_time: u32, // in ms
) !void {
    try timer_queue.update(timq_inst, element_id, reload_time, reload_time);
}

fn timerReloadProxy(
    notification: *const anyopaque,
) void {
    const notify: *const DevNotification = @ptrCast(@alignCast(notification));
    switch (notify.params.timq) {
        .expired, .reloaded => app_led.toggle() catch unreachable,
        else => unreachable,
    }
}
};

const app_led = struct {
    const builtin_led_pin: svc_services.DevSvcInstance = config.blinky_pin;

    fn start() !void {
        try gpio.outputConfig(
            builtin_led_pin, // pin number
            .push_pull, // output type
            .@"12mA", // drive strength
            .none, // pull up resistance
        );
    }

    fn toggle() !void {
        try gpio.write(builtin_led_pin, .pin_toggle);
    }

    fn off() !void {
        try gpio.write(builtin_led_pin, .pin_clear);
    }
};

const app_wdog = struct {
    fn start() !void {
        try svc_services.wdog.init(0, 4000, 3500, wdogDevNotifier, 0);
        try svc_services.wdog.start(0, true); // allow watchdog to reset
    }

    fn wdogDevNotifier(
        notification: *const anyopaque,

```

```

    ) void {
        const wdogNotification: *const DevNotification = @ptrCast(@alignCast(
notification));
        svc_services.wdog.restart(wdogNotification.instance) catch
            @panic("Failed to restart watchdog timer");
    }
};

```

Epoch Time

Time is so important almost all computers have some means of counting the pulses of a fixed frequency clock. This can be used to keep track of calendar time by:

- picking an arbitrary date, known as the *epoch*,
- fixing that date to be the zero of time, and
- incrementing a counter based on a fixed frequency clock pulse.

In this way, a counter, plus the implied side agreement as to the what calendar time was chosen for the epoch, yields a relatively simple way to keep track of calendar time by relating the count value back to conventional human time concepts. One common epoch date is, 1970-01-01 00:00:00+0000 (GMT), which is used in UNIX derived systems. The same epoch date is adopted in this design.

In this chapter, the system timer peripheral is revisited and the ability to track epoch time using the system timer as a counter is added. Code for its initialization was given in [Chapter 2](#). The design strategy is to provide background requests to set and get the time of day in the form of the epoch count. When the count is set, the system timer peripheral is reset to zero. When the time is read, the system timer counter value is added to the last supplied epoch count. In other words, the system timer peripheral counter is used as an offset for the last set epoch time. This offset strategy allows for mapping between any differences in the units and resolution of the epoch count and the those of the system timer peripheral.

There is still a quandary. There is no authoritative source of the current time of day. That problem here is not solved here and it is assumed the current time is obtained from outside the system, perhaps using an external clock such as a GPS receiver or a battery backed RTC which has been previously set or even a [WWVB](#) clock. Given an epoch count from somewhere, the progress of time can be accurately tracked.

The interfaces to calendar time functions have been long established in the POSIX/UNIX-like world and they are also appropriate for this usage. Substitute implementations of some standard library functions that interface properly to the these specific time functions are provided.

Epoch time count

The system timer on the Apollo 3 survives normal resets and is only cleared at power up. Consequently, a place to store the base epoch time count that also survives reset is required. Fortunately, the `.noinit_bss` section, which was set aside previously, serves this purpose.

```

<<system_services: system service definitions>>=
/// Epoch time services
pub const epoch_time = struct {
    <<epoch time services>>

```

```
};
```

```
<<sys svc service requests>>=
set_epoch_time,
get_epoch_time,
```

```
<<sys svc service request prongs>>=
.set_epoch_time => &epoch_time.set,
.get_epoch_time => &epoch_time.get,
```

```
<<system proxies>>=
/// Epoch time service proxies
const epoch_time = struct {
    const EpochTimeValue = svc_services.epoch_time.EpochTimeValue;
    const SystemTick = svc_services.epoch_time.SystemTick;
    <<epoch time proxies>>
};
```

```
<<epoch time services>>=
pub const EpochTimeValue = i64;
```

```
<<epoch time proxies>>=
var epoch_base: EpochTimeValue = 0;
```

This value is typed as `i64` to match the current conventions used in the UNIX-like world and avoid any potential “[year 2038](#)” problems.

Time resolution

The clock used to drive the system timer on the Apollo3 runs a 32 KiHz. That is relatively slow by modern standards and the time resolution is approximately 30.5 μ s. We will deal with the epoch time and timestamps in units of μ s with the recognition that the resolution is *not* 1 μ s.

Setting epoch time

```
<<epoch time services>>=
pub fn set(
    etime: EpochTimeValue,
) SvcError!void {
    const input_params = SetInputParam.marshal(.{
        .epoch_time = etime,
    });
    return sysRealmSvcCall(.set_epoch_time, @ptrCast(&input_params), null, null);
}
```

```
<<epoch time services>>=
pub const SetInputParam = DefineSvcRequestParam(struct {
```

```

    epoch_time: EpochTimeValue,
});

```

The implementation of the request foreground proxy involves the following steps:

- Store the input time value into the epoch_base variable. The input is an i64 in units of μs .
- Zero the system timer counter registers. In this way, the system timer counter becomes an offset to be added to epoch_base giving the current time.

```

<<epoch time proxies>>=
fn set(
    _: SysSvcRequest,
    input: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const SetInputParam = svc_services.epoch_time.SetInputParam;
    const input_params, const status = SetInputParam.receive(input.?);
    if (status != .success) return status;

    apollo3.timer.st_cfg.setBitField(.freeze);
    apollo3.timer.st_cfg.setBitField(.clear);
    apollo3.timer.st_cfg.clearBitField(.clear);
    apollo3.timer.s_nvr[0] = 0; ①
    apollo3.timer.s_nvr[1] = 0;
    apollo3.timer.s_nvr[2] = 0;
    apollo3.timer.s_nvr[3] = 0;
    epoch_base = input_params.epoch_time;
    apollo3.timer.st_cfg.clearBitField(.freeze);
    return .success;
}

```

- ① Clear out all the non-volatile RAM location that hold the overflow counts from the system timer even though only one is used given the clock frequency of the system timer.

Getting epoch time

```

<<epoch time services>>=
pub fn get() SvcError!EpochTimeValue {
    var output_params: GetOutputParam = .initial;
    try sysRealmSvcCall(.get_epoch_time, null, @ptrCast(&output_params), null);
    return output_params.params.epoch_time;
}

```

```

<<epoch time services>>=
pub const GetOutputParam = DefineSvcRequestParam(struct {
    epoch_time: EpochTimeValue,
});

```

```

<<epoch time proxies>>=
fn get(
    _: SysSvcRequest,
    _: ?*const anyopaque,
    output: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const GetOutputParam = svc_services.epoch_time.GetOutputParam;
    return GetOutputParam.send(.{
        .epoch_time = deviceEpochTime(),
    }, output.?);
}

```

```

<<epoch time proxies>>=
fn deviceEpochTime() EpochTimeValue {
    const timer_value = apollo3.stimer.st_tmr;
    const timer_overflow = apollo3.stimer.s_nvr[0];
    const overflow_us = (@as(u63, timer_overflow) << 17) * time.us_per_s;
    //const rounding_adj = @as(u63, 1 << 14);
    //const ticks_us = ((@as(u63, timer_value) + rounding_adj) * time.us_per_s) >> 15;
    const ticks_us = (@as(u63, timer_value) * time.us_per_s) >> 15;
    return epoch_base + overflow_us + ticks_us;
}

```

```

<<epoch time services>>=
pub const SystemTick = packed struct(u32) {
    frac: u15,
    sec: u17,

    pub fn format(
        value: SystemTick,
        writer: *std.Io.Writer,
    ) std.Io.Writer.Error!void {
        try writer.print("{d}.{d:0>3}", .{ value.sec, value.frac });
    }

    pub fn epochTimeFromSystemTick(
        tick: SystemTick,
    ) EpochTimeValue {
        const frac_bits = @bitSizeOf(@TypeOf(@as(SystemTick, undefined).frac));
        return (@as(i64, tick.sec) * time.us_per_s) +
            ((@as(u63, tick.frac) * time.us_per_s) >> frac_bits);
    }
};

```

Timestamps

We provide a few timestamp services for 1 second, 1 millisecond, and 1 microsecond resolution.

```

<<epoch time services>>=
pub fn timestamp() i64 {
    return math.divCeil(i64, milliTimestamp(), time.ms_per_s) catch unreachable;
}

pub fn milliTimestamp() i64 {
    return math.divCeil(i64, microTimestamp(), time.us_per_ms) catch unreachable;
}

pub fn microTimestamp() i64 {
    return get() catch unreachable;
}

```

Testing

```

<<epoch_time_unit_tests.zig>>=
<<edit warning>>
<<copyright info>>

const std = @import("std");
const log = std.log;
const time = std.time;
const testing = @import("resee_testing");

const svc_services = @import("svc_services");
const epoch_time = svc_services.epoch_time;
const SystemTick = epoch_time.SystemTick;
const EpochTimeValue = epoch_time.EpochTimeValue;
const rtc = svc_services.rtc;
const DevNotification = svc_services.DevNotification;
const exec_control = svc_services.exec_control;

fn alarmExpired(
    notify: *const anyopaque,
) void {
    const notification: *const DevNotification = @ptrCast(@alignCast(notify));
    const done: *volatile bool = @ptrFromInt(notification.closure);
    done.* = true;

    log.info("alarm expired: {f}", .{notification.params.rtc.time});
}

test "system tick functions" {
    const ticks = [_]SystemTick{
        .{ .sec = 0, .frac = 0 },
        .{ .sec = 0, .frac = 0b100_000_000_000_000 }, // 1/2 as U.15
        .{ .sec = 0, .frac = 0b010_000_000_000_000 }, // 1/4 as U.15
        .{ .sec = 1, .frac = 0 },
    };
    try testing.expectEqual(@as(EpochTimeValue, 0), ticks[0].epochTimeFromSystemTick());
    try testing.expectEqual(@as(EpochTimeValue, time.us_per_s / 2), ticks[1]

```

```

].epochTimeFromSystemTick());
    try testing.expectEqual(@as(EpochTimeValue, time.us_per_s / 4), ticks[2]
].epochTimeFromSystemTick());
    try testing.expectEqual(@as(EpochTimeValue, time.us_per_s), ticks[3
].epochTimeFromSystemTick());
}

test "set epoch time" {
    try epoch_time.set(0);
    const current = epoch_time.get();
    try testing.expectEqual(@as(epoch_time.EpochTimeValue, 0), current);
}

test "one second polling" {
    testing.log_level = .debug;
    // Use RTC as a comparison timing source
    const current: rtc.RtcTime = .{
        .year = 2021,
        .month = 12,
        .date = 31,
        .hour = 23,
        .minute = 59,
        .second = 59,
        .hundredth = 0,
        .weekday = 2,
    };
    // Use RTC alarm for a 1 second tick
    const alarmAt: rtc.RtcAlarm = .{
        .repeat = .sec,
        .month = 0,
        .date = 0,
        .hour = 0,
        .minute = 0,
        .second = 0,
        .hundredth = 99,
        .weekday = 0,
    };
    var alarm_expired: bool = false;

    try rtc.set(0, &current);
    try rtc.setAlarm(0, &alarmAt, &alarmExpired, @intFromPtr(&alarm_expired));
    try exec_control.runWait(&alarm_expired); // discard first alarm to get synced up

    try epoch_time.set(0);
    for (1..10) |alarm_count| {
        try exec_control.runWait(&alarm_expired);
        const ts = epoch_time.milliTimestamp();
        try testing.expectEqual(alarm_count * time.ms_per_s, ts);
    }

    try rtc.cancelAlarm(0);
}

```

}

Human Time

Humans have had a keen interest in measuring time long before possessing adequate technology to make precise measurements. The further from the equator one lives the more extensive the variation of the exposure to the Sun, making the adaptations required for survival and predicting the astronomical cycles essential.

By the twentieth century CE, technology advanced sufficiently to conquer time. The Earth's rotation around the Sun now can be tracked with such precision as to observe small variabilities in the period. Atomic clocks are accurate enough to observe relativistic time warp even at the slow speeds of aircraft (relative to the speed of light).

The periods of astronomical bodies such as the Earth and Moon are not convenient multiples of each other. For example, there are not an integral number of rotations of the Earth on its axis within one revolution around the Sun. The rotation of the Moon around the Earth also does not align evenly with the rotation of the Earth around the Sun.

Such complications have lead to a long history of intricate rules for keeping time in a manner to which humans have become accustomed. Humans are adept at keeping track of such rules. For computers, keeping time in a manner convenient to human rules is a "fussy" undertaking. It is not uncommon for modern microcontrollers to have specialized peripherals which count time and perform the some of the modulus arithmetic associated with day, months, years, etc.

Overview of the RTC

The Apollo 3 has a Real Time Clock (RTC) peripheral which is capable of keeping track of time in human oriented units as well as signaling when the current time matches an alarm time. The Apollo 3 data sheet describes the details of the peripheral register interface. The RTC features used are:

- The RTC keeps time in a *broken out* format, i.e. there are separate register fields for each time component such as hour, minute, etc.
- The RTC is set to run in the 21st century only.
- The RTC has an alarm function which can signal a match for one given date and which can repeatedly signal matches on selected date fields. Access to the alarm and its repeat function are provided.
- The hours counter is run in so called, *24 hour*, mode as opposed to a 12 hour mode which uses an additional bit for AM or PM.

BCD Conversion

As is common for RTC of peripherals^[3], the time values in the peripheral registers are in Binary Coded Decimal (BCD) format. It is more convenient, outside of the device specific code, to deal with conventional 2's complement binary numbers.

RTC time value

The following structure is used to hold the values for the broken out time used by the RTC.

```
<<rtc services>>
pub const RtcTime = struct {
```

```

year: u16,
month: u8,
date: u8,
hour: u8,
minute: u8,
second: u8,
hundredth: u8,
weekday: u8,

pub fn format(
    value: RtcTime,
    writer: *std.Io.Writer,
) std.Io.Writer.Error!void {
    try writer.print("{d}-{d:0>2}-{d:0>2}T{d:0>2}:{d:0>2}:{d:0>2}.{d:0>3}D{d}", .{
        value.year,
        value.month,
        value.date,
        value.hour,
        value.minute,
        value.second,
        value.hundredth,
        value.weekday,
    });
}
};

```

The structure members correspond directly to the RTC peripheral register values, except for the `year` member. When dealing with an RTC time value in the application, a complete year value in the current era (CE) is used, e.g. 2022. In the RTC peripheral registers, there is only a single byte of BCD encoded year (which can hold 0 to 99) and the *century* is indicated by a separate bit. This use of the peripheral is only concerned with keeping time in the 2000's CE and the century bit is set accordingly. Before writing the year into the RTC register, century portion is subtracted off.

```

<<rtc proxies>>=
pub const century_base: u16 = 2000;

```

Reading the RTC time

The following function reads the RTC time from the hardware and returns the time in device units. Because the peripheral packs several components of the time into a single register, the code contains the necessary unpacking into the time structure.

```

<<device control methods>>=
fn readRtcTime(
    self: *const DevControl,
    rtc_time: *RtcTime,
) SvcResult {
    const upper: apollo3.RealTimeClock.CtrUpReg,
    const lower: apollo3.RealTimeClock.CtrLowReg,
    const status: SvcResult = self.readRtcCounters(); ①
}

```

```

    if (status != .success) return status;

    // Convert to binary here. There is not much that can be done
    // with BCD valued fields.
    rtc_time.weekday = upper.ctr_wkdy;
    rtc_time.year = (@as(u8, upper.ctr_yr.h) * 10 + upper.ctr_yr.l) + century_base;
    rtc_time.month = @as(u8, upper.ctr_mo.h) * 10 + upper.ctr_mo.l;
    rtc_time.date = @as(u8, upper.ctr_date.h) * 10 + upper.ctr_date.l;
    rtc_time.hour = @as(u8, lower.ctr_hr.h) * 10 + lower.ctr_hr.l;
    rtc_time.minute = @as(u8, lower.ctr_min.h) * 10 + lower.ctr_min.l;
    rtc_time.second = @as(u8, lower.ctr_sec.h) * 10 + lower.ctr_sec.l;
    rtc_time.hundredth = @as(u8, lower.ctr_100.h) * 10 + lower.ctr_100.l;

    return .success;
}

```

- ① Because there are retries involved, reading the RTC counters is factored into a separate function.

```

<<device control methods>>=
fn readRtcCounters(
    self: *const DevControl,
) struct {apollo3.RealTimeClock.CtrUpReg, apollo3.RealTimeClock.CtrLowReg, SvcResult} {
    const rtdev = self.device;
    var lower = rtdev.ctr_low.read();
    var upper = rtdev.ctr_up.read();
    var status = rtdev.ctr_up.readField(.ct_err);

    var retries: usize = 0;
    const max_retries: usize = 3;
    while (status == 1 and retries < max_retries) :
        (status = rtdev.ctr_up.readField(.ct_err)) ①
    {
        lower = rtdev.ctr_low.read();
        upper = rtdev.ctr_up.read();
        retries += 1;
    }

    return .{
        upper,
        lower,
        if (status == 0) .success else .operation_failed,
    };
}

```

- ① Reading the time from the RTC requires two register reads. The peripheral “freezes” the upper register value when the lower one is read. On the off chance that an interrupt occurs after the lower register read but before the upper register read and that interrupt executes long enough so that the 100th counter increments, the data between the two reads could be inconsistent. The peripheral diagnoses this situation using an error bit in the upper register to indicate that the read attempt was not successful. The retry count guards against the remote possibility of an infinite loop.

Updating the RTC time

The following function updates the time in the RTC peripheral. Time in this case is considered to be in device units. In this case the majority of the code takes the broken-out time representation and packs it into the device registers.

```
<<device control methods>>=
fn updateRtcTime(
    self: *const DevControl,
    rtc_time: *const RtcTime,
) SvcResult {
    if (builtin.mode == .Debug) {
        if (rtc_time.year < century_base or rtc_time.year >= century_base + 100) return
            .invalid_param;
        if (rtc_time.month < 1 or rtc_time.month > 12) return .invalid_param;
        if (rtc_time.date < 1 or rtc_time.date > 31) return .invalid_param;
        if (rtc_time.hour >= 24) return .invalid_param;
        if (rtc_time.minute >= 60) return .invalid_param;
        if (rtc_time.second >= 60) return .invalid_param;
        if (rtc_time.hundredth >= 100) return .invalid_param;
        if (rtc_time.weekday >= 7) return .invalid_param;
    }

    // Convert to BCD in preparation for writing the device registers.
    const lower: apollo3.RealTimeClock.CtrLowReg = .{
        .ctr_100 = .{
            .h = @intCast(rtc_time.hundredth / 10),
            .l = @intCast(rtc_time.hundredth % 10),
        },
        .ctr_sec = .{
            .h = @intCast(rtc_time.second / 10),
            .l = @intCast(rtc_time.second % 10),
        },
        .ctr_min = .{
            .h = @intCast(rtc_time.minute / 10),
            .l = @intCast(rtc_time.minute % 10),
        },
        .ctr_hr = .{
            .h = @intCast(rtc_time.hour / 10),
            .l = @intCast(rtc_time.hour % 10),
        },
    };

    const upper: apollo3.RealTimeClock.CtrUpReg = .{
        .ctr_wkdy = @intCast(rtc_time.weekday),
        .ctr_yr = .{
            .h = @intCast((rtc_time.year - century_base) / 10), // device only holds 2
digits of year
            .l = @intCast((rtc_time.year - century_base) % 10),
        },
        .ctr_mo = .{
            .h = @intCast(rtc_time.month / 10),
        }
    };
}
```

```

        .l = @intCast(rtc_time.month % 10),
    },
    .ctr_date = .{
        .h = @intCast(rtc_time.date / 10),
        .l = @intCast(rtc_time.date % 10),
    },
};

const rtdev = self.device;
rtdev.rtc_ctl.updateField(.wrtc, .en); ①
rtdev.ctr_low.write(lower);
rtdev.ctr_up.write(upper);
rtdev.rtc_ctl.updateField(.wrtc, .dis);

return .success;
}

```

- ① *N.B.* it is important to ensure that the RTC is not stopped when attempting to write to the counter registers (*i.e.* the RST0P bit of RTCCTL must be 0). As determined empirically, if the RTC is stopped, you can't write to the counter registers regardless of the value of the WRTC bit. Seem that the RST0P is intended for "stop watch" style operations. There is an brief mention of this in the data sheet. The WRTC field only controls the ability to write to CTRLLOW and CTRUP. Other RTC registers don't seem to be affected by the WRTC setting. The data sheet is vague in this area.

RTC alarm value

The RTC has compare registers that can generate an interrupt when they match the counter values of the RTC. Only a single set of alarm registers is provided by the RTC and so Only a single alarm setting at a time is supported.

The alarm generates repeating alarms. It is either disabled, or generates a periodic interrupt. The repeat function controls how many alarm registers are considered in determining if the current time matches the alarm.

```

<<rtc services>>=
pub const AlarmRepeat = apollo3.RealTimeClock.AlarmRepeat;

```

The broken out structure for an alarm is similar to that of the RTC counters. Note there is no year member. The alarm operates by matching all the counter values that are more frequent than the selected alarm repeat.

```

<<rtc services>>=
pub const RtcAlarm = struct {
    repeat: AlarmRepeat,
    month: u8,
    date: u8,
    hour: u8,
    minute: u8,
    second: u8,
    hundredth: u8,
    weekday: u8,
};

```

The alarm comparison match registers are also held in BCD encoding.

Updating the Alarm time

The following function updates the alarm time in the RTC peripheral. Time in this case is considered to be in device units and is inserted directly into the RTC peripheral registers. Like the RTC time fields, the alarm values are inserted into bit fields contained in two peripheral registers.

```
<<device control methods>>=
fn updateRtcAlarm(
    self: *const DevControl,
    alarm_time: *const RtcAlarm,
) SvcResult {
    // Convert to BCD in preparation for writing the device registers.
    const lower: apollo3.RealTimeClock.AlmLowReg = .{
        .alm_100 = .{
            .h = @intCast(alarm_time.hundredth / 10),
            .l = @intCast(alarm_time.hundredth % 10),
        },
        .alm_sec = .{
            .h = @intCast(alarm_time.second / 10),
            .l = @intCast(alarm_time.second % 10),
        },
        .alm_min = .{
            .h = @intCast(alarm_time.minute / 10),
            .l = @intCast(alarm_time.minute % 10),
        },
        .alm_hr = .{
            .h = @intCast(alarm_time.hour / 10),
            .l = @intCast(alarm_time.hour % 10),
        },
    };
    const upper: apollo3.RealTimeClock.AlmUpReg = .{
        .alm_wkdy = @intCast(alarm_time.weekday),
        .alm_mo = .{
            .h = @intCast(alarm_time.month / 10),
            .l = @intCast(alarm_time.month % 10),
        },
        .alm_date = .{
            .h = @intCast(alarm_time.date / 10),
            .l = @intCast(alarm_time.date % 10),
        },
    };
    const rtdev = self.device;
    rtdev.alm_low.write(lower);
    rtdev.alm_up.write(upper);
    rtdev.rtc_ctl.updateField(.rpt, alarm_time.repeat);
    rtdev.rtc_ctl.updateField(.rstop, .run);

    return .success;
}
```

}

RTC Requests

In this section, device interface and control code for the RTC device is shown. The following diagram shows the peripherals involved in the RTC device code.

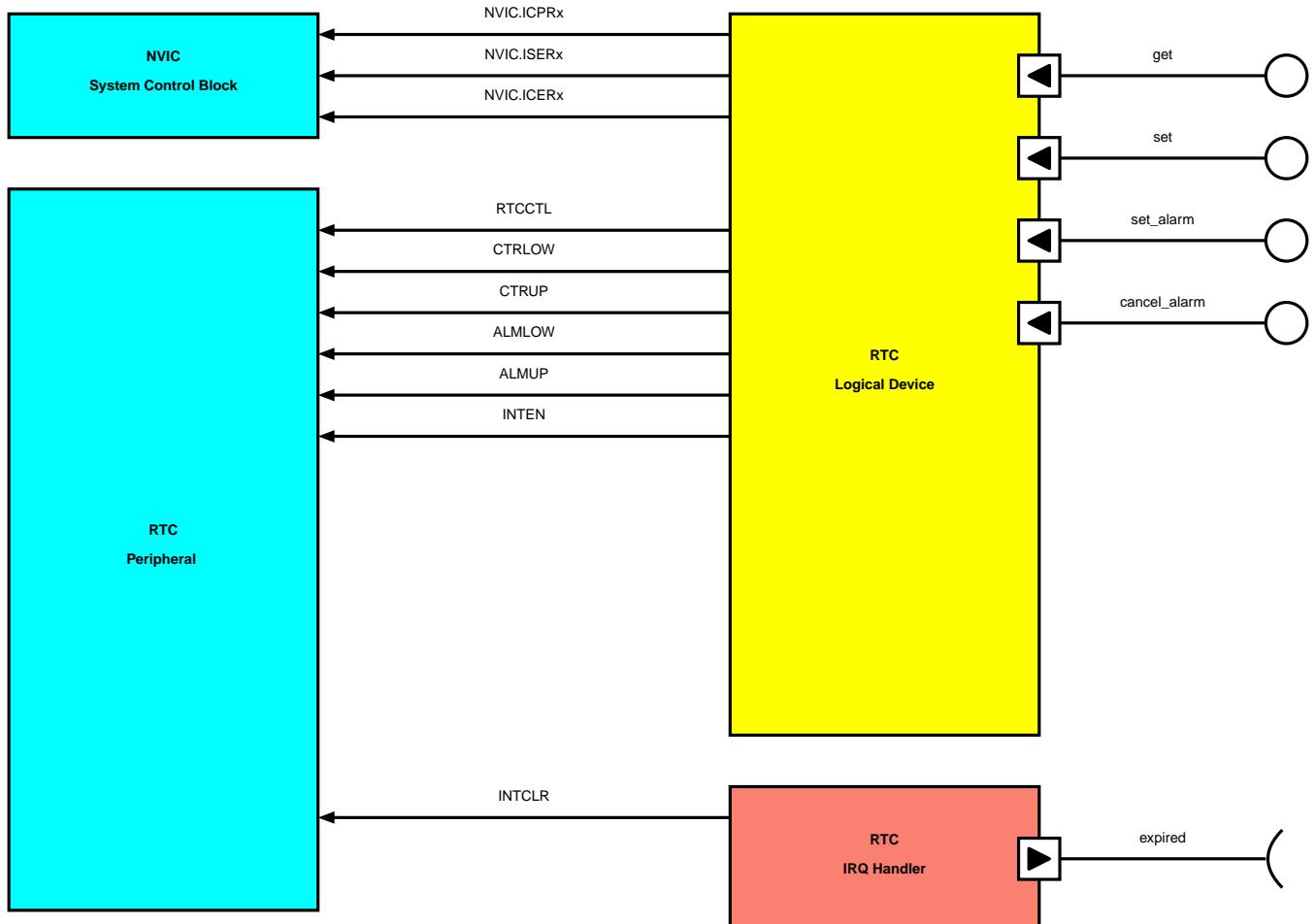


Figure 20. RTC Interface Components

RTC request encoding

An encoding for the operations supported by the RTC device is given by the following enumeration.

```
<<system_services: device service definitions>>=
/// Real Time Clock services
pub const rtc = struct {
    <<rtc services>>
};
```

```
<<rtc services>>=
pub const Operations = DefineDeviceOperations(
    .rtc,
    &.{"set", "get", "set_alarm", "cancel_alarm"},
```

```
);
```

Background Notification

Since the alarming capability of the RTC is interrupt driven, the structure of the background notification which is sent when the RTC signals the alarm must be defined. The background notification carries a status value to indicate what happened.

```
<<rtc services>>=
pub const NotifyReason = enum {
    expired,
};
```

The expired status indicates that the alarm values match the current time as governed by the alarm repeat control. The discarded status indicates that a previous alarm request was discarded because the time value in the peripheral was changed by a `rtc_set` operation or that a new alarm was set by the `rtc_set_alarm` operation. Changing the time in the RTC causes any pending alarm request to be canceled automatically. Setting a new alarm when there is one already pending cancels the previous alarm. The discarded status indicates to background processing that the pending alarm will never be received.

```
<<dev notification specs>>=
.{ "rtc", struct {
    reason: rtc.NotifyReason,
    time: rtc.RtcTime,
}
},
```

Note that in addition to the alarm notification reason, the current RTC time when the alarm expired is also included in the background notification as a convenience.

The background notification proxy takes as an argument the device specific form of the notification.

The code to send the RTC background notification is factored into a single function.

```
<<control block methods>>=
fn notifyToBackground(
    rcb: *ControlBlock,
) SvcResult {
    var status: SvcResult = .success;

    if (rcb.notification) |note| {
        status = rcb.dev_control.readRtcTime(&note.params.rtc.time);
        if (status != .success) return status;

        //BEGIN PRIORITY SECTION
        const section =
            v7m.basepri.PrioritySection.enter(system_config.irq_encoded_priority);
        defer section.leave();
        status = dev_notify_queue.sendNotification(note);
        //END PRIORITY SECTION
    }
}
```

```

} else {
    status = .operation_failed;
}

return status;
}

```

There is only one RTC peripheral in the Apollo 3 SOC.

```

<<rtc services>>=
pub const instances: DevSvcInstance = 1;

```

Even though there is only one RTC peripheral, it is convenient to associate its IRQ number to the peripheral registers address block.

```

<<rtc proxies>>=
const DevControl = struct {
    /// The interrupt number associated with the RTC compare register.
    irq_num: v7m.IrqNumber,
    /// A pointer to the memory mapped I/O for the RTC registers.
    device: *volatile apollo3.RealTimeClock,

    fn disableAlarmIrq(
        self: *const DevControl,
    ) void {
        self.device.rtc_ctl.updateField(.rpt, .dis);
        self.device.int_en.clearBitField(.alm);
        self.clearAlarmIrq();
        v7m.nvic.disableIrq(self.irq_num);
    }

    fn enableAlarmIrq(
        self: *const DevControl,
    ) void {
        self.clearAlarmIrq();
        self.device.int_en.setBitField(.alm);
        v7m.nvic.enableIrq(self.irq_num);
    }

    fn clearAlarmIrq(
        self: *const DevControl,
    ) void {
        self.device.int_clr.writeBitField(.alm);
    }

    fn setControls(
        self: *const DevControl,
    ) void {
        self.device.rtc_ctl.updateField(.hr_12_24, .@"24hr");
        self.device.rtc_ctl.updateField(.rpt, .dis);
    }
}

```

```

<<device control methods>>
};

```

Following the established pattern, a data structure is defined to hold all the required control information for the RTC.

```

<<rtc proxies>>=
const ControlBlock = struct {
    notification: ?*svc_services.DevNotification,
    /// The parameters needed to control the system timer hardware.
    dev_control: DevControl,

    <<control block methods>>
};


```

The control block is initialized at compile time and is ready for use when the system starts.

```

<<rtc proxies>>=
var rtc_devices = [instances]ControlBlock{
    .{ // 0
        .notification = null,
        .dev_control = .{
            .irq_num = apollo3.RealTimeClock.irq_number,
            .device = apollo3.rtc,
        },
    },
};

var rtc_notifications: [instances]DevNotification = undefined;

```

Setting the time of day

In this section and the following, the usual pattern of functions for device control are given for the RTC case. First, the interface to background processing and then the foreground proxy function which runs with privilege and access the peripheral device.

```

<<rtc services>>=
/// The `set` function sets the Real Time Clock (RTC) instance given by, `rtc`, to
/// the time value pointed to by, `time`. It is an error to attempt to set the
/// RTC time when there is a pending alarm.
pub fn set(
    /// The instance number of the RTC peripheral which is to be set.
    inst: DevSvcInstance,
    /// A pointer to a broken out time to which the RTC is to be set.
    rtc_time: *const RtcTime,
) SvcError!void {
    const input_params = SetInputParam.marshal(.{
        .time = rtc_time.*,
    });
}

```

```

const request = Operations.makeRequest(.set, inst);
return devRealmSvcCall(request, @ptrCast(&input_params), null, null);
}

```

Following the usual pattern, an input structure is defined to pass the time argument to the foreground proxy.

```

<<rtc services>>=
pub const SetInputParam = DefineSvcRequestParam(struct {
    time: RtcTime,
});

```

```

<<device proxies>>=

/// Real Time Clock service proxies
const rtc = struct {
    const NotifyReason = svc_services.rtc.NotifyReason;
    const RtcTime = svc_services.rtc.RtcTime;
    const RtcAlarm = svc_services.rtc.RtcAlarm;
    const instances = svc_services.rtc.instances;

    <<rtc proxies>>
};


```

```

<<rtc proxies>>=
fn set(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const instance = req.instance;
    if (instance >= instances) return .unknown_instance;

    const SetInputParam = svc_services.rtc.SetInputParam;
    const input_params, const status = SetInputParam.receive(input.?);
    if (status != .success) return status;

    const rcb = &rtc_devices[instance];
    if (rcb.notification != null) return .operation_failed; ❶

    rcb.dev_control.disableAlarmIrq();
    rcb.dev_control.setControls(); ❷
    return rcb.dev_control.updateRtcTime(&input_params.time);
}

```

- ❶ A non-null notification indicates that an alarm has been posted. Fail the operation in that case since setting the time most probably will cause that alarm periodicity to be disturbed.
- ❷ Since the RTC is run in a particular configuration, *i.e.* 24 hour mode, 20th century, etc. the configuration is set each time the RTC time is set. This ensures the peripheral is operated as this

code is expecting. Again, note it is *not* the purpose to try to build a completely general interface which supports every possible configuration of the peripheral device.

Getting the time of day

```
<<rtc services>>=
/// The `get` function obtains the time of day from the Real Time Clock (RTC)
/// instance given by, `rtc`, and returns the time into the memory object pointed
/// to by, `time`.
pub fn get(
    /// The instance number of the RTC peripheral which is to be accessed.
    inst: DevSvcInstance,
    /// A pointer to a broken out time which is used to return the current time of
    /// day.
    rtc_time: *RtcTime,
) SvcError!void {
    const input_params = GetInputParam.marshal(.{
        .time = rtc_time,
    });
    const request = Operations.makeRequest(.get, inst);
    return devRealmSvcCall(request, @ptrCast(&input_params), null, null);
}
```

```
<<rtc services>>=
pub const GetInputParam = DefineSvcRequestParam(struct {
    time: *RtcTime,
});
```

```
<<rtc proxies>>=
fn get(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const instance = req.instance;
    if (instance >= instances) return .unknown_instance;

    const GetInputParam = svc_services.rtc.GetInputParam;
    const input_params, var status = GetInputParam.receive(input.?);
    if (status != .success) return status;

    const rcb = &rtc_devices[req.instance];

    var rtc_time: RtcTime = undefined;
    status = rcb.dev_control.readRtcTime(&rtc_time);
    if (status != .success) return status;

    memcpyToUnpriv(mem.asBytes(input_params.time), mem.asBytes(&rtc_time));
    return status;
```

```
}
```

Setting an alarm

```
<<rtc services>>=
/// The `set_alarm` function sets an alarm specified by the object pointed to by,
/// `alarm`, on the RTC given by, `rtc`. When the alarm expires, a notification
/// with a reason of `expired` is placed in the background notification queue.
/// It is an error to attempt to set an RTC alarm if one is already pending.
pub fn setAlarm(
    /// The instance number of the RTC peripheral on which the alarm is to be set.
    inst: DevSvcInstance,
    /// A pointer to a broken out alarm time which is to be set.
    rtc_alarm: *const RtcAlarm,
    /// A pointer to a background notification function which is passed through in
    /// the notification.
    notify_proxy: DevSvcNotifyProxy,
    /// User specified data value returned in the alarm notification.
    notify_closure: usize,
) SvcError!void {
    const input_params = SetAlarmInputParam.marshal(.{
        .alarm = rtc_alarm.*,
        .notify_proxy = notify_proxy,
        .notify_closure = notify_closure,
    });
    const request = Operations.makeRequest(.set_alarm, inst);
    return devRealmSvcCall(request, @ptrCast(&input_params), null, null);
}
```

```
<<rtc services>>=
pub const SetAlarmInputParam = DefineSvcRequestParam(struct {
    alarm: RtcAlarm,
    notify_proxy: DevSvcNotifyProxy,
    notify_closure: usize,
});
```

```
<<rtc proxies>>=
fn setAlarm(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const instance = req.instance;
    if (instance >= instances) return .unknown_instance;

    const SetAlarmInputParam = svc_services.rtc.SetAlarmInputParam;
    const input_params, var status = SetAlarmInputParam.receive(input.?);
    if (status != .success) return status;
```

```

const rcb = &rtc_devices[req.instance];

if (rcb.notification != null) return .operation_failed;

status = rcb.dev_control.updateRtcAlarm(&input_params.alarm);
if (status != .success) return status;

rcb.notification = &rtc_notifications[instance];
rcb.notification.?.* = DevNotification.init(
    req.class,
    instance,
    input_params.notify_proxy,
    input_params.notify_closure,
    .{ .rtc =
        .{
            .reason = .expired,
            .time = undefined,
        },
    },
);

rcb.dev_control.enableAlarmIrq();
return .success;
}

```

Cancelling an alarm

```

<<rtc services>>=
pub fn cancelAlarm(
    /// The instance number of the RTC peripheral for which the alarm is to be
    /// cancelled.
    inst: DevSvcInstance,
) SvcError!void {
    const request = Operations.makeRequest(.cancel_alarm, inst);
    return devRealmSvcCall(request, null, null, null);
}

```

```

<<rtc proxies>>=
fn cancelAlarm(
    req: DevSvcRequest,
    _: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const instance = req.instance;
    if (instance >= instances) return .unknown_instance;

    const rcb = &rtc_devices[req.instance];
    rcb.dev_control.disableAlarmIrq();

    rcb.notification = null;
}

```

```
    return .success;
}
```

Dispatching RTC Requests

In keeping with the established pattern for device operations, RTC requests have an additional level of dispatch and a function is required to find the foreground proxy function associated with the device operation.

```
<<svc device classes>>=
rtc,
```

```
<<dev svc service request prongs>>=
 rtc => &rtc.devClassHandler,
```

Finally, the UART class handler performs the dispatch to the operation functions of requests.

```
<<rtc proxies>>=
/// The `devClassHandler` function dispatches RTC operations based
/// on the value of the `req` argument. This function is invoked as a
/// class level dispatcher for the timer queue device.
fn devClassHandler(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    output: ?*anyopaque,
    err: ?*anyopaque,
) SvcResult {
    const Operation = svc_services.rtc.Operations.Operation;
    const rtc_operation: Operation = @enumFromInt(req.operation);
    const proxy = switch (rtc_operation) {
        .set => &rtc.set,
        .get => &rtc.get,
        .set_alarm => &rtc.setAlarm,
        .cancel_alarm => &rtc.cancelAlarm,
    };
    return proxy(req, input, output, err);
}
```

RTC IRQ Handling

IRQ handling for the RTC only involves clearing the interrupt and sending a background notification.

```
<<rtc proxies>>=
export fn rtcIrqHandler() void {
    const rcb = &rtc_devices[0]; ①
    rcb.dev_control.clearAlarmIrq();
    const status = rcb.notifyToBackground();
    if (status != .success) {
        rcb.dev_control.disableAlarmIrq();
```

```
    }  
}
```

- ❶ There is, after all, only one RTC peripheral in the Apollo 3 SOC. If there were more, then we would have determine which one is interrupting either by it having a separate IRQ number or reading some register to know.

Testing

```
<<rtc_unit_tests.zig>>=  
<<edit warning>>  
<<copyright info>>  
  
const std = @import("std");  
const log = std.log;  
const testing = @import("resee_testing");  
  
const svc_services = @import("svc_services");  
const rtc = svc_services.rtc;  
const DevNotification = svc_services.DevNotification;  
const exec_control = svc_services.exec_control;  
  
const current: rtc.RtcTime = .{  
    .year = 2021,  
    .month = 12,  
    .date = 31,  
    .hour = 23,  
    .minute = 59,  
    .second = 59,  
    .hundredth = 0,  
    .weekday = 2,  
};  
  
test "set rtc time" {  
    testing.log_level = .info;  
  
    log.info("current = {f}", .{current});  
    const status = rtc.set(0, &current);  
    try testing.expectEqual({}, status);  
}  
  
test "get rtc time" {  
    var now: rtc.RtcTime = undefined;  
  
    const status = rtc.get(0, &now);  
    try testing.expectEqual({}, status);  
    log.info("now = {f}", .{now});  
    try testing.expectEqual(current, now);  
}  
  
fn alarmExpired(  
    notify: *const anyopaque,
```

```

) void {
    const notification: *const DevNotification = @ptrCast(@alignCast(notify));
    const done: *volatile bool = @ptrFromInt(notification.closure);
    done.* = true;

    log.info("alarm expired: {f}", .{notification.params.rtc.time});
}

test "one second alarms" {
    const alarmAt: rtc.RtcAlarm = .{
        .repeat = .sec,
        .month = 0,
        .date = 0,
        .hour = 0,
        .minute = 0,
        .second = 0,
        .hundredth = 99,
        .weekday = 0,
    };

    var alarm_expired: bool = false;

    var status = rtc.setAlarm(0, &alarmAt, &alarm_expired, @intFromPtr(&alarm_expired));
    try testing.expectEqual({}, status);

    try exec_control.runWait(&alarm_expired); // discard first alarm to get synced up

    var now: rtc.RtcTime = undefined;
    var alarm_count: usize = 0;
    while (alarm_count <= 10) : (alarm_count += 1) {
        status = exec_control.runWait(&alarm_expired);
        try testing.expectEqual({}, status);
        try testing.expectEqual(true, alarm_expired);
        try rtc.get(0, &now);
        try testing.expectEqual(alarm_count, now.second);
    }

    status = rtc.cancelAlarm(0);
    try testing.expectEqual({}, status);
}

test "one minute and 10 second alarms" {
    const alarmAt: rtc.RtcAlarm = .{
        .repeat = .min,
        .month = 0,
        .date = 0,
        .hour = 0,
        .minute = 0,
        .second = 9,
        .hundredth = 99,
        .weekday = 0,
    };
}

```

```

var alarm_expired: bool = false;

try rtc.set(0, &current);
var status = rtc.setAlarm(0, &alarmAt, &alarmExpired, @intFromPtr(&alarm_expired));
try testing.expectEqual({}, status);

var now: rtc.RtcTime = undefined;
var alarm_count: usize = 0;
while (alarm_count <= 3) : (alarm_count += 1) {
    alarm_expired = false;
    status = exec_control.runWait(&alarm_expired);
    try testing.expectEqual({}, status);
    try testing.expectEqual(true, alarm_expired);
    try rtc.get(0, &now);
    try testing.expectEqual(alarm_count, now.minute);
    try testing.expectEqual(@as(u8, 10), now.second);
}
try rtc.cancelAlarm(0);
}

```

Computer Time

The first microprocessors were simple enough devices that instruction execution timing was determined solely by the frequency of the system clock. With careful, albeit tedious, coding, instruction execution could be used as a means to measure time. Modern devices, with caches, wait states, and performance enhancing features, no longer have a direct relationship between instruction execution and system clock cycles. Counting instructions makes for a poor approximation to time. In addition to being wasteful of power, timer peripherals just do a better job.

However, the performance of software still has a direct relationship to the number of system clock cycles required to execute a given piece of code. Knowledge of that relationship is required to make informed engineering trade-offs between speed and memory. For the ARM v7-M architecture, there is a cycle counter which counts the number of system clock ticks as a program executes. With knowledge of the system clock frequency, the cycle counter is an accurate representation of execution time.

Our primary tool for measuring execution is a built-in cycle counter. In this chapter, a set of system realm request function to access the cycle counter which resides in the Data Watchpoint and Trace (DWT) unit of the Cortex-M4 processor are provided. There is one difficulty that you notice immediately. Access to the cycle counter is a privileged operation. This means foreground requests via the SVC exception must be made to operate on the cycle counter. This implies that the number of cycles measured is also going to contain cycles associated with leaving privileged context after the foreground request has manipulated the counter. This implies that only relative comparisons ,rather than the absolute numbers, of cycle counts are meaningful since the cycle counts includes the overhead of the foreground request mechanisms.

Cycle Counter Services

```

<<system_services: system service definitions>>=
/// Cycle counter services
pub const cycle_counter = struct {
    <<cycle counter services>>

```

```
};
```

```
<<sys svc service requests>>=
start_cycle_counter,
stop_cycle_counter,
read_cycle_counter,
```

```
<<sys svc service request prongs>>=
.start_cycle_counter => &cycle_counter.start,
.stop_cycle_counter => &cycle_counter.stop,
.read_cycle_counter => &cycle_counter.read,
```

```
<<system proxies>>=
/// Cycle counter service proxies
const cycle_counter = struct {
    <<cycle counter proxies>>
};
```

Starting the cycle counter

```
<<cycle counter services>>=
pub fn start() SvcError!void {
    return sysRealmSvcCall(.start_cycle_counter, null, null, null);
}
```

```
<<cycle counter proxies>>=
fn start(
    _: SysSvcRequest,
    _: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    if (v7m.dwt.dwt_ctrl.testBitField(.no_cyc_cnt)) return .operation_failed; ①
    v7m.dcb.demcr.setBitField(.trc_ena);
    v7m.dwt.dwt_ctrl.setBitField(.cyc_cnt_ena);
    v7m.dwt.dwt_cyc_cnt = 0;
    return .success;
}
```

① Strictly speaking the DWT block is optional, although I haven't seen a chip in the wild that does not have it.

Stopping the cycle counter

```
<<cycle counter services>>=
pub fn stop() SvcError!void {
    return sysRealmSvcCall(.stop_cycle_counter, null, null, null);
```

```
}
```

```
<<cycle counter proxies>>=
fn stop(
    _: SysSvcRequest,
    _: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    if (v7m.dwt.dwt_ctrl.testBitField(.no_cyc_cnt)) return .operation_failed;
    v7m.dcb.demcr.clearBitField(.trc_ena);
    v7m.dwt.dwt_ctrl.clearBitField(.cyc_cnt_ena);
    return .success;
}
```

Reading the cycle counter

```
<<cycle counter services>>=
pub fn read() SvcError!u32 {
    var output_params: ReadOutputParam = .initial;
    try sysRealmSvcCall(.read_cycle_counter, null, @ptrCast(&output_params), null);
    return output_params.params.cycle_count;
}
```

```
<<cycle counter services>>=
pub const ReadOutputParam = DefineSvcRequestParam(struct {
    cycle_count: u32,
});
```

```
<<cycle counter proxies>>=
fn read(
    _: SysSvcRequest,
    _: ?*const anyopaque,
    output: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    if (v7m.dwt.dwt_ctrl.testBitField(.no_cyc_cnt)) return .operation_failed;
    const ReadOutputParam = svc_services.cycle_counter.ReadOutputParam;
    return ReadOutputParam.send(.{
        .cycle_count = v7m.dwt.dwt_cyc_cnt,
    }, output.?);
}
```

```
<<cycle_counter_unit_tests.zig>>=
<<edit warning>>
<<copyright info>>

const std = @import("std");
```

```

const log = std.log;
const testing = @import("resee_testing");

const svc_services = @import("svc_services");
const cycle_counter = svc_services.cycle_counter;

test "start cycle counter" {
    testing.log_level = .info;

    try cycle_counter.start();
    const counter = try cycle_counter.read();
    log.info("counter = {d}", .{counter});
    try testing.expect(counter < 1_000);

    var prev = try cycle_counter.read();
    for (0..10) |_| {
        const current = try cycle_counter.read();
        log.info("counter = {d}, diff = {d}", .{current, current - prev});
        prev = current;
    }
}

test "stop cycle counter" {
    try cycle_counter.stop();
    const counter = try cycle_counter.read();
    try testing.expect(counter != 0);
}

```

[1] There are also capture registers in the system timer block, but they are not used here

[2] Apollo3 Blue MCU Errata List, Doc.ID:SE-A3_2p09, Revision 2.0, July 2019

[3] Undoubtedly for some backwards compatible reason.

Shoring Up the Foundation

To recap, the previous chapters have accomplished:

- a. Start up code that performs the necessary initializations to reach the `main()` function.
- b. Code execution has been separated into privileged and unprivileged modes.
- c. A mechanism based upon the SVC exception was built to allow unprivileged and privileged code to interact.
- d. System and logging consoles have been created using the UARTs on the Apollo 3.
- e. Several time keeping services based on the system timer have been realized.

However, this foundation is not strong enough to support robust deployment. The handling of system faults and missing IRQ handlers does not support saving any information which might help in diagnosing the fault. This chapter shores up the exception and error handling for a wider set of conditions.

Software detected faults

The processor core generates system faults when it detects that the external environment is not responding properly or when the executing program does not operate the processor properly. For example, misbehaving memory can result in a Bus Fault and software execution of a divide operation with a zero value for the divisor can result in a Usage Fault.

Similar circumstances can arise when operating peripheral devices or manipulating system software structures. For example, many UART devices offer status and an interrupt, to indicate conditions such as transmitter overflow or receiver underflow. Transmitter overflow results from attempting to place data into the transmitter when it is already full. Similarly, receiver underflow occurs when software attempts to read the receiver when it is empty. Both of these conditions indicate that the UART is being operated improperly since there is always sufficient status information available from the UART registers to determine if the write or read operation is allowed. There are many circumstances where hardware peripherals can detect improper operation by software. These situations often have interrupts associated with them that can be used to detect run time operational problems.

A **panic condition** is defined by analogy to a system fault. The panic condition is not intended to be recoverable. It signifies a situation where software has detected a condition and does not have any way to proceed without the possibility of uncontrolled execution.

The Zig language also has a concept of panic as exemplified by the `@panic` builtin function. Zig allows panic handling to be supplied by an application. There is a default which works well in an OS environment, but for a microcontroller there are other considerations. In this chapter, we show how to override the default `@panic` behavior so as to gracefully bring the system down and to record reasons as to why it went down.

An easy solution to a panic would be to invoke `systemAbEnd()`. Recall that invoking `systemAbEnd` was defined as “[hitting rock bottom](#)”. But, just invoking `systemAbEnd` does not give any information about the circumstances of the detected problem. Some clue of the exceptional condition which caused the panic condition needs to be saved.

You must also consider execution privilege when determining a design for panic handling. Only privileged code can invoke `systemAbEnd` directly. Storing away the circumstances of the panic in memory that survives reset is also a privileged operation. There can also be panic conditions

detected during privileged execution.

In this section, we show the design of the panic condition handling.

1. The Zig compiler @panic compiler builtin function is overridden to supply a function that handles panic conditions in a manner compatible with the other components of the system.
2. A new system service call, sysPanic, is supplied as the fundamental building block for the panic function that replaces the Zig compiler's default action.
3. The system foreground proxy supplies a function that will be used both to complete the sysPanic service and that can be directly invoked by privileged code. This function is called, privPanic and will accept arguments that contain information about the panic condition.
4. The privPanic function stores away the panic condition information, writes a message to the privileged console, and ultimately calls systemAbEnd.

Zig Panic

The Zig compiler resolves the @panic builtin to be handled by the Panic namespace. Code in `builtin.zig` determines which Panic namespace is used. For us to override the compiler default, it is necessary to have a Panic namespace declaration in our root file. The starting point for overriding the compiler's panic is the `FormattedPanic.zig` file from the standard library debug directory. That file will then be modified to adapt it to our needs.

```
<<panic service>>=
// This code contains a modified version of the default Panic namespace
// from the Zig standard library for version 0.14.0-dev.3026+c225b780e
// of the Zig compiler. It has been modified to adapt it to the ReSEE
// environment.
pub fn call(msg: []const u8, ra: ?usize,) noreturn {
    @branchHint(.cold);
    sysPanic(ra, msg) catch unreachable; // sysPanic is the ReSEE specific panic
    unreachable;
}
pub fn sentinelMismatch(expected: anytype, found: @TypeOf(expected)) noreturn {
    @branchHint(.cold);
    // formattedPanic accomplishes the same as std.debug.panicExtra does for ReSEE
    formattedPanic(
        @returnAddress(),
        "sentinel mismatch: expected {any}, found {any}",
        .{ expected, found },
    );
    unreachable;
}
pub fn unwrapError(err: anyerror) noreturn {
    @branchHint(.cold);
    formattedPanic(
        @returnAddress(),
        "attempt to unwrap error: {s}",
        .{@errorName(err)},
    );
    unreachable;
}
```

```

pub fn outOfBounds(index: usize, len: usize) noreturn {
    @branchHint(.cold);
    formattedPanic(
        @returnAddress(),
        "index out of bounds: index {d}, len {d}",
        .{ index, len },
    );
    unreachable;
}

pub fn startGreater ThanEnd(start: usize, end: usize) noreturn {
    @branchHint(.cold);
    formattedPanic(
        @returnAddress(),
        "start index {d} is larger than end index {d}",
        .{ start, end },
    );
    unreachable;
}

pub fn inactiveUnionField(active: anytype, accessed: @TypeOf(active)) noreturn {
    @branchHint(.cold);
    formattedPanic(
        @returnAddress(),
        "access of union field '{any}' while field '{any}' is active",
        .{ active, accessed },
    );
    unreachable;
}

pub fn sliceCastLenRemainder(src_len: usize) noreturn {
    @branchHint(.cold);
    formattedPanic(
        @returnAddress(),
        "slice length '{d}' does not divide exactly into destination elements",
        .{src_len},
    );
}

pub fn reachedUnreachable() noreturn {
    @branchHint(.cold);
    call("reached unreachable code", @returnAddress());
}

pub fn unwrapNull() noreturn {
    @branchHint(.cold);
    call("attempt to use null value", @returnAddress());
}

pub fn castToNull() noreturn {
    @branchHint(.cold);
    call("cast causes pointer to be null", @returnAddress());
}

pub fn incorrectAlignment() noreturn {
    @branchHint(.cold);
    call("incorrect alignment", @returnAddress());
}

pub fn invalidErrorCode() noreturn {

```

```

@branchHint(.cold);
call("invalid error code", @returnAddress());
}

pub fn integerOutOfBounds() noreturn {
@branchHint(.cold);
call("integer does not fit in destination type", @returnAddress());
}

pub fn integerOverflow() noreturn {
@branchHint(.cold);
call("integer overflow", @returnAddress());
}

pub fn shlOverflow() noreturn {
@branchHint(.cold);
call("left shift overflowed bits", @returnAddress());
}

pub fn shrOverflow() noreturn {
@branchHint(.cold);
call("right shift overflowed bits", @returnAddress());
}

pub fn divideByZero() noreturn {
@branchHint(.cold);
call("division by zero", @returnAddress());
}

pub fn exactDivisionRemainder() noreturn {
@branchHint(.cold);
call("exact division produced remainder", @returnAddress());
}

pub fn integerPartOutOfBounds() noreturn {
@branchHint(.cold);
call("integer part of floating point value out of bounds", @returnAddress());
}

pub fn corruptSwitch() noreturn {
@branchHint(.cold);
call("switch on corrupt value", @returnAddress());
}

pub fn shiftRhsTooBig() noreturn {
@branchHint(.cold);
call("shift amount is greater than the type size", @returnAddress());
}

pub fn invalidEnumValue() noreturn {
@branchHint(.cold);
call("invalid enum value", @returnAddress());
}

pub fn forLenMismatch() noreturn {
@branchHint(.cold);
call("for loop over objects with non-equal lengths", @returnAddress());
}

pub fn copyLenMismatch() noreturn {
@branchHint(.cold);
call("@memcpy arguments have non-equal lengths", @returnAddress());
}

pub fn memcpyAlias() noreturn {

```

```

@branchHint(.cold);
call("@memcpy arguments alias", @returnAddress());
}

pub fn noreturnReturned() noreturn {
    @branchHint(.cold);
    call("'noreturn' function returned", @returnAddress());
}

```

sysPanic

The `sysPanic` function makes a background service request to handle a panic condition. Ultimately, there is no return from `sysPanic` and the default behavior is to reset the microcontroller. As expected, the function formulates a request to its corresponding foreground proxy.

```

<<system_services: system service definitions>>=
/// The panic service is used to stop system execution and either
/// return to an attached debugger or reset the processor core.
pub const panic = struct {
    <<panic service>>
};

```

Each system SVC service must add a name to the request enumeration.

```

<<sys svc service requests>>=
panic,

```

```

<<sys svc service request prongs>>=
.panic => &panic.sysPanic,

```

A function interface for the background request must be defined. The `sysPanic` function requires an input argument to carry the panic message and the return address data.

```

<<panic service>>=
/// The `sysPanic` function generates a foreground system request to store
/// the `msg` as a panic message along with an optional return `address`.
/// This function can only be invoked from unprivileged execution since it requires
/// an SVC operation.
pub fn sysPanic(
    address: ?usize,
    msg: []const u8,
) SvcError!void {
    const input_params = PanicInputParam.marshal(.{
        .address = address orelse @returnAddress(),
        .msg = msg,
    });
    return sysRealmSvcCall(.panic, @ptrCast(&input_params), null, null);
}

```

```

<<panic service>>=

```

```

pub const PanicInputParam = DefineSvcRequestParam(struct {
    address: usize,
    msg: []const u8,
});
```

```

<<panic service>>=
/// The `formattedPanic` function formats the `args` argument according to the given
/// `format` and passes the formatted output along with the `address` operation to
/// the `sysPanic` function.
pub fn formattedPanic(
    address: ?usize,
    comptime format: []const u8,
    args: anytype,
) void {
    var buf: [128]u8 = undefined;
    const msg = std.fmt.bufPrint(&buf, format, args) catch blk: {
        const trunc = "...";
        @memcpy(buf[buf.len - trunc.len..], trunc);
        break :blk &buf;
    };
    sysPanic(address orelse @returnAddress(), msg) catch unreachable;
}
```

The implementation of `sysPanic` follows the pattern established in a [previous chapter](#) of the book. The `sysPanic` function makes a request and there is a foreground proxy which fulfills the request.

The other half of the system realm request is implemented by the foreground proxy function for `sysPanic`. Again, we use literate program chunks to accumulate the proxy functions.

```

<<system proxies>>=
/// The panic proxy implements the foreground portion of
/// the `sysPanic` service.
const panic = struct {
    <<panic proxies>>
};
```

```

<<panic proxies>>=
fn sysPanic(
    _: SysSvcRequest,
    input: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const InputParam = svc_services.panic.PanicInputParam;

    const input_param, const result = InputParam.receive(input.?);
    if (result != .success) {
        systemAbEnd();
    }
}
```

```

var msg: PanicMessage = undefined;
const msg_len = @min(msg.len, input_param.msg.len);
memcpyFromUnpriv(msg[0..msg_len], input_param.msg[0..msg_len]);

privPanic(input_param.address, msg[0..msg_len]);
unreachable;
}

```

```

<<panic proxies>>=
const PanicMessage = svc_services.panic.PanicMessage;

```

```

<<panic proxies>>=
fn privPanic(
    address: ?usize,
    msg: []const u8,
) noreturn {
    var panic_item = panic_storage.addOneBounded() catch blk: {
        _ = panic_storage.pop();
        break :blk panic_storage.addOneBounded() catch unreachable;
    };

    panic_item.timestamp = epoch_time.deviceEpochTime();
    panic_item.address = address orelse @returnAddress();
    const msg_len = @min(panic_item.msg.len, msg.len);
    @memcpy(panic_item.msg[0..msg_len], msg[0..msg_len]);
    panic_item.len = msg_len;

    var priv_console: PrivConsole = .init();
    var priv_writer = priv_console.writer(&.{});
    const privOut = priv_writer.interface();
    priv_console.config.setColor(privOut, .bright_blue) catch {};
    privOut.print("panic: {f}\n", .{panic_item}) catch {};
    priv_console.config.setColor(privOut, .reset) catch {};

    systemAbEnd();
}

```

Panic message storage

Storage of panic message is allocated in the `.noinit_data` section. First, sizing information must be specified. The values chosen here are reasonable for expected use.

```

<<panic service>>=
pub const panic_msg_size: usize = 128; // need config for this
pub const panic_msg_count: usize = 4; // need config for this
pub const PanicMessage = [panic_msg_size]u8;
pub const PanicRecord = struct {
    timestamp: i64,
    address: usize,
}

```

```

msg: PanicMessage,
len: usize,

pub fn format(
    value: PanicRecord,
    writer: *std.Io.Writer,
) std.Io.Writer.Error!void {
    // Timestamp is an i64 value in units of seconds
    try writer.print("{s}, timestamp = {d}.{d:0>}, address = {x:0>8}", .{
        value.msg[0..value.len],
        @divFloor(value.timestamp, time.us_per_s),
        @mod(value.timestamp, time.us_per_s),
        value.address,
    });
}
};


```

This design manages panic message buffers in the simplest fashion. The allocation structure is a bounded array. Should there be an overflow, the last element of the array is overwritten with new data. Thus, the first "N" and the last panic messages are preserved.

```

<<panic proxies>>=
var panic_buffer: [panic_msg_count]PanicRecord linksection(".noinit_bss") = undefined;
var panic_storage: std.ArrayList(PanicRecord) = .initBuffer(&panic_buffer);

```

Panic Record Requests

Simply storing away the panic information is not sufficient. We need a way to retrieve the information after coming out of reset.

```

<<sys svc service requests>>=
get_panic,

```

```

<<sys svc service request prongs>>=
.get_panic => &panic.getPanic,

```

```

<<panic service>>=
/// The `getPanic` function retrieves the chronologically youngest panic record
/// that was generated by invoking the `sysPanic` request. Once retrieved the
/// panic record is no longer available, i.e. its storage space is reclaimed.
/// If no panic records are available, the `operation_failed` error is returned.
pub fn getPanic(
    record: *PanicRecord,
) SvcError!void {
    const input_params = GetPanicInputParam.marshal(.{
        .panic_record = record,
    });
    return sysRealmSvcCall(.get_panic, @ptrCast(&input_params), null, null);
}

```

```
<<panic service>>=
pub const GetPanicInputParam = DefineSvcRequestParam(struct {
    panic_record: *PanicRecord,
});
```

```
<<panic proxies>>=
const panic_msg_size = svc_services.panic.panic_msg_size;
const panic_msg_count = svc_services.panic.panic_msg_count;
const PanicRecord = svc_services.panic.PanicRecord;
```

```
<<panic proxies>>=
fn getPanic(
    _: SysSvcRequest,
    input: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const GetPanicInputParam = svc_services.panic.GetPanicInputParam;
    const input_params, const result = GetPanicInputParam.receive(input.?.);
    if (result != .success) {
        return result;
    }

    const possible_panic_item = panic_storage.pop();
    if (possible_panic_item) |*panic_item| {
        memcpyToUnpriv(mem.asBytes(input_params.panic_record), mem.asBytes(panic_item));
    } else {
        return .operation_failed;
    }

    return .success;
}
```

sysHalt

Sometimes it is required to simply halt the system rather than panic and reboot. When running test programs, for example, normal behavior main behavior would be to return. This causes a panic condition, resets the system, restarts the test program, and the cycle is stuck in an infinite reset loop. For those occasions when we just want to stop and force an external reset to restart the system, we have the sysHalt service.

```
<<system_services: system service definitions>>=
/// The panic service is used to stop system execution and either
/// return to an attached debugger or reset the processor core.
pub const halt = struct {
    <<halt service>>
};
```

Each system SVC service must add a name to the request enumeration.

```
<<sys svc service requests>>=
halt,
```

```
<<sys svc service request prongs>>=
.halt => &halt.sysHalt,
```

```
<<halt service>>=
/// The `sysHalt` function generates a foreground system request to halt the system.
pub fn sysHalt() SvcError!void {
    return sysRealmSvcCall(.halt, null, null, null);
}
```

```
<<system proxies>>=
const halt = struct {
    <<halt proxies>>
};
```

```
<<halt proxies>>=
fn sysHalt(
    _: SysSvcRequest,
    _: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    var priv_console: PrivConsole = .init();
    var priv_writer = priv_console.writer(&.{});
    const privOut = priv_writer.interface();

    priv_console.config.setColor(privOut, .bright_blue) catch {};
    privOut.writeAll("\nhalt") catch {};
    priv_console.config.setColor(privOut, .reset) catch {};
    privOut.writeAll(": system halted\n") catch {};

    v7m.disableFaultIrq();
    v7m.wfe();
    unreachable;
}
```

Missing exception handler

The default exception handler defined previously provided a better way to handle exceptions than the default handlers usually provided in CMSIS start up files. That exception handler deferred the work of handling a missing exception to a function named, `missingHandler()`. The symbol for the function is defined as `weak` so it can be overridden. The default implementation of the function just invokes `systemAbEnd()`. Abnormal endings break into the debugger if it is attached and reset the system otherwise. This behavior does cover all the *necessary* conditions to prevent the processor core from running away and potentially locking up.

Neither of the actions of `systemAbEnd()` is *sufficient* for a robust system. If you happen to have an attached debugger, execution stops and your debugger may display some useful information, but it is still necessary to examine a number of core system registers to determine the exact cause of the exception. Code is needed to perform that examination and report the information in a convenient form. If there is no debugger attached, then the system is reset and the reason for the reset remains a mystery. Capturing the circumstances that have caused a reset is an essential component for a post-mortem examination.

In this section, a replacement for the default `missingHandler()` function is developed that solves the need for detailed information about system faults and missing IRQ handlers. To be clear, when an exception is triggered for which no specific handler has been put into vector table, then the following is required:

- Capture the essential information about the exception in a manner that supports post-mortem examination.
- Provide a means to access the exception fault records so they may be examined or printed.

Capturing the exception information breaks down into two parts:

1. Reading the system registers which contain fault information and
2. Storing the register values somewhere which is accessible after a reset.

The missing exception handler presented also handles the major system faults, *i.e.* HardFault, BusFault, MemManageFault, and UsageFault. These are important system faults as they indicate that the processor has detected misbehavior on the part of the program or the operating environment. These faults are considered unrecoverable, *i.e.* they must ultimately end in an abnormal termination. That approach makes the system faults good candidates to handle as a missing exception. Some systems may need to attempt some form of recovery. In those cases, the default fault handler for the fault can be overridden to supply any required recovery processing.

Faulting Information

The following data structure defines the information collected when a missing exception occurs. Note that it is not specific to the type of fault. Rather, all the information which might pertain to diagnosing the cause of the fault is gathered and post-mortem processing must determine what is applicable.

The total fault information is a combination of that provided by the core and some additional information provided by the Apollo 3 SOC.

```
<<fault service>>=
pub const FaultRecord = struct {
    timestamp: i64,
    cortex_status: CortexFaultRecord,
    apollo3_status: Apollo3FaultRecord,

    pub fn format(
        record: FaultRecord,
        writer: *std.Io.Writer,
    ) std.Io.Writer.Error!void {
        // Timestamp is units of microseconds.
        try writer.print("\nfault: timestamp = {d}.{d:0>}\\n", .{
            @divFloor(record.timestamp, time.us_per_s),
```

```

        @mod(record.timestamp, time.us_per_s),
    });
    try writer.print("cortex status:\n{f}\n", .{record.cortex_status});
    try writer.print("apollo3 status:\n{f}", .{record.apollo3_status});
}
};

```

```

<<fault service>>=
pub const CortexFaultRecord = struct {
    exc_frame: v7m.ExceptionFrame,
    exc_return: usize,
    ipsr: v7m.psr.ipsr,
    cfsr: v7m.SystemControlBlock.CfsrReg,
    hfsr: v7m.SystemControlBlock.HfsrReg,
    dfsr: v7m.SystemControlBlock.DfsrReg,
    mmfar: usize,
    bfar: usize,

    pub fn capture(
        exc_frame: *v7m.ExceptionFrame,
        exc_return: usize,
        ipsr: v7m.psr.ipsr,
    ) CortexFaultRecord {
        return .{
            .exc_frame = exc_frame.*,
            .exc_return = exc_return,
            .ipsr = ipsr,
            .cfsr = v7m.scb.cfsr.read(),
            .hfsr = v7m.scb.hfsr.read(),
            .dfsrr = v7m.scb.dfsr.read(),
            .mmfar = v7m.scb.mmfar.read(),
            .bfar = v7m.scb.bfar.read(),
        };
    }

    pub fn format(
        record: CortexFaultRecord,
        writer: *std.Io.Writer,
    ) std.Io.Writer.Error!void {
        try writer.print("{f}\n", .{record.ipsr});
        try writer.print("{f}\n", .{record.cfsr});
        try writer.print("{f}\n", .{record.hfsr});
        try writer.print("{f}", .{record.dfsr});
        if (record.cfsr.mmfsr.mm_ar_valid == 1) try writer.print("\nmfar: {x:0>8}", .{record.mmfar});
        if (record.cfsr.bfsr.bf_ar_valid == 1) try writer.print("\nbfar: {x:0>8}", .{record.bfar});
        try writer.print("\nexception return: {x}, ", .{record.exc_return});
        switch (record.exc_return) {
            0xfffffffff1 => try writer.print("Handler, MSP, no float", .{}),
            0xfffffffff9 => try writer.print("Thread, MSP, no float", .{}),
        }
    }
}

```

```

        0xfffffffffd => try writer.print("Thread, PSP, no float", .{}),
        0xffffffffe1 => try writer.print("Handler, MSP, float", .{}),
        0xffffffffe9 => try writer.print("Thread, MSP, float", .{}),
        0xffffffffed => try writer.print("Thread, PSP, float", .{}),
        else => try writer.print("unrecognized", .{}),
    }
    try writer.writeAll("\n");
    try writer.print("r0: {x:0>8}, r1: {x:0>8}, r2: {x:0>8}, r3: {x:0>8}\n", .{
        record.exc_frame.r0,
        record.exc_frame.r1,
        record.exc_frame.r2,
        record.exc_frame.r3,
    });
    try writer.print("ip: {x:0>8}, lr: {x:0>8}, pc: {x:0>8}\n", .{
        record.exc_frame.ip,
        record.exc_frame.lr,
        record.exc_frame.pc,
    });
    try writer.print("{f}", .{record.exc_frame.xpsr});
}
};


```

```

<<fault service>>=
pub const Apollo3FaultRecord = struct {
    fsr: apollo3.McuControl.FaultStatusType,
    icode_far: usize,
    dcode_far: usize,
    sys_far: usize,
    //reset_status: apollo3.ResetStatus.StatusType,

    pub fn capture() Apollo3FaultRecord {
        //const noinit_status =
        //    @extern(*NoinitSegmentStatus, .{ .name = "noinit_status", .linkage = .strong
    });
    return .{
        .fsr = fblk: {
            const status = apollo3.mcuctrl.fault_status.read();
            apollo3.mcuctrl.fault_status.writeDirect(1); ①
            break :fblk status;
        },
        .icode_far = apollo3.mcuctrl.icode_fault_addr,
        .dcode_far = apollo3.mcuctrl.dcode_fault_addr,
        .sys_far = apollo3.mcuctrl.sys_fault_addr,
        // TODO: need access to the noinit_status, right now its namespace is
        // unaccessible.
        // .reset_status = noinit_status.reset_status,
    };
}

pub fn format(
    record: Apollo3FaultRecord,

```

```

        writer: *std.Io.Writer,
    ) std.Io.Writer.Error!void {
        try writer.print("{f}", .{record.fsr});
        if (record.fsr.icode_fault == 1) try writer.print("\nicode_far: {x:0>8}", .{record.icode_far});
        if (record.fsr.dcode_fault == 1) try writer.print("\ndcode_far: {x:0>8}", .{record.dcode_far});
        if (record.fsr.sys_fault == 1) try writer.print("\nsys_far: {x:0>8}", .{record.sys_far});
    }
};

```

- ❶ The write after read is necessary to clear the fault status bits.

Fault Record Storage

In Chapter 2, [linker sections](#) that survive reset were set up and they are only initialized at power up. That memory is used to store the fault information. Following the same pattern used for panic record storage, we us an `ArrayList` to store fault records.

```

<<fault proxies>>=
const fault_msg_count: usize = 4; // need a config item for this
var fault_buffer: [fault_msg_count]FaultRecord linksection(".noinit_bss") = undefined;
var fault_storage: std.ArrayList(FaultRecord) = .initBuffer(&fault_buffer);

```

One particular, albeit rare, circumstance must be accommodated. It is possible for a system to get into a continual reset situation. Consider the following scenario. A system fault causes the execution to store away the status information and reset. The fault might be caused by a hardware malfunction. If the failure mode persists, the system may run for a short while and encounter the same situation which caused the original fault. This would cause another set of fault status information to be recorded and the processor would be reset. If this happens repeatedly, you end up with an overflowing fault status storage pool with very low quality information. It is likely that all the entries are the same except for the timestamp. To avoid this rare circumstance, the `BoundedArray` is used in the same manner as for panic records. In the case of overflow, the last value is simply overwritten and the previous N records remain unchanged.

Some systems may need to catch repeated reset scenarios and take action to completely shut down the system. This might be required to prevent damage to the either the system itself or damage to the environment where the system interacts. In those cases, a *rate* of reset may be calculated (from the timestamp and count information) and if it exceeds a threshold, additional actions can be taken to break the cycle and place the system in a mode where the resets stop. One possibility is to disable all interrupts and place the system in its lowest power mode. A continual reset cycle left unchecked can easily deplete a battery powered system. Such requirements are more stringent than implemented here.

Fault Record Retrieval

```

<<system_services: system service definitions>>=
/// Fault record retrieval service.
pub const fault = struct {
    <<fault service>>
};

```

We add the service name to the request enumeration.

```
<<sys svc service requests>>=
get_fault,
```

```
<<sys svc service request prongs>>=
.get_fault => &fault.getFault,
```

```
<<fault service>>=
/// The `getFault` service obtains the next stored fault record.
pub fn getFault(
    record: *FaultRecord,
) SvcError!void {
    const input_params = InputParam.marshal(.{
        .fault_record = record,
    });

    try sysRealmSvcCall(.get_fault, @ptrCast(&input_params), null, null);
}
```

Since this services is requesting an output, an `InputParam` is used to specify a pointer to where the notification should be placed.

```
<<fault service>>=
pub const InputParam = DefineSvcRequestParam(struct {
    fault_record: *FaultRecord,
});
```

```
<<system proxies>>=
const fault = struct {
    <<fault proxies>>
};
```

```
<<fault proxies>>=
const FaultRecord = svc_services.fault.FaultRecord;
```

```
<<fault proxies>>=
fn getFault(
    _: SysSvcRequest,
    input: ?*const anyopaque,
    _: ?*anyopaque,
    _: ?*anyopaque,
) SvcResult {
    const InputParam = svc_services.fault.InputParam;
    const input_params, const result = InputParam.receive(input.?);
    if (result != .success) {
        return result;
    }
}
```

```

const possible_fault_item = fault_storage.pop();
if (possible_fault_item) /*fault_item| {
    memcpyToUnpriv(mem.asBytes(input_params.fault_record), mem.asBytes(fault_item));
} else {
    return .operation_failed;
}

return .success;
}

```

missingHandler()

The implementation of the default exception handler is two steps.

1. Capture the fault status information and store it for later examination.
2. Terminate the program via a panic condition.

```

<<fault proxies>>=
fn missingHandler(
    exc_frame: *v7m.ExceptionFrame,
    exc_return: usize,
    ipsr: v7m.psr.ipsr,
) callconv(.{ .arm_aapcs = .{} }) void {
    const CortexFaultRecord = svc_services.fault.CortexFaultRecord;
    const Apollo3FaultRecord = svc_services.fault.Apollo3FaultRecord;

    var fault_rec = fault_storage.addOneBounded() catch blk: {
        _ = fault_storage.pop();
        break :blk fault_storage.addOneBounded() catch unreachable;
    };

    fault_rec.timestamp = epoch_time.deviceEpochTime();
    fault_rec.cortex_status = CortexFaultRecord.capture(exc_frame, exc_return, ipsr);
    fault_rec.apollo3_status = Apollo3FaultRecord.capture();

    var priv_console: PrivConsole = .init();
    var priv_writer = priv_console.writer(&.{} );
    const privOut = priv_writer.interface();

    priv_console.config.setTextColor(privOut, .bright_blue) catch {};
    privOut.print("{f}\n", &{fault_rec}) catch {};
    priv_console.config.setTextColor(privOut, .reset) catch {};

    var panic_msg: [] const u8 = undefined;
    if (ipsr.isr_number < v7m.system_exception_count) {
        const exc_number: v7m.SystemException = @enumFromInt(ipsr.isr_number);
        panic_msg = switch(exc_number) {
            .reset => "reset exception",
            .nmi => "nmi exception",
            .hard_fault => "hard fault exception",
        };
    }
}

```

```

        .mem_manage => "memory management exception",
        .bus_fault => "bus fault exception",
        .usage_fault => "usage fault exception",
        .sv_call => "supervisor call exception",
        .debug_monitor => "debug monitor exception",
        .pend_sv => "pend service exception",
        .sys_tick => "system tick exception",
        else => "reserved system exception",
    };
} else {
    panic_msg = "missing system exception";
}
panic.privPanic(exc_frame.pc, panic_msg);
}

comptime {
    @export(&missingHandler, {.name = "missingHandler", .linkage = .strong });
}

```

Panic Output Example

This example extends the Console I/O example to include interpreting the received input so as to generate panic conditions.

```

<<panic-console-test.zig>>=
//! This file contains a simple test application that causes system panic
//! conditions to demonstrate the system behavior during panic.
<<edit warning>>
<<copyright info>>

comptime {
    _ = @import("start_main");
}

const std = @import("std");
const mem = std.mem;
const log = std.log;
const svc_services = @import("svc_services");

pub const panic = svc_services.panic;
pub const std_options: std.Options = .{
    .logFn = svc_services.logToConsole,
};

pub fn main() !void {
    svc_services.Console.openAllConsoles();

    const sys_console = &svc_services.Console.sys_console;
    var writer_buf: [256]u8 = undefined;
    var sys_writer = sys_console.writer(&writer_buf);
    const sysOut = sys_writer.interface();

```

```

var reader_buf: [256]u8 = undefined;
var sys_reader = sys_console.reader(&reader_buf);
const sysIn = sys_reader.interface();

const usage =
    \usage:
    \i <index>          // write value at <index>, <index> > 15 ==> panic
    \r <addr>            // read word at <addr>
    \w <addr> <value>    // write word <value> to <addr>
    \p                // print saved panic messages
    \f                // print saved fault messages
    \x                // exit main with an error
    \?                // print usage message
++ "\n";

var test_buffer: [16]u8 = undefined;

while (true) {
    try sysOut.writeAll("All your codebase are belong to us.\n# ");

    const command = try sysIn.takeDelimiterExclusive('\n');
    if (command.len == 0) continue;
    var cmd_iter = mem.tokenizeAny(u8, command, " \t");
    const cmd_letter = cmd_iter.next().?[0];

    switch (cmd_letter) {
        'i' => {
            const next_arg = cmd_iter.next();
            if (next_arg) |index_arg| {
                if (std.fmt.parseUnsigned(usize, index_arg, 0)) |index| {
                    test_buffer[index] = 42;
                } else |err| {
                    log.err("failed to parse '{s}' as an unsigned: '{s}'", .{
                        index_arg,
                        @errorName(err),
                    });
                }
            } else {
                try sysOut.writeAll(usage);
            }
        },
        'r' => {
            const next_arg = cmd_iter.next();
            if (next_arg) |addr_arg| {
                if (std.fmt.parseUnsigned(usize, addr_arg, 0)) |addr| {
                    const ptr: *allowzero usize = @ptrToInt(addr);
                    const value = ptr.*;
                    try sysOut.print("value = 0x{x}\n", .{value});
                } else |err| {
                    log.err("failed to parse '{s}' as an unsigned: '{s}'", .{
                        addr_arg,
                        @errorName(err),
                    });
                }
            }
        },
    }
}

```

```

        });
    }
} else {
    try sysOut.writeAll(usage);
}
},
'w' => {
    const addr = try std.fmt.parseUnsigned(usize, cmd_iter.next().?, 0);
    const value = try std.fmt.parseUnsigned(usize, cmd_iter.next().?, 0);
    const ptr: *allowzero usize = @ptrToInt(addr);
    ptr.* = value;
},
'p' => {
    var panic_rec: svc_services.panic.PanicRecord = undefined;
    while (svc_services.panic.getPanic(&panic_rec)) |_| {
        try sysOut.print("panic: {f}\n", .{panic_rec});
    } else |err| {
        if (err != svc_services.SvcError.operation_failed) {
            log.err("failed to obtain panic record: '{s}'", {@errorName(err)});
        }
    }
},
'f' => {
    var fault_rec: svc_services.fault.FaultRecord = undefined;
    while (svc_services.fault.getFault(&fault_rec)) |_| {
        try sysOut.print("{f}\n", .{fault_rec});
    } else |err| {
        if (err != svc_services.SvcError.operation_failed) {
            log.err("failed to obtain fault record: '{s}'", {@errorName(err)});
        }
    }
},
'x' => return error.Exit,
'?' => try sysOut.writeAll(usage),
else => {
    log.err("unrecognized command '{c}'", .{cmd_letter});
    try sysOut.print("unrecognized command '{c}'\n", .{cmd_letter});
    try sysOut.writeAll(usage);
},
}
}
}

```

Summary

In this chapter, several functions that are used to catch unanticipated behavior have been provided. The `SystemMissingException()` function serves this role for exception processing and the `panic()` and `sysPanic()` functions provide similar abilities to software. To support the newlib “C” library, an implementation of `exit()` is also included to ensure that all fatal errors from the

standard library end up creating a panic condition and are caught. Invoking any of these functions is taken as non-recoverable and ultimately causes an abnormal termination. Their value lies in storing away the circumstances of the situation in memory that is not initialized upon routine resets.

Note this subject area is not entirely complete. No means have been provided to retrieve the data exception fault status. For the time being it is necessary to have a debugger or a connected terminal. This area is revisited later to complete the functionality needed for post-mortem examination of abnormal system termination.

Chip Whisperer

Modern microcontroller SOC's have an amazing number of integrated peripherals. But, there is such a large number of sensors and transducers, it is not practical to have them all integrated into an SOC. The solution is to have the means to connect external sensors to the microcontroller by some general means. That usually means via some serial protocol. As mentioned in [chapter 7](#), we have specialized peripherals that can generate the proper waveforms for communicating with external devices.

In this chapter, we develop an SVC service that can communicate using [I2C](#). I2C is a synchronous serial protocol designed for attaching peripheral devices. Our intended use for an I2C bus is to obtain data from transducers inserted into the environment and connected to the I2C bus. For example, temperature and real-time clocks are common I2C peripherals. We are not designing for large amounts of data transfer or high speed. An I2C standard mode bus runs at 100 kHz max, fast mode is 400 kHz max, and fast plus mode is clocked at 1000 kHz. These frequencies are the bit time frequencies clocked onto the physical serial interface, so overall throughput in bytes is closer to one tenth the bus frequency. The I2C bus is a two wire serial bus that runs on a so called *open drain* connection. The idle state of the bus is when both pins are high, implying that the bus is connected to a voltage with a pull-up resistor. The action of the controller and peripherals is to pull the bus signals low to indicate clock transitions and data values.

The I2C bus has several options that are useful at extending its utility in certain systems. For example, it is possible to have multiple bus masters on the same physical bus and they can arbitrate access to the bus.

The I2C standard only describes the wire protocol for the interconnect. There are several ways in which peripheral devices present their data interface on the bus. One common way is to address the peripheral and send it a “register offset” value. Then data is read or written to the implied register. Most devices support sending or receiving multiple bytes in a single bus transaction. In that case the peripheral device automatically increments the implied *offset* value for each successive byte after the first one.

For our purposes, we choose a simple set of requirements:

1. All I2C buses are single master.
2. Standard, fast, and fast plus mode speeds are supported.
3. Only 7-bit slave addresses are supported.
4. The Apollo 3 GPIO pads that are used for I2C support an internal pull-up resistors and we provide access to them.

Apollo 3 I/O Master Peripheral

The Apollo 3 has six replicated instances of a serial peripheral device that can be used for either SPI or I2C. Given the way the I/O pins were allocated when the chip is part of the Sparkfun Artemis Micromod processor board, only two I2C buses are exposed. If you follow the pin out diagrams for the Micromod board, it turns out that I/O Master #4 and #1 are the I/O Master peripherals that are accessible and intended for an I2C bus.

The I/O Master peripheral is a complex device. It supports transferring data by one of three methods.

Interrupt driven

Using the built-in 32 byte FIFOs as a data staging area the processor moves data on and off the

peripheral.

DMA driven

The peripheral supports DMA access to the FIFOs, alleviating any burden on the processor to move data.

Command queue driven

The peripheral supports reading register values from a command queue and writing those values into the peripheral to support a more streaming oriented interface.

Given that our requirements are not stringent, one might be tempted to use a simple interrupt driven approach. This was the approach used for the UART and in that case there was no choice since the peripheral did not support DMA operations. However, there is an erratum^[1] that regards the use of the threshold interrupt for servicing the I/O Master FIFOs. There is no resolution and no workaround is provided based on the reason that the HAL code does not use the FIFO threshold interrupt. This makes the interrupt driven approach unavailable: this implementation uses a DMA driven approach.

Representing the I2C Logical Device

Following our usual approach, the following figure shows how Apollo3 peripherals are assembled into a I2C logical device.

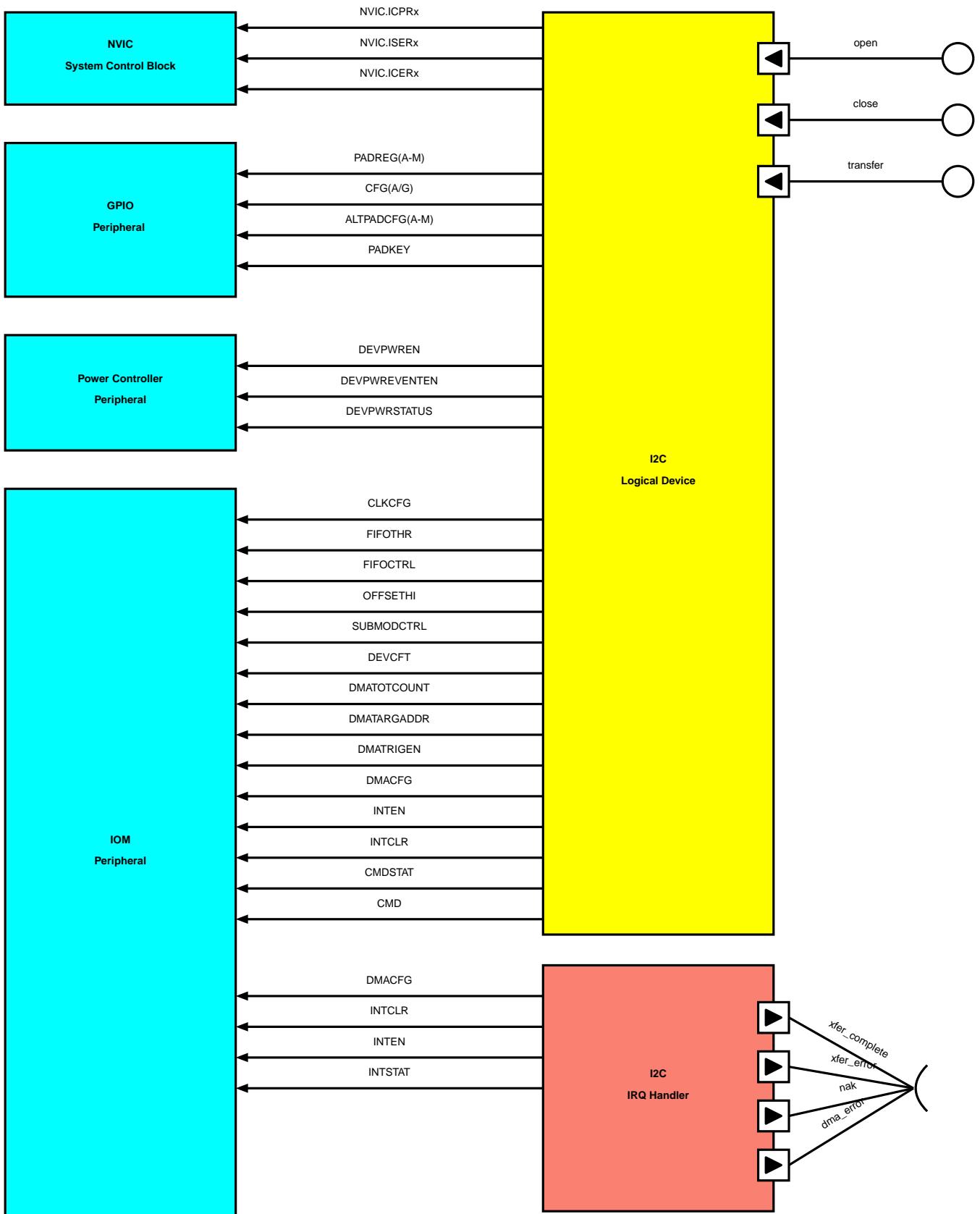


Figure 21. I2C Interface Peripheral Components

Since I/O Master peripherals are a common system resource, we create a device allocator for them.

```
<<device access>>=
const iom_allocator = DeviceAllocator(apollo3.IOMaster.module_count, &{});
```

We can represent the I2C device by the following data structure.

```
<<i2c proxies>>=
/// The core data structure used to control the I2C bus device.
const I2cDevice = struct {
    instance: DevSvcInstance,
    /// Data required to control the I/O Master peripheral.
    iom_control: IomControl,
    /// Data required to control the interactions with the NVIC.
    irq: IrqControl,
    /// Data control required to power the I/O Master up and down.
    power_control: PowerControl,
    /// Data to control the GPIO pins that connect the I/O Master to the I2C bus.
    pin_control: PinControl,
    /// Data required to control a transfer on the I2C bus.
    xfer_control: XferControl,
    /// The background notification proxy for I2C happenings.
    proxy: DevSvcNotifyProxy,
    /// A data value supplied by the user of the I2C device and returned in the
    /// background notifications from the I2C.
    closure: usize,
    <<i2c control functions>>
};
```

IOM Device Control

The Apollo 3 I/O Modules are relatively complicated peripherals since they support generating the serial waveforms for either I2C or SPI. Here, we provide the necessary controls over an I/O Module for I2C only.

```
<<i2c proxies>>=
/// The `IomControl` data and methods provide for direct control of the
/// IOM peripheral itself.
const IomControl = struct {
    iom: *volatile apollo3.IOMaster,
    <<iom control>>
};
```

The following table is from the Apollo3 datasheet and gives the recommended settings for various I2C speeds. We intend only to support standard mode (100 kHz), fast mode (400 kHz), and fast plus mode (1000 kHz) operation. These are the only I2C clocking modes supported by the IOM peripheral.

```
<<i2c proxies>>=
```

```
const I2cOperationMode = svc_services.i2c.I2cOperationMode;
```

Mode	FSEL	DIV3	DIV EN	TOT PER	LOW PER	SMP CNT	SDAEN DLY	SCLEN DLY	Notes
Standard (100 kHz)	1	0	1	243	159	12	15	0	Effective Freq 100 kHz
	2	0	1	121	79	6	15	0	Effective Freq 100 kHz
	3	0	1	60	39	3	15	0	Effective Freq 100 kHz
	4	0	1	30	19	1	15	0	Effective Freq 93.7 kHz
	5	0	1	16	9	1	6	0	Effective Freq 93.7 kHz, Low power
Fast Mode (400 kHz)	1	0	1	62	39	7	15	4	Effective Freq 400 kHz
	2	0	1	31	19	15	15	2	Effective Freq 400 kHz
	3	0	1	15	9	2	7	1	Effective Freq 375 kHz
	4	0	1	7	3	1	3	0	Effective Freq 375 kHz
	5	0	1	5	3	1	3	0	Effective Freq 375 kHz, Low power
Fast+ Mode (1 MHz)	1	0	1	24	12	1	7	0	Effective Freq 1 MHz
	2	0	1	12	6	1	6	0	Effective Freq 1 MHz
	3	0	1	6	3	1	3	0	Effective Freq 1 MHz
10 kHz	5	0	1	149	90	7	15	0	Effective Freq 10 kHz
50 kHz	4	0	1	59	35	3	15	0	Effective Freq 50 kHz

Figure 22. I2C Clock Speeds per Apollo3 datasheet

Interestingly, the HAL code does not use the suggestions from the data sheet. We have chosen to use the numbers from the HAL code for standard and fast modes based on the presumption that they work well. For fast plus mode, we use the datasheet numbers. For the HAL code the numbers are:

Mode	FSEL	DIV3	DIV EN	TOT PER	LOW PER	SMP CNT	SDAEN DLY	SCLEN DLY
std	2	0	1	119	59	3	15	0
fast	2	0	1	29	14	3	15	2
fast	4	0	1	6	3	33	3	0 (1)

Table 6. I2C Clock Speeds from Ambiq HAL code

(1) The HAL code states this configuration gives an I/O clock of approximately 860 kHz.

Using these values, we can create the necessary bitfield patterns that can be written directly to the IOM peripheral.

There are two registers that are required to set up the clocking and I2C bus properties.

```
<<i2c proxies>>=
const I2cSpeedConfig = struct {
    clk_cfg: apollo3.IOMaster.ClkCfgReg,
```

```
i2c_cfg: apollo3.IOMaster.Mi2cCfgReg,  
};
```

The register values are constant and kept in a simple array indexed by the operation mode.

```
<<i2c proxies>>=  
// The configuration values for the I/O clock and the i2c configuration come  
// from the Ambiq HAL code. These values do not correspond to suggested values  
// in the datasheet (no surprise there). We use the HAL values because we presume  
// they work, but previous incarnations of this code have used data sheet suggested  
// values and those seem to work also.  
const clk_config = enums.EnumArray(I2c0OperationMode, I2cSpeedConfig).init(.{  
    .std_mode = .{  
        .clk_cfg = .{  
            .io_clk_en = 0,  
            .fsel = .hfrc_div2,  
            .div3 = 0,  
            .div_en = 1,  
            .low_per = 119,  
            .tot_per = 59,  
        },  
        .i2c_cfg = .{  
            .addr_sz = .addr_sz7,  
            .i2c_lsb = .msb_first,  
            .arb_en = .arb_dis,  
            .sda_dly = 3,  
            .scl_en_dly = 0,  
            .sda_en_dly = 15,  
            .smp_cnt = 3,  
            .str_dis = 0,  
        },  
    },  
    .fast_mode = .{  
        .clk_cfg = .{  
            .io_clk_en = 0,  
            .fsel = .hfrc_div2,  
            .div3 = 0,  
            .div_en = 1,  
            .low_per = 14,  
            .tot_per = 29,  
        },  
        .i2c_cfg = .{  
            .addr_sz = .addr_sz7,  
            .i2c_lsb = .msb_first,  
            .arb_en = .arb_dis,  
            .sda_dly = 3,  
            .scl_en_dly = 2,  
            .sda_en_dly = 15,  
            .smp_cnt = 3,  
            .str_dis = 0,  
        },  
    },  
};
```

```

    },
    .fast_plus_mode = .{
        .clk_cfg = .{
            .io_clk_en = 0,
            .fsel = .hfrc_div4,
            .div3 = 0,
            .div_en = 1,
            .low_per = 3,
            .tot_per = 6,
        },
        .i2c_cfg = .{
            .addr_sz = .addr_sz7,
            .i2c_lsb = .msb_first,
            .arb_en = .arb_dis,
            .sda_dly = 3,
            .scl_en_dly = 3,
            .sda_en_dly = 3,
            .smp_cnt = 1,
            .str_dis = 0,
        },
    },
}),
);

```

```

<<iom control>>=
/// The `configClock` function installs the predetermined register values for
/// the supported I2C modes into the peripheral registers.
fn configClock(
    self: *const IomControl,
    transfer_mode: I2c0OperationMode,
) void {
    const config_info = clk_config.getPtrConst(transfer_mode);
    self.iom.clk_cfg.write(config_info.clk_cfg);
    self.iom.mi2c_cfg.write(config_info.i2c_cfg);
}

```

```

<<iom control>>=
/// The `configFifos` function configures the FIFO thresholds in the I/O Master
/// peripheral.
fn configFifos(
    self: *const IomControl,
) void {
    self.resetFifos();
    // 16 bytes => 4 words as suggested by the data sheet
    self.iom.fifo_thr.writeFields(.{
        .fifo_rthr = 16,
        .fifo_wthr = 16,
    });
}

```

```
<<iom control>>=
```

```

/// The `resetFifos` function clears all data from the FIFOs present in the
/// I/O Master peripheral.
fn resetFifos(
    self: *const IomControl,
) void {
    self.iom fifo_ctrl.clearBitField(.fifo_rstn);
    self.iom fifo_ctrl.setBitField(.fifo_rstn);
}

```

To support sending the “register offset” byte as part of a bus transfer, the I/O Master module will prepend 0 to 3 bytes of data in the transfer before sending or receiving any data. This interface is clumsy as it places bytes in different registers. The following function configures bytes 2 and 3 into the proper register.

```

<<iom control>>=
fn configHiOffsets(
    self: *const IomControl,
    offsets: []const u8,
) void {
    self.iom.offset_hi.writeFields(.{
        .offset_1 = offsets[1],
        .offset_2 = offsets[2],
    });
}

```

The I/O Master peripheral must be configured to be either an I2C or a SPI device. The following function always chooses the I2C configuration.

```

<<iom control>>=
fn enableI2cSubmodule(
    self: *const IomControl,
) void {
    self.iom.submod_ctrl.writeFields(.{ .smod0_en = 0, .smod1_en = 1 });
}

```

```

<<iom control>>=
fn disableI2cSubmodule(
    self: *const IomControl,
) void {
    self.iom.submod_ctrl.writeFields(.{ .smod0_en = 0, .smod1_en = 0 });
}

```

The I/O Master has a separate enable for the clock that generates the external signals to the bus.

```

<<iom control>>=
fn enableIoClk(
    self: *const IomControl,
) void {
    self.iom.clk_cfg.setBitField(.io_clk_en);
}

```

```
}
```

```
<<iom control>>=
fn disableIoClk(
    self: *const IomControl,
) void {
    self.iom.clk_cfg.clearBitField(.io_clk_en);
}
```

Bus transfers are always directed to a particular slave address, which is set by the following function.

```
<<iom control>>=
fn setSlaveAddress(
    self: *const IomControl,
    address: u10,
) void {
    self.iom.dev_cfg.writeField(.dev_addr, address);
}
```

As discussed [previously](#), this implementation uses the DMA capability of the I/O Master peripheral to transfer data to and from the I2C bus.

```
<<iom control>>=
fn configDma(
    self: *const IomControl,
    direction: XferDirection,
    buffer: []u8,
) void {
    self.iom.dma_tot_count.writeField(.tot_count, @truncate(buffer.len));
    self.iom.dma_targ_addr.writeDirect(@intFromPtr(buffer.ptr));
    self.iom.dma_trig_en.writeFields(.{ .dthr_en = 1 });
    const dma_config: apollo3.IOMaster.DmaCfgReg = .{
        .dma_en = 0,
        .dma_dir = if (direction == .write) .m2p else .p2m,
        .dma_pri = .low,
        .dpwr_off = .dis,
    };
    self.iom.dma_cfg.write(dma_config);
}
```

```
<<iom control>>=
fn enableDma(
    self: *const IomControl,
) void {
    self.iom.dma_cfg.setBitField(.dma_en);
}
```

```
<<iom control>>=
```

```

fn disableDma(
    self: *const IomControl,
) void {
    self.iom.dma_cfg.clearBitField(.dma_en);
}

```

There are a few complicating details of using the peripheral interrupts. The main problem is determining when the transfer is complete. If we are writing, DMA will complete transferring data into the FIFO before the command completes transferring the data from the FIFO to the I2C bus. Command complete is used to know when the transfer has finished. When reading, the opposite is true. The command will complete receiving data from the I2C bus into the FIFO before the DMA is can finish transferring the data from the FIFO to memory. So, DMA complete is used to determine the end of the transfer.

```

<<iom control>>=
fn enableIomInterrupts(
    self: *const IomControl,
    direction: XferDirection,
) void {
    self.clearIomInterrupts();

    var int_en: apollo3.IOMaster.IntReg = @bitCast(@as(mmio.WordRegister, 0));
    // Interrupts that indicate an error are:
    // .iacc, .icmd, .nak and .derr
    int_en.iacc = 1;
    int_en.icmd = 1;
    int_en.nak = 1;
    int_en.derr = 1;
    // The interrupt that indicates transfer completion.
    if (direction == .read) {
        int_en.dcmp = 1;
    } else {
        int_en.cmd_cmp = 1;
    }
    self.iom.int_en.write(int_en);
}

```

```

<<iom control>>=
fn disableIomInterrupts(
    self: *const IomControl,
) void {
    self.iom.int_en.writeDirect(0);
    self.clearIomInterrupts();
}

```

```

<<iom control>>=
fn clearIomInterrupts(
    self: *const IomControl,
) void {
    self.iom.int_clr.writeDirect(math.maxInt(mmio.WordRegister));
}

```

```
}
```

```
<<iom control>>=
fn pendingIomInterrupts(
    self: *const IomControl,
) apollo3.IOMaster.IntReg {
    const int_stat: mmio.WordRegister = @bitCast(self.iom.int_stat.read());
    const int_en: mmio.WordRegister = @bitCast(self.iom.int_en.read());
    return @bitCast(int_stat & int_en);
}
```

```
<<iom control>>=
fn isCmdIdle(
    self: *const IomControl,
) bool {
    return self.iom.cmd_stat.readField(.cmd_stat) == .idle;
}
```

```
<<iom control>>=
fn startXfer(
    self: *const IomControl,
    command: apollo3.IOMaster.CmdReg,
) void {
    self.iom.cmd.write(command);
}
```

GPIO Pin Control

Since the Apollo3 IO Module peripheral must be wired to the external environment, GPIO pins must be configured with the proper function and electrical characteristics. The I2C bus requires a pull-up resistor to operate properly. The Apollo3 SOC provides those resistors on those pins that can also function as SCL and SDA for the bus.

As was done in the device code for the UART, we must track through various pin labels to create a map between Micromod naming, Apollo 3 pin numbering, and the required pin function selection. Keep in mind that there are 6 IO Master modules on the Apollo 3, so we must make sure to choose the right combination of IO Module, pins, and pin functions.

In the Micromod configuration, two I2C buses are provided. They are labeled, interestingly enough, as “I2C” and “I2C1”. The tables below show the required pin and function select mapping for the Micromod I2C and I2C1 buses, respectively.

Micromod Pin#	Apollo3 Pin #	Function Select	I2C function	IO Master #
14	D39	4 (m4scl)	SCL	4
12	D40	4 (m4sdawir3)	SDA	4

Table 7. I2C Pin Mapping

Micromod Pin#	Apollo3 Pin #	Function Select	I2C function	IO Master #
53	D8	0 (m1scl)	SCL	1
51	D9	0 (m1sdawir3)	SDA	1

Table 8. I2C1 Pin Mapping

The I2C data and clock pins are organized as pairs. It is important to keep the pin numbers and function selections coordinated. To that end, we introduce a structure to handle the operations on the I/O pins as a pair.

```
<<i2c proxies>>=
const Gpio = apollo3.Gpio;
/// The `PinControl` type holds that information for handling the
/// I/O pin interactions required to attach the apollo3 IO Module to external pins.
const PinControl = struct {
    scl_pin: I2cIoPin,
    sda_pin: I2cIoPin,

    fn allocate(
        self: *const PinControl,
    ) !void {
        try self.scl_pin.allocate();
        errdefer self.scl_pin.free();
        try self.sda_pin.allocate();
    }

    fn free(
        self: *const PinControl,
    ) void {
        self.scl_pin.free();
        self.sda_pin.free();
    }

    fn configureI2cPins(
        self: *const PinControl,
        pull_up: ?apollo3.Gpio.PullupSelector,
    ) void {
        self.scl_pin.configure(pull_up);
        self.sda_pin.configure(pull_up);
    }

    fn disableI2cPins(
        self: *const PinControl,
    ) void {
        self.scl_pin.disable();
        self.sda_pin.disable();
    }
};
```

Each pin used for I2C ultimately uses the facilities of the GPIO proxy code to control the three registers that configure the I/O pin usage.

```
<<i2c proxies>>=
/// The `I2cIoPin` type collects together the information to control
/// the pins and pads required to run the I2C bus. Note the use of functions
/// provided by the `gpio` proxy that perform the physical I/O for
/// configuring GPIO pins and pads.
const I2cIoPin = struct {
    const PadConfigType = apollo3.Gpio.PadConfigType;
    const GpioConfigType = apollo3.Gpio.GpioConfigType;
    const AltPadConfigType = apollo3.Gpio.AltPadConfigType;

    pin: u8,
    pad_cfg: PadConfigType,
    pin_cfg: GpioConfigType,
    alt_pad_cfg: AltPadConfigType,

    fn allocate(
        self: I2cIoPin,
    ) !void {
        try gpio_allocator.allocate(self.pin);
    }

    fn free(
        self: I2cIoPin,
    ) void {
        gpio_allocator.free(self.pin);
    }

    fn configure(
        self: I2cIoPin,
        opt_pull_up: ?apollo3.Gpio.PullupSelector,
    ) void {
        var pad_cfg = self.pad_cfg;
        if (opt_pull_up) |pull_up| {
            pad_cfg.pull = 1;
            pad_cfg.r_sel = pull_up;
        }
        gpio.pin_control.configure(self.pin, pad_cfg, self.pin_cfg, self.alt_pad_cfg);
    }

    fn disable(
        self: I2cIoPin,
    ) void {
        gpio.pin_control.disable(self.pin);
    }
};
```

Transfer Control

To perform the data portion of an I2C bus transfer, we only need to know the direction of data flow and a pointer to the location in memory which is used as the input or output buffer for the transfer.

```
<<i2c proxies>>=
const XferControl = struct {
    direction: XferDirection,
    xfer_data: []u8,
};
```

I2C Logical Device Instantiation

With all the pieces at hand, we can capture the data that represents our logical view of the device.

```
<<i2c proxies>>=
var i2c_devices = [svc_services.i2c.instances]I2cDevice{
    .{ // I2C -- device instance 0
        .instance = 0,
        .iom_control = .{
            .iom = apollo3.iom[4], // IO Master 4 is wired to Micromod I2C bus
        },
        .irq = .{
            .irq_number = apollo3.IOMaster.irq_number[4],
        },
        .power_control = .init(.pwr_iom_4, .hcpc),
        .pin_control = .{
            .scl_pin = .{
                .pin = 39,
                .pad_cfg = .{
                    .pull = 0,
                    .inp_en = 1,
                    .fnc_sel = Gpio.padFunctionSelector(39, .m4scl) catch unreachable,
                    .r_sel = .@"1.5K", // ==0 and ignored since .pull is zero by default
                },
                .pin_cfg = .{ // no interrupts directly from the pin
                    .in_cfg = .read_en,
                    .int_dir = .int_lh,
                },
                .alt_pad_cfg = .{},
            },
            .sda_pin = .{
                .pin = 40,
                .pad_cfg = .{
                    .pull = 0,
                    .inp_en = 1,
                    .fnc_sel = Gpio.padFunctionSelector(40, .m4sdawir3) catch unreachable,
                    .r_sel = .@"1.5K",
                },
                .pin_cfg = .{

```

```

        .in_cfg = .read_en,
        .int_dir = .int_lh,
    },
    .alt_pad_cfg = .{},
},
},
.xfer_control = undefined,
.proxy = undefined,
.closure = 0,
},
.{ // I2C1 -- device instance 1
.instance = 1,
.iom_control = .{
    .iom = apollo3.iom[1], // IO Master 1 is wired to Micromod I2C1 bus
},
.irq = .{
    .irq_number = apollo3.IOMaster.irq_number[1],
},
.power_control = .init(.pwr_iom_1, .hcpb),
.pin_control = .{
    .scl_pin = .{
        .pin = 8,
        .pad_cfg = .{
            .pull = 0,
            .inp_en = 1,
            .fnc_sel = Gpio.padFunctionSelector(8, .m1scl) catch unreachable,
            .r_sel = .@"1.5K",
        },
        .pin_cfg = .{
            .in_cfg = .read_en,
            .int_dir = .int_lh,
        },
        .alt_pad_cfg = .{},
    },
    .sda_pin = .{
        .pin = 9,
        .pad_cfg = .{
            .pull = 0,
            .inp_en = 1,
            .fnc_sel = Gpio.padFunctionSelector(9, .m1sdawir3) catch unreachable,
            .r_sel = .@"1.5K",
        },
        .pin_cfg = .{
            .in_cfg = .read_en,
            .int_dir = .int_lh,
        },
        .alt_pad_cfg = .{},
    },
},
.xfer_control = undefined,
.proxy = undefined,
.closure = 0,
}

```

```
},  
};
```

I2C Services

The interface to the I2C device for background processing is provided by the following service functions:

- Open the I2C
- Close the I2C
- Transfer data on the I2C bus.

```
<<i2c services>>=  
pub const I2cOperations = DefineDeviceOperations(  
    .i2c,  
    &{ "open", "close", "transfer" },  
);
```

```
<<system_services: device service definitions>>=  
/// I2C services  
pub const i2c = struct {  
    <<i2c services>>  
};
```

```
<<i2c services>>=  
// Note only 2 IO Master modules can be used for I2C buses because of the way the  
// Apollo 3 is wired onto the Micromod processor module. The Micromod processor board  
// wires IO Master #4 and IO Master #1 to the external I2C connectors.  
pub const instances: DevSvcInstance = 2;  
pub const PullupSelector = apollo3.Gpio.PullupSelector;
```

```
<<device proxies>>=  
/// I2C service proxies  
const i2c = struct {  
    const XferDirection = svc_services.i2c.XferDirection;  
    <<i2c proxies>>  
};
```

I2C Open

As we have done with other devices, it is necessary to open the device so it may be initialized and made ready to transfer data.

```
<<i2c services>>=  
/// The `open` function makes the I2C bus master given by the `inst` argument  
/// ready to perform transfers. This function must be invoked before any  
/// transfer can take place. The I2C master is configured to the mode
```

```

/// given by `transfer_mode`. A pull-up resistor, available on the Apollo3
/// pin pad, is used if `pull_up` is given as non-null. If `pull_up` is
/// not specified, then it is turned off. Otherwise, `pull_up` specifies
/// which value of resistance is placed on the bus pins. Notifications are sent
/// to the background via the `proxy` and those notifications contain the
/// value of `closure`.
pub fn open(
    inst: DevSvcInstance,
    transfer_mode: I2cOperationMode,
    pull_up: ?apollo3.Gpio.PullupSelector,
    proxy: DevSvcNotifyProxy,
    closure: usize,
) SvcError!void {
    const input_params = OpenInputParam.marshal(.{
        .transfer_mode = transfer_mode,
        .pull_up = pull_up,
        .proxy = proxy,
        .closure = closure,
    });
    const request = I2cOperations.makeRequest(.open, inst);
    return devRealmSvcCall(request, @ptrCast(&input_params), null, null);
}

```

```

<<i2c services>>=
pub const I2cOperationMode = enum(u2) {
    std_mode = 0,
    fast_mode,
    fast_plus_mode,
};

```

```

<<i2c services>>=
pub const OpenInputParam = DefineSvcRequestParam(struct {
    transfer_mode: I2cOperationMode,
    pull_up: ?apollo3.Gpio.PullupSelector,
    proxy: DevSvcNotifyProxy,
    closure: usize,
});

```

```

<<i2c proxies>>=
pub fn open(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    _: ?* anyopaque,
    _: ?* anyopaque,
) SvcResult {
    const inst = req.instance;
    if (inst >= svc_services.i2c.instances) return .unknown_instance;

    const OpenInputParam = svc_services.i2c.OpenInputParam;
    const open_params, const result = OpenInputParam.receive(input.?);

```

```

if (result != .success) return result;

const idev = &i2c_devices[inst];

// Allocate the IO Master module.
// Note the function returns an enumeration value, so `try` and
// `errdefer` are not available.
iom_allocator.allocate(inst) catch return .operation_failed;
// allocate I/O Pins
idev.pin_control.allocate() catch {
    iom_allocator.free(inst);
    return .operation_failed;
};

// Power up IO Master module
idev.power_control.power_up();
// enable I2C submodule
idev.iom_control.enableI2cSubmodule();
// configure clocking for the requested transfer rate
idev.iom_control.configClock(open_params.transfer_mode);
// configure FIFO's
idev.iom_control.configFifos();
// enable the I/O clock
idev.iom_control.enableIoClk();
// configure I/O Pins
idev.pin_control.configureI2cPins(open_params.pull_up);

idev.proxy = open_params.proxy;
idev.closure = open_params.closure;

return .success;
}

```

I2C Close

After completing bus transfers, the device may be closed to reduce power consumption.

```

<<i2c services>>=
/// The `close` function decommissions the I2C bus given by `inst`,
/// cutting power to the peripheral device.
/// Programs seeking low power operation should close
/// the I2C bus after performing transfers across the bus.
pub fn close(
    inst: DevSvcInstance,
) SvcError!void {
    const request = I2cOperations.makeRequest(.close, inst);
    return devRealmSvcCall(request, null, null, null);
}

```

```

<<i2c proxies>>=
pub fn close(

```

```

req: DevSvcRequest,
_: ?*const anyopaque,
_: ?* anyopaque,
_: ?* anyopaque,
) SvcResult {
    const inst = req.instance;
    if (inst >= svc_services.i2c.instances) return .unknown_instance;

    const idev = &i2c_devices[inst];

    idev.iom_control.disableIomInterrupts();
    idev.irq.disable();
    idev.iom_control.disableIoClk();
    idev.iom_control.disableI2cSubmodule();
    idev.power_control.power_down();
    // N.B. we do not change the state of the I/O pins connected to the bus.
    // Such changes can cause spurious signals on the bus.
    idev.pin_control.free();
    iom_allocator.free(inst);

    return .success;
}

```

I2C Transfer

Finally, the transfer service performs the actual data transfer to or from the slave device.

```

<<i2c services>>=
/// The `transfer` function causes the I2C device to act as a bus master and
/// moves data onto or off of a slave device connected to the I2C bus.
/// The direction of data flow is determined by the `direction` argument.
/// The `.read` direction transfers data from the slave device to the bus master.
/// Conversely, the `.write` direction transfers data from the master to the
/// slave device. The `buffer` argument points either to a buffer where read
/// data is placed or from which write data is taken. The length of `buffer`
/// determines the length of the transfer. The `offset` argument points to
/// from 0 to 3 bytes which are transferred before any data.
/// Typical usage is that `offset` points to 1 or 2 bytes which form a _register offset_
/// informing the slave device which internal register is to be
/// affected by the transfer.
pub fn transfer(
    inst: DevSvcInstance,
    address: u7,
    offset: []const u8,
    direction: XferDirection,
    buffer: []u8,
) SvcError!void {
    if (offset.len > 3) return SvcError.invalid_param;
    if (buffer.len > 512) return SvcError.invalid_param;
    const request = I2cOperations.makeRequest(.transfer, inst);
    const input_params = TransferInputParam.marshal(.{

```

```

    .address = address,
    .offset = offset,
    .direction = direction,
    .buffer = buffer,
});
return devRealmSvcCall(request, @ptrCast(&input_params), null, null);
}

```

```

<<i2c services>>=
pub const XferDirection = enum(u1) {
    write = 0,
    read = 1,
};

```

```

<<i2c services>>=
pub const TransferInputParam = DefineSvcRequestParam(struct {
    address: u7,
    offset: []const u8,
    direction: XferDirection,
    buffer: []u8,
});

```

```

<<i2c proxies>>=
pub fn transfer(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    _: ?* anyopaque,
    _: ?* anyopaque,
) SvcResult {
    const inst = req.instance;
    if (inst >= svc_services.i2c.instances) return .unknown_instance;

    const TransferInputParam = svc_services.i2c.TransferInputParam;
    const input_params, const result = TransferInputParam.receive(input.?);
    if (result != .success) return result;
    if (input_params.offset.len > 3) return .invalid_param;
    if (input_params.buffer.len > 512) return .invalid_param;

    // Check that the boundaries of the data buffer are unprivileged
    // By accessing the first and last elements. This insures that
    // the background does not attempt to transfer data to/from
    // a privileged memory area.
    if (input_params.buffer.len > 0) {
        mem.doNotOptimizeAway(v7m.ldrbt(&input_params.buffer[0]));
        mem.doNotOptimizeAway(v7m.ldrbt(&input_params.buffer[input_params.buffer.len - 1]));
    }

    const idev = &i2c_devices[inst];
    if (!iom_allocator.isAllocated(inst)) return .operation_failed;
}

```

```

if (!idev.iom_control.isCmdIdle()) return .retry_operation;

// Fetch the register offset values from the unprivileged caller.
var reg_offsets = [3]u8{0, 0, 0};
memcpyFromUnpriv(&reg_offsets, input_params.offset);

idev.iom_control.configHiOffsets(&reg_offsets);
idev.iom_control.setSlaveAddress(input_params.address);
idev.iom_control.configDma(input_params.direction, input_params.buffer);

idev.xfer_control.direction = input_params.direction;
idev.xfer_control.xfer_data = input_params.buffer;

idev.iom_control.clearIomInterrupts();
idev.irq.clearPending();
idev.iom_control.enableIomInterrupts(input_params.direction);
idev.irq.enable();

const i2c_cmd: apollo3.IOMaster.CmdReg = .{
    .cmd = if (input_params.direction == .write) .write else .read,
    .offset_cnt = @truncate(input_params.offset.len),
    .cont = 0,
    .tsize = @truncate(input_params.buffer.len),
    .offset_lo = reg_offsets[0],
};

idev.iom_control.enableDma();
idev.iom_control.startXfer(i2c_cmd);

return .success;
}

```

I2C IRQ Handling

Each I/O Master device has a separate interrupt vector in the NVIC. We factor the processing in to a single function parameterized by the logical device instance number.

```

<<i2c proxies>>=
export fn ioMstr4IrqHandler() void {
    irqHandler(&i2c_devices[0]);
}

```

```

<<i2c proxies>>=
export fn ioMstr1IrqHandler() void {
    irqHandler(&i2c_devices[1]);
}

```

```

<<i2c proxies>>=
fn irqHandler(
    idev: *I2cDevice,

```

```

) void {
    const iom = idev.iom_control.iom;

    const irqs = idev.iom_control.pendingIomInterrupts();
    if (irqs.dcmp == 1 or irqs.cmd_cmp == 1) {
        idev.notifyToBackground(.xfer_complete);
    } else if (irqs.derr == 1) {
        idev.notifyToBackground(.dma_error);
    } else if (irqs.nak == 1) {
        idev.notifyToBackground(.nak);
    } else {
        idev.notifyToBackground(.xfer_error);
    }

    idev.iom_control.disableDma();
    iom.int_clr.write(irqs);
}

```

I2C Notifications

The I2C device issues notifications for the following reasons:

```

<<i2c services>>=
pub const NotifyReason = enum {
    xfer_complete, // transfer complete
    xfer_error, // transfer error on I2C bus
    dma_error, // DMA error transferring data to/from IO Master
    nak, // unexpected NAK on the I2C bus
};

```

In addition to the reason, the notification contains the direction of transfer and the slice pointer present in the transfer request.

```

<<dev notification specs>>=
.{ "i2c", struct {
    reason: i2c.NotifyReason,
    dir: i2c.XferDirection,
    msg: []u8
},
},

```

```

<<i2c control functions>>=
/// The `notifyToBackground` function sends a background notification to the
/// background proxy given by the `proxy` argument when the I2C device was opened.
/// Note this function is only called from IRQ level and does not implement
/// any exclusion with other proxy activities.
fn notifyToBackground(
    idev: *const I2cDevice,
    reason: svc_services.i2c.NotifyReason,
) void {

```

```

const result: SvcResult, const notification: *DevNotification =
    dev_notify_queue.reserveNotification();
if (result != .success) panic.privPanic(null, "cannot allocate I2C notification");

notification.* = DevNotification.init(
    .i2c, // operation
    idev.instance, // instance,
    idev.proxy, // proxy
    idev.closure, // closure
    .{
        .i2c = .{
            .reason = reason,
            .dir = idev.xfer_control.direction,
            .msg = idev.xfer_control.xfer_data,
        }
    }, // I2C specific notification information
);

dev_notify_queue.commitNotification();
}

```

Dispatching I2C Requests

In keeping with the device foreground proxy design, a class level handler is used to dispatch background requests to the appropriate proxy that handles the request.

```
<<svc device classes>>=
i2c,
```

```
<<dev svc service request prongs>>=
.i2c => &i2c.devClassHandler,
```

Finally, the I2C class handler performs the dispatch to the operation functions of requests.

```

<<i2c proxies>>=
/// The `devClassHandler` function dispatches watchdog operations based
/// on the value of the `req` argument. This function is invoked as a
/// class level dispatcher for the I2C device.
fn devClassHandler(
    req: DevSvcRequest,
    input: ?*const anyopaque,
    output: ?*anyopaque,
    err: ?*anyopaque,
) SvcResult {
    const Operation = svc_services.i2c.I2cOperations.Operation;
    const i2c_operation: Operation = @enumFromInt(req.operation);
    const proxy = switch (i2c_operation) {
        .open => &open,
        .close => &close,
        .transfer => &transfer,

```

```

    };
    return proxy(req, input, output, err);
}

```

Test cases

```

<<i2c_unit_tests.zig>>=
<<edit warning>>
<<copyright info>>

const std = @import("std");
const log = std.log;
const mem = std.mem;
const time = std.time;
const testing = @import("resee_testing");

const svc_services = @import("svc_services");
const i2c = svc_services.i2c;
const DevNotification = svc_services.DevNotification;
const exec_control = svc_services.exec_control;

<<i2c-test: declarations>>
<<i2c-test: tests>>

```

```

<<i2c-test: declarations>>
const rv_8803_addr = 0x32;

```

```

<<i2c-test: declarations>>=
const Rv8803 = extern struct {
    seconds: u8,
    minutes: u8,
    hours: u8,
    weekday: u8,
    date: u8,
    month: u8,
    year: u8,
};

```

```

<<i2c-test: declarations>>=
const i2c_transfer = struct {
    var rv_8803_regs: [0x30]u8 = undefined;
    var sync: bool = false;

    fn transferred_proxy(
        notification: *const anyopaque,
    ) void {
        const notify: *const DevNotification = @ptrCast(@alignCast(notification));
        const done: *volatile bool = @ptrFromInt(notify.closure); ❶
        done.* = true;
    }
};

```

```
    }
};
```

```
<<i2c-test: tests>>=
test "open i2c bus" {
    try i2c.open(
        0,
        .fast_mode,
        null,
        &i2c_transfer.transferred_proxy,
        @intFromPtr(&i2c_transfer.sync),
    );
}
```

```
<<i2c-test: tests>>=
test "read rv-8803 rtc" {
    testing.log_level = .info;

    for (1..20) |ndx| {
        i2c.transfer(
            0,
            rv_8803_addr,
            &.{0},
            .read,
            i2c_transfer.rv_8803_regs[0..ndx],
        ) catch |err| {
            log.err("{s}: with index: {d}", .{ @errorName(err), ndx });
        };
        try exec_control.runWait(&i2c_transfer.sync);
    }

    var date_time: Rv8803 = mem.zeroes(Rv8803);
    try i2c.transfer(0, rv_8803_addr, &.{0}, .read, mem.asBytes(&date_time));
    try exec_control.runWait(&i2c_transfer.sync);
    log.info("time: {x:0>2}/{x:0>2}/{x:0>2}T{x:0>2}:{x:0>2}:{x:0>2}", .{
        date_time.year,
        date_time.month,
        date_time.date,
        date_time.hours,
        date_time.minutes,
        date_time.seconds,
    });
}
```

```
<<i2c-test: tests>>=
test "write rv-8803 rtc" {
    testing.log_level = .info;

    var year = [_]u8{0};
    var other: u8 = 0;
```

```

// read the year
try i2c.transfer(0, rv_8803_addr, &.{6}, .read, &year);
try exec_control.runWait(&i2c_transfer.sync);
log.info("read year: '{x:0>}'", .{ year[0] });
// increment by 1
year[0] += 1;
other = year[0];
// write year back
try i2c.transfer(0, rv_8803_addr, &.{6}, .write, &year);
try exec_control.runWait(&i2c_transfer.sync);
// read again
try i2c.transfer(0, rv_8803_addr, &.{6}, .read, &year);
try exec_control.runWait(&i2c_transfer.sync);
log.info("read back year: '{x:0>}'", .{ year[0] });
try testing.expectEqual(other, year[0]);
// decrement by 1
year[0] -= 1;
other = year[0];
// write year back
try i2c.transfer(0, rv_8803_addr, &.{6}, .write, &year);
try exec_control.runWait(&i2c_transfer.sync);
// read back
try i2c.transfer(0, rv_8803_addr, &.{6}, .read, &year);
try exec_control.runWait(&i2c_transfer.sync);
log.info("third year read: '{x:0>}'", .{ year[0] });
try testing.expectEqual(other, year[0]);

// write year back '25' to keep the year correct
year[0] = 0x25;
try i2c.transfer(0, rv_8803_addr, &.{6}, .write, &year);
try exec_control.runWait(&i2c_transfer.sync);
}

```

```

<<i2c-test: tests>>
test "read/write rv-8803 ram location" {
    var out_buf: [1]u8 = .{0};
    var in_buf: [1]u8 = undefined;
    // repeat several times to bang on the interface
    for (0..20) |_ {
        while (out_buf[0] != std.math.maxInt(u8)) : (out_buf[0] += 1) {
            try i2c.transfer(0, rv_8803_addr, &.{7}, .write, &out_buf);
            try exec_control.runWait(&i2c_transfer.sync);
            try i2c.transfer(0, rv_8803_addr, &.{7}, .read, &in_buf);
            try exec_control.runWait(&i2c_transfer.sync);
            try testing.expectEqual(out_buf[0], in_buf[0]);
        }
    }
}

```

```
<<i2c-test: tests>>
```

```
test "close i2c bus" {
    try i2c.close(0);
}
```

Conclusion

This is the end of Part I and it is appropriate to review what has been covered. There are three primary subjects in Part I.

1. Foundational systems programming for the ARM Cortex-M4 core.
2. Design of a mechanism to support the concurrency associated with handling the system environment.
3. System specific programming to incorporate the Apollo 3 peripheral devices.

Systems Programming of the Core

The Cortex-M4 core, like any other processor core, requires specific code to use its capabilities. As microcontroller cores go, the Cortex-M4 is quite capable, providing several useful features which can be used to design more robust execution environments. Compared with older 8-bit and 16-bit microcontrollers, the v7-M architecture brings capability usually found in general purpose computers. The design of the core determines those capabilities and several choices were made as to how the core is operated. Specifically:

Dual stacks

Two stacks were used to have better flexibility in allocating stack memory. A two stack arrangement makes it easier to handle the required stack space for higher priority exceptions and for those exceptions whose priority is fixed. The Hard Fault and NMI exceptions do not have configurable priorities and the fixed priority assigned to them is higher than the priority of other configurable exceptions. This means that there must always be sufficient space on any stack to accommodate handling those exceptions. By separating the Main Stack from the Process Stack, the Process Stack does not need to account for the stack usage that might happen as a result of preemption of one exception by another. Once one first exception is taken, any subsequent exception that runs as a result of preemption uses the Main Stack. The Process Stack then only needs to be sized to handle the requirements of the application plus the space required to stack the core context when any exception is taken. Sizing the Main Stack can be tailored to the exception priority scheme used. In this design, the [priority scheme](#) has only four possibilities for preemption: fixed priority system faults, configurable priority system faults, debug monitor, and interrupt requests. The two stack design also interacts with the use of memory protection ensuring that unprivileged code does not have access to the Main Stack.

Privileged execution

Execution privilege was separated so that application logic runs in an unprivileged manner. This separation is essential for general purpose operating systems and beneficial for managing the run time of a microcontroller. This is in keeping with the emphasis on separating system level concerns. Interactions with hardware peripherals and the external environment are considered privileged. The intent is to be able to build well honed and tested privileged code which changes little. Application logic varies as the primary purpose of the system and is subject to more frequent changes.

Memory protection unit

The MPU is used to enforce access to memory based on the separation of execution privilege. The scheme used in this book is simple and directly based on the execution privilege and the use of the memory by the running program. This accounts for the difference in instructions and data as well as the difference between flash memory and RAM.

System Environment Concurrency

A goal of the design in Part I was to manage the concurrency of the environment using the capabilities of the core and a minimum of additional software. The central concept is to use a notification queue as the mechanism to transport data and control into the application processing.

The separation of privileged and unprivileged execution necessitated the use of the SVC instruction to allow unprivileged code to request privileged operations. Those requests run to completion before returning to the requesting background code.

Concurrent interactions with the system environment are represented by interrupt requests. Handling an IRQ can result in additional computing that needs to be done in the background. Deferring execution to background code shortens the execution time of the IRQ handler and so lowers the latency of responding to another interrupt. The Background Notification Queue is the mechanism where notifications are sent to the background for further action. Because the timing of arrival of interrupt requests is often not predictable, the queue serves as a buffer, allowing the IRQ handling and the subsequent background processing to be disconnected in time from each other.

The particulars of the operation of the notification queue allow flexibility in the size and contents of the notification messages. The queue is the only mechanism provided or needed for foreground processing to interact with background processing. In particular, *ad hoc* methods with shared global data between foreground and background are unavailable because of execution privilege separation. This necessitated a design of the notification queue that is completely independent of the subject matter of the application.

Peripheral Device Operations

Since microcontroller-based reactive systems depend heavily on interacting with the system environment through their peripheral devices, several chapters were devoted to demonstrating how the concurrency scheme embodied in the Background Notification Queue could be used to present logical devices for use by application software.

The first example shown, a watchdog timer, was based on a timer peripheral as a matter of convenience. Timers used for generating a stimulus do not have physical connections to the outside world. They simply count the ticks in the “ether.”

Connections to the outside world are important, so control of simple GPIO pins was also demonstrated. Since pins are shared between direct external connections and peripheral device needs, a simple allocation scheme was developed to support the needs of both.

Time is also important and several aspects of supplying time-based interactions were shown. A timer queue was developed to support software execution requirements and time representations based on calendar time and timestamp notions were developed.

Finally, UART and I2C communications using dedicated peripherals were demonstrated. These communications capabilities are fundamental to further development by providing interfaces both to humans and to additional hardware peripherals.

In future parts of the book, additional peripheral device control will be shown. Device control always makes up a substantial part of a microcontroller system. Such code is often more difficult to produce since it involves not only complex logic of the peripheral device but often considerations of timing and scale must be accommodated. Note that there has been no attempt to create a Hardware Abstraction Layer, *per se*. This is because no attempt has been made to support peripheral device functionality that did not have an immediate use. There are clear places in the design where the direct operations on a peripheral device were factored out. But there is no

concerted effort to make every possible configuration of a peripheral accessible from a function in the code. This is an effort to create a running system, not a software development kit (SDK).

Baton

Abstract

This part of the book shows an execution model that works together with the core system from Part I to supply the necessary mechanisms to execute background processing. Part II introduces the notion of a higher-order execution model and then provides an implementation of that model. It introduces the Reactive System Execution Environment (ReSEE) as the means to execute background processing in response to environmental happenings detected in Part I. ReSEE factors all the data management and execution sequencing away from the application logic and provides a consistent and constrained set of mechanisms to be used in computing responses to the system environment. The data management facilities are based on the relational model of data and a complete relational algebra implementation is provided. Execution sequencing is based on finite state models. This part provides all the necessary infrastructure to interface to the control mechanisms of Part I.

The View from the Mesa

Part I of this book discussed how to program a microcontroller starting with only basic tools. That part accomplished three primary goals:

1. Basic systems programming of the processor core to create the desired base execution environment. This included designing which aspects and features of the Cortex-M4 core were used and building a foundational body of code to support a constrained execution.
2. A mechanism to support the concurrency of the system environment. This mechanism was based on interrupts, execution priority, and the use of a background notification queue as a means of continuing processing which was started by an interrupt request.
3. Peripheral device control code for a small set of basic peripheral devices. Included in the peripheral devices were timers, GPIO pins, and a UART.

This body of code allowed us to take the first step toward a fully functioning system. The design of the code in Part I was in many ways constrained by the capabilities of the microcontroller architecture. The ARM® v7e-M architecture provides a given set of processor mechanisms to implement a wide variety of system execution schemes. Much of the design effort was spent mapping the microcontroller capabilities to our goal of handling the concurrency present in the environment. The design split processing between privileged foreground code and non-privileged background code. Background code interacts with the foreground code by means of the SVC exception. Foreground code interacts with the background code using a queue. A queue is a clear choice to serialize the notifications of environmental happenings by privileged code, e.g. as detected by a transducer inserted into the environment, while background processing is computing a response. The ordering of the queue helps construct a deterministic path for processing order.

On several occasions, Part I stated that the execution mechanism used to handle the system environment was insufficient for a complete system. In Part II, the next step is taken which will enable the system to scale up its interactions with the environment by providing an execution scheme for background processing. From a program point of view, Part I includes a *weak definition* of a function called, `serviceBackground`. To cause the system to execute, we must provide a strong definition of `serviceBackground` and arrange for the `runWait` function to be invoked after any initialization code has prepared the system to run. The `serviceBackground` is the sole interface provided to resolve foreground processing requests.

At this point we are somewhat at a loss to say what requirements drive the design of background processing. We have moved out of the realm of hardware dictates and must devise a scheme capable of running applications that fall into a broad class of reactive systems as we have defined them.

Design decisions for a background execution scheme can't be derived by a mapping onto hardware architecture. Rather, we must have some knowledge of the execution model we are attempting to implement. That can only come from "higher up" the logic chain. Those requirements are known, but since we are coming from the bottom up, we postpone the discussion of why the particular execution model we are about to implement has been chosen. The chosen execution model has deep roots in predicate logic and finite state automata, but it remains until Part III to show how the pieces come together to provide a complete implementation of the model execution concepts. Here, we must deal with being in the middle, mapping higher-order execution concepts onto the system base generated in Part I.

Core Execution Model Concepts

To perform a mapping from higher-order execution concepts onto the base system from Phase I, it is

necessary to state exactly what the higher-order concepts are. The statement here does *not* attempt to rationalize or describe the concepts in detail. We take the following in an axiomatic sense.

- A system is composed of a set of interacting *domains*.
- The criterion for decomposing the system into domains is *subject matter*. The use of *domain* here is in the sense of [Domain of Discourse](#).
- The subject matter content of the system domains is a creative output of analyzing the system requirements and capturing the analysis results as a domain model. This is distinctly different pattern of analysis than those techniques based on functionality or flow of execution.
- The decomposition of a system into domains is expected to yield a *separation of concerns* between the various domains.
- The separation of concerns exhibited by the domains is expected to yield a *semantic gap* between their interactions.
- The dependencies between domains are described in terms of the *requirements* that domains impose upon each other. The domain requirement dependencies form an acyclic directed graph.
- Domains interact by means of *bridges*. Bridges close the semantic gap between domain interactions and map the domain dependencies onto the flow of control and data between the domains.
- A domain is described by a *model* and consists of three facets or projections of the domain subject matter. Each facet is modeled separately as follows:
 1. Data: The data facet of a domain is modeled based on the [relational theory of data](#).
 2. Dynamics: The dynamics facet is modeled by extended [Moore-type state models](#).
 3. Processing: The processing facet is modeled based on data flow semantics of [process algebra](#).
- The domain is the unit of encapsulation.
- The domain is the unit of reuse.
- A domain implementation is characterized as *translated* or *realized*. A translated domain is constructed by a mechanical transformation of a domain model. A realized domain is constructed by supplying necessary interfacing code to otherwise conventionally coded software. Real-world systems are almost always a mixture of both, with realized domains often forming a direct connection to the system environment's control mechanisms.

Reactive System Execution Environment (ReSEE)

In keeping with the above model concepts, Part II shows the design and implementation of a *Reactive System Execution Environment* (ReSEE). The focus of this part of the book is on those model concepts which contribute to program execution, namely the three facets of a domain model:

1. Data,
2. Dynamics, and
3. Processing.

In keeping with our previous approaches, the target is primarily microcontroller-based systems with support for small to medium scale systems deployed on conventional computing environments. The later are intended as testing and simulation environments for the microcontroller-based systems.

Note, this is not an attempt to create a universal execution environment. Different engineering

goals and constraints require different engineering trade-offs to realize an efficient solution even if the abstract execution model is the same. For example, here we provide no external persistent data storage. That would be a serious shortcoming for most commercial data processing applications. It is necessary to choose an execution environment that fits the computational demands of the target class of applications. Note, this is *not* to say that every application needs a custom tailored execution scheme. Application demands on the computing environment fall into broad categories that are generally based on memory and concurrency demands.

It is fitting to reiterate that our intent here is to factor *all* management of data as well as *all* sequencing of control and synchronization into the ReSEE. The goal is to remove from the application logic the implementation specifics of how data is allocated, managed, and accessed. Application code has a limited set of data management capabilities available to it. Similarly with execution sequencing, the application code does not directly decide which computation to perform next^[1]. It does have the capability to request actions be taken, but the decision as to what computation is next dispatched is done by the ReSEE. This approach is challenging both to understand and to implement, and represents a distinctly different computational approach. Our goal in the approach is to control complexity by separating application logic, which we consider as essential complexity, from computational mechanisms, which we consider as accidental complexity.

Because the targets are reactive systems, we place an additional requirement on ReSEE to be deterministic. The response of the system is to be strictly driven by the timing of arrival of happenings in the system environment. This is to say that if the same conditions arise in the system environment with the same arrival times, the system must respond in the same manner.

The next chapter is a gentle introduction to the idea that programming involves a mapping from problem understanding onto a model of execution. Afterwards, we launch into the chapters that describe and implement the data and execution management facilities of ReSEE.

By the end of Part II, we will have the necessary function to resolve the weak linkages needed by the code from Part I as well as functions to be used to manage data and sequence control for the application logic.

Additional Background for Readers

Because additional subject matters are explored, additional background is required from the reader. In Part I, readers had their knowledge of both Zig and ARM microcontrollers tested. The particular processor chosen for the implementation was based on the ARM® v7e-M architecture and knowledge of its details was helpful. In Part II, there is additional background knowledge that is assumed of a reader, namely:

- The data management scheme described in Part II is based on the [relational model of data](#). Some readers will be familiar with the relational model as the underlying data model for Relational Database Management Systems (RDMS). We organize all the application data relationally and provide a relational algebra to manipulate the data. Note we are **not** attempting to create a RDMS. For the target systems considered here, all data is held in memory and there is no automatic persistence of application data to external storage. The relational model provides a uniform way to encode application logic into data and to draw logical conclusions using well defined algebraic operations on the data.
- Execution sequencing and synchronization is accomplished using event-driven finite state models. In particular, extended [Moore type models](#) are used to implement life cycle behavior.
- The [Zig](#) language is used as the implementation language. Zig is a relatively new systems programming language that has many of the characteristics of “C” required for systems programming, solves many of the deficiencies of “C”, and can interact with “C” code directly. Zig has shown itself as a capable tool to solve elegantly several situations where “C” would

create a more complicated solution. Zig has several distinctive capabilities that are useful when coding the relational algebraic operations. Relational operations are often *type generators*, i.e. the result of the operation is a relation with a different heading which is a different static data type. Fortunately for our usage, all the relation definitions and the operations used are known at compile time. With Zig, it is possible to execute ordinary Zig code at compile time. This combined with language built-in functions allows for the creation of new data types at compile time. The Zig language also supports a form of compile time “[duck typing](#)” which allows the compiler to generate type specific code for generated types. These language features that support compile time type generation allow much easier implementations of relational algebra operations than “C” and is a primary reason for using the language. As we have seen in Part I, Zig is also a capable language for dealing directly with the processor’s operation. For “C” programmers, the syntax is familiar and access to and control of memory is part of the language. However, there are many differences and readers will need some familiarity with Zig if the code sequences are to be fully understood.

We understand this requires significant additional background knowledge. In keeping with the suggestions from Part I, read by skipping around. Revisit areas of interest which required extra background knowledge once you have acquired the necessary background.

[1] Albeit application code can invoke functions and procedures directly without other intervention

The Execution Model

Executing with the Cerebral Computer

Despite its continued use by journalists, comparing a computer to a human brain stretches the analogy beyond recognition. A computer does not function on the same principles as a human brain. The human brain represents a marvelous biochemical solution evolved over hundreds of millions of years of natural selection. Computers, by comparison, are a crude collection of electronic switches. What computers possess is astounding speed. What both possess is the ability to evaluate logic, albeit by distinctly different mechanisms.

As programmers, a lot of time is spent executing code in our head. Whether writing or reading computer code, the logic and actions in the code statements are verified against our understanding of what is needed to solve a particular part of some larger software problem. Consider the following simplistic example in “C”.

Example 1. Point on a line — integer

```
int
point_on_line_int(
    int x,
    int slope,
    int intercept)
{
    return (x * slope) + intercept ;      // yes, the parentheses are not necessary
}

int y = point_on_line_int(4, 10, 2) ;
```

Anyone with only a passing knowledge of “C” would be confident that the value contained in the variable named, **y**, is 42. But we **know** that real computers do not operate exactly in accordance with the way the “C” source is stated. Computer memory is not typed. It may hold integer values, floating point values, code, character data, or even program instructions themselves. For computers, instructions have type in the sense that machine instructions *assume* that the operands have specific encodings.

It is instructive to examine the assembly language output of the compilation of the above code.^[1].

Example 2. Point on a line — integer, assembly

```
point_on_line_int:
    mla r0, r1, r0, r2
    bx lr
```

The result is somewhat surprising in that only a single instruction is generated for the arithmetic (the **bx lr** instruction returns from the function). The ARM Cortex v7e-M instruction set has a single instruction which performs both the multiplication and the addition. This usage of the **MLA** instruction multiplies the value in R1 times the value in R0 and adds to the product the value in R2, placing the result in the R0 register. This is a so-called *multiply and accumulate* operation which

occurs frequently enough that the instruction set designers decided it was worth its own instruction. If we carefully contrive to put the 2's complement representation of 4 into R0, of 10 into R1, and of 2 into R2, then when the function returns, the value of 42 will be present in the R0 register. Notice the register usage follows the AAPCS which we have encountered previously.

However, if the three registers used by the MLA instruction contain something other than integer data, say bytes from a character string, the instruction executes and produces nonsense. The MLA instruction does not generate any exceptions. You simply get the wrong answer since multiplication and addition are not operations with a defined meaning when applied to character data. The result is probably not what was intended to be correct. If the careful contrivance of how data is placed in the R0 - R2 registers is not correct, you have a bug which the processor core itself cannot help diagnose. Applying and tracking the types of variables is one, among many, of the difficulties associated with programming in assembly language.

Even if the values in R0 - R2 are in 2's complement format, a wrong result may still happen. Since the result occupies a 32-bit register, overflow is possible. The result of the operation is performed modulo 2^{32} . A status flag is set to indicate overflow, but there is no direct way to make use of that information in the program. Further, overflow may be an acceptable result in some circumstances.

It is also instructive to examine what happens if the data types are changed from `int` to `float`.

Example 3. Point on a line — float

```
float
point_on_line_float(
    float x,
    float slope,
    float intercept)
{
    return (x * slope) + intercept ;
}
```

The only textual change is that the variables are declared as `float` rather than `int`. Note in particular that the expression computing the function result is the same text as in the integer case. Compilation yields a distinctly different result.

Example 4. Point on a line — float, assembly

```
point_on_line_float:
    vmul.f32    s0, s0, s1
    vadd.f32    s0, s0, s2
    bx         lr
```

This result matches expectations. There is an instruction for multiplying (VMUL.F32) the slope and x values and another instruction (VADD.F32) for adding the intercept value to the product. But the instructions are completely different from those of the integer case. There is even a different set of registers in use, namely S0, S1, and S2. The processor core contains a completely different hardware component to perform floating point arithmetic. That component includes both a separate set of registers and instructions to handle the complexity of the detailed logic of floating point arithmetic. It is also the case that floating point arithmetic is an optional component of the

processor core. Keeping the required registers and instructions separate makes implementing such an option simpler.

The important point here is that the text of the expression, `x * slope + intercept`, is exactly the same in both cases. This means that multiplication and addition are language supported *polymorphic* operations on the fundamental, core-supplied integer and floating point data types. The compiler *chooses* machine instructions based on how the data is encoded in memory. One of the fundamental mappings the compiler makes is to impose a type system, track the data types associated with data stored in memory, and select the correct *instructions* based on the value types of the operands.

Computational Models of Languages

The discussion in the previous section simply reiterates what programmers already know about the difference between operations performed by a physical computing machine and those operations as they appear within the execution model of a programming language. Executing code in your head carries with it an implicit model of computation which is supplied by the particular language in which the program is encoded. All languages carry a set of *semantics* which reflects the computational model supplied by the language. The semantics of a programming language must be *mapped* onto the set of instructions provided by the processor and that mapping is performed by a compiler^[2]. The most often implemented model is sequential execution, *i.e.* one statement is executed after another with appropriate language constructs for conditional execution and repeated execution.

Conventional wisdom holds that the “C” language is “low level” or “close to the machine.” This is true in the sense that “C” provides access to bit level manipulations and to the layout of memory. Other, more recently defined languages, provide additional capabilities and the language processors perform the added mapping to implement the program semantics using machine instructions. Polymorphic overloading of functions based on parameter type information is a simple example. This construct provides the compiler with sufficient information to generate different function invocations depending upon the types of the function’s parameters. As processors have become more capable, language processing tools have become more advanced and take on a greater role of mapping more higher-order programming strategies onto the underlying machine language.

The computational model provided by a language has a profound and unavoidable influence on our approach to solving software problems and, consequently, on how we conceive the specifics of a software solution which must use those facilities provided by the language. As a group, programmers often have ardent preferences for one language over others. The popularity of specific programming languages is measured, tracked, and discussed with great regularity and seemingly unending energy. The computational models of some languages even evoke cynical comments:

It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration. ^[3]

— Edsger Dijkstra, How do we tell truths that might hurt?

With the proliferation of languages now available, another Dijkstra quote seems appropriate.

About the use of language: it is impossible to sharpen a pencil with a blunt axe. It is equally vain to try to do it with ten blunt axes instead.

— Edsger Dijkstra, How do we tell truths that might hurt?

Higher Order Computational Models

It is an inconvenient truth that the size of a program eventually limits our ability to understand it when it is expressed in a general-purpose programming language. The sheer number of statements needed to solve large software problems eventually overwhelms our mental capabilities to understand the entire solution.

The usual strategy to handle this situation is to undertake some form of the *divide and conquer* strategy. The simple example of the previous section, showed the use of functions as a means grouping computation that can be invoked from multiple call sites. The ability to have sub-programs and to control the execution of a program to traverse the implied call graph is a universal capability^[4] of all programming languages.

Other techniques at modularization are common. For “C”, the modularization techniques are simplistic and are directly associated with compiling from the data files provided by the underlying operating system. Libraries are little more than a packaging of several compiled object files into a single archive file. Other languages provide much more capable *module* systems. Languages must be accompanied by additional tools than just the compiler to manage ever larger volumes of program code. Current trends in language implements recognize the need for the additional tooling required to write large programs and the language implementations provide an *ecosystem* to support program development.

Yet even in the face of better tooling, our ability to understand how the functional requirements of a program are realized in a computer language implementation are still subject to being overwhelmed by the scale of a program. Despite recent attentions in the greater programming world for more *readable* code, one goal for sustainable programs is system *understandability*. The goal here is make progress toward understandability by using a higher order computational model which can be *translated* into the specifics of any given programming language.

It is important to be precise about our definition of the term, *model*. All mature engineering disciplines use some form of model to accomplish a separation of problem solution from its implementation. Architects use scale models for buildings. Construction of buildings uses blueprints. Electrical engineers use schematic diagrams. Civil and mechanical engineers use extensive 3-D static and dynamic modeling. Chemical engineers build pilot plants before any attempts at a large scale processing plant.

Of course engineering software is different from other disciplines, but we must still contend with our limitation of conceiving large scale problem solutions. For our purposes, a model is a view of software system which *systematically* excludes certain aspects of the system to focus better on other aspects. Specifically, we exclude all aspects of the implementation technology of the software system to better focus on capturing the underlying application logic.

A software model which excludes aspects of the problem implicitly implies some mechanism whereby the exclusions are systematically resolved to achieve the complete software solution. Indeed, we have several criteria for what is acceptable as modeling for software solutions.

1. A software model must predict the functional actions^[5] of the resulting program. This implies that the model can be executed in your head in a manner analogous to executing programming language statements in your head and that the model can serve as a functional prototype of the desired system.
2. A software model must directly support the creation of the final program solution. It makes little sense to build software artifacts which do not directly contribute to the final solution.

The next two sections are an introduction to the ReSEE execution model. The next section presents the idea of encoding the logical predicates of the domain subject matter into data. The second

section shows how computation is sequenced and synchronized. Afterwards, we dive into the details of managing data and execution to show how programming with sets and state models gives much greater leverage over capturing a problem solution.

Encoding Logic in Data

In Part I, a simple example that [blinked an LED](#) at a programmable rate was demonstrated. This example was simple enough that it could run entirely as part of a background notification proxy function. The example did not even save the state of the GPIO pin controlling the LED. Instead, the pin value was toggled directly by a GPIO background service request and the state of the pin value was held directly in hardware.

To start the introduction to ReSEE, we expand that example to include controlling the blinking by a push button. This example is also not complicated, but contains enough functionality to begin an informal exploration of the ReSEE execution model. Broadly, the problem is to use a push button to start an LED to blink. The LED continues to blink until the button is pressed again. Thus, rather than being time controlled, the LED blinking is controlled by detecting a button push in the environment.

In the following two sections, we expand upon the example to present the core ideas of the execution model that ReSEE supports. By the end of Part-II, a complete implementation is given.

As always, we must state precisely what is to be done and give the following specific requirements to meet.

1. Buttons are connected directly to an input pin. Given the nature of the mechanical spring construction of the button, pressing and releasing the button may cause spurious signals to be generated and potentially detected. There must be a “debouncing” mechanism for the button signal to enable reliable detection of button presses.
2. An external electrical circuit determines the value of the pin signal when a button is pressed and we want to accommodate both choices of the pin’s active value.
3. The debouncing mechanism operates by ignoring any signal changes during some time interval after initial detection and insisting the input pin still be at its active value after that interval. The debouncing interval also determines how long a button press must be held by a user to be recognized.
4. After detecting a button press, we inhibit detecting another press until some time duration has elapsed. This has the effect of setting the maximum frequency of button presses we are willing to take.
5. When the button is pressed, the LED is made to blink and continues blinking on and off until a subsequent button press.
6. The on and off durations of the LED, *i.e.* the shape of the implied square wave generated on the LED output pin, must be configurable.
7. Like the button, external circuitry determines the value to which the output pin is set to turn the LED on or off. We support both configurations.

Analysis of these requirements shows that the intended behavior of the button can be parameterized by the following values.

1. The pin number for the GPIO associated with the button.
2. The pin value of the input point has when it is active.
3. The confirmation interval time. This is the duration that the pin’s value must remain active in order for the button push to be recognized.

4. The relaxation interval time. This is the duration after recognizing a button push during which no addition button push is accepted.

A graphical representation of the parameters along with some names for them is usually easier to comprehend at a glance. We can represent those parameters using a UML class diagram graphic notation. Again, we emphasize that our usage of UML is limited to representing using the graphical symbols without the accompanying object-oriented semantics.

Bouncy Button	
Pin	: Pin_number {I}
Active value	: Pin_value
Confirm interval	: ms
Relax interval	: ms

Figure 23. Bouncy Button

A similar requirements analysis shows that the behavior of the LED can be parameterized as follows.

1. The pin number for the GPIO associated with the LED.
2. The GPIO pin value needed to turn on the LED.
3. The GPIO pin value needed to turn off the LED.
4. The time interval during one cycle that the LED is to be on.
5. The time interval during one cycle that the LED is to be off.

For the LED, the graphical representation is:

Blinky LED

Pin	:	Pin_number	{ I }
Active value	:	Pin_value	
Inactive value	:	Pin_value	
On interval	:	ms	
Off interval	:	ms	

Figure 24. Blinky LED

The parameterization of the button and LED, as shown in the graphical representation, represent abstractions of an actual push button and LED. The choice of attributes, and consequently the fidelity of the abstraction to the real world, is guided by the way in which the push button and LED were conceived to operate together. For example, there is no attribute for the color of the push button or the color of the LED. Such data exists, but is not useful for the purposes of this particular program. There is always a context which pertains to abstracting the real world. The subject matter of the domain under consideration provides that context. We are not interested in every conceivable push button or LED. Rather, we are only interested in push buttons and LED's as they pertain to meeting the requirements of the system being built. The abstractions of push buttons and LED's is one particular classification of real world push buttons and LED's and we call such abstractions a *class*.

There's that *class* word again

The term, *class*, is used in so many contexts in software that it is in danger of loosing any precise meaning. Object-oriented programming languages support notions of class as a means of associating data and computation together. The use of *class* here is precise and different than its use in object-oriented software contexts. *Class* is such a useful word that we risk the dissonance that might occur for readers who will undoubtedly bring their own mental model to the term, *class*, based on past experiences. The usage of *class* in this book is based on the relational model of data rather than object-oriented programming concepts. It is an interesting subject to discuss and compare the differences, but those discussions are postponed. At this time, it is sufficient to say that the two approaches share many similarities but have significant logical differences.

The class definition is a logical predicate and the attributes form the variables of that predicate. We may say:

A Bouncy Button is identified by the pin to which it is attached and characterized by the value of the pin when the button is pressed, the time interval the button must remain pressed to be

considered valid, and the time interval the button is ignored after a successful press.

Similarly, the predicate associated with a Blinky LED may be stated as:

A Blinky LED is identified by the pin to which it is attached and characterized by the values used to turn the LED on or off, the time interval the LED is to be on, and the time interval the LED is to be off.

When a class predicate is supplied values for its attributes, the resulting class instance is a logical proposition that is considered **true**. The set of all classes and their instances determine the full extent of the information about the domain under consideration. Truth here is really self-consistent truth and any proposition not stated as a class instance is considered false. This is known as the [Closed World Assumption](#).

Note that the `Pin` attribute of both Bouncy Button and Blinky LED is marked as an identifier, `{I}`. In this usage of the concept of class, the instances of a class are constrained to form a mathematical set. Sets have no duplicated elements, so there must be some means of identifying each instance. An identifier is defined as a non-empty set of class attributes which, taken together, must be unique for each instance of the class.

A class may also have multiple identifiers. In the case of multiple identifiers, the attributes of the identifiers may not be subsets (proper or improper) of each other. However, it is not unusual for identifiers to have a non-empty intersection of their attributes, i.e. the identifiers share some common attributes.

Deciding how to identify class instances requires consideration of where the responsibility is placed to ensure that the values of the identifying attributes are guaranteed to be unique. A class instance may be identified by so-called *natural* attribute values or, in many cases the identification is arbitrary, and *system supplied* identifiers are a better choice.

Consider the example of a collection of children's marbles in a bag. Let's say all the marbles are blue cat's eye marbles. They all look the same, yet there are many such marbles in the bag. By the physical laws of rigid bodies, we could identify each marble by the coordinates of its center of gravity relative in some suitable coordinate system. Unfortunately, each time the marble bag is moved or jostled, the identifiers of each marble might change and need to be updated. This is not a convenient identifier for marbles in a bag. Rather, we would be much better off scratching a unique number on each marble or attaching some distinguishing label to the marbles. In that manner, the process of identifying the marbles can be made once and the identifiers do not change. This is an example of a system contrived identifier and similar contrivances are often used.

For the case of the Bouncy Button, the GPIO pin number serves as a natural identifier. We need only verify that the hardware numbering gives a unique number for each pin and assert that no two buttons may use the same hardware GPIO. Both these conditions are easily verified and do not change with a given system.

The problems of using *natural identifiers* are legendary, especially in attempts to identify people. Names, birthdays, Social Security numbers and many other characteristics of people are not unique and place control over the uniqueness requirement outside of the software development process. In such cases, systems usually make up an identification scheme that can be ensured to be unique and then relate characteristics like name and birthday to the generated identifier. Such is how we all have far too many *account numbers* to keep track of. It is often the case that developers fail to realize both when uniqueness is required and when they have achieved something that is unique^[6].

It is important to understand that class identifiers are a *formalism* of the modeling approach. The fact that class instances form a set with an identification scheme has many uses which are explored later. The implementation of identification schemes is not necessarily directly in keeping with what

the model formalism itself might suggest. There are several implementation approaches for an identification scheme which we explore later.

A Bouncy Button has a real world association with a Blinky LED and vice versa. The Bouncy Button *controls the blinking of* a Blinky LED and a Blinky LED *has its blinking controlled by* a Bouncy Button. The rules in this world are that one button controls one LED and one LED is controlled by one button, *i.e.* this is a one to one association between push buttons and LED's. The rule is certainly not profound nor does it give deep insight into the system. Most such rules are simple and our goal is to ensure the rules are cataloged, clearly stated, and unambiguously applied. A consequence of the one-to-one nature of this rule is there are no push buttons which are not controlling an LED and no LED that is not controlled by a push button. The logical statements implied by the model can be used to verify that the modeled world captures that portion of the real world in which the domain operates.

The relationship, which by convention we choose to call R1, is between the Bouncy Button and Blinky LED sets. In this case, the relationship is a mathematical function between the two sets and, because it is defined as one-to-one, the relationship is a [bijective function](#). Graphically, we can represent the association as follows (again using a minimal set of UML graphical notation).

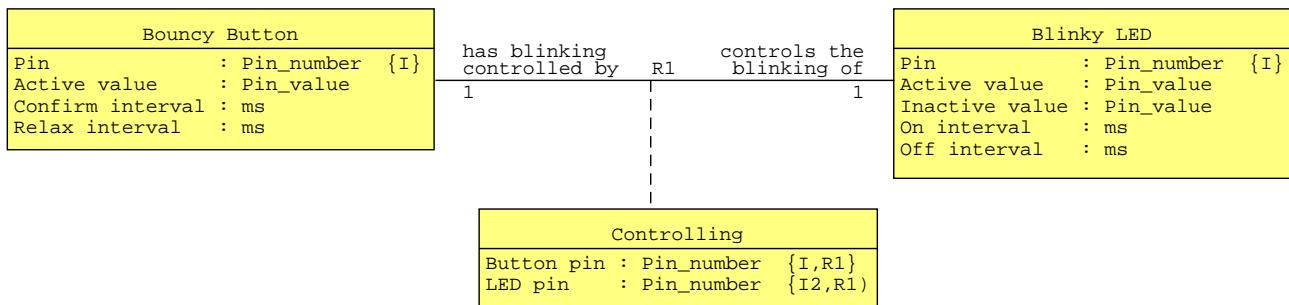


Figure 25. Relationship for LED Blinking Control

Consider the **Controlling** class from the previous diagram. It has two attributes, **Button pin** and **Led pin**. The attributes are *referential* as well as *identifying*. The instances of **Controlling** form a map between the set of instances of **Bouncy Button** and **Blinky LED**. The **Button pin** attribute has the same *value* as the **Pin** attribute of some instance of **Bouncy Button** and is marked with an R1 tag. Similarly, the **Led pin** attribute has the same *value* as the **Pin** attribute of some instance of **Blinky LED** and is also marked with the R1 tag. It is important to realize that the mapping between the two sets is *value based* and the mapping implied by the association is realized by *attribute equality*.

Since the association is one-to-one, there are as many instances of **Controlling** as there are instances of either **Bouncy Button** or **Blinky LED**. Also, the number of instances of **Bouncy Button** is the same as the number of instances of **Blinky LED** (by virtue of the one-to-one property of the association). Since the **Pin** attribute of either **Bouncy Button** or **Blinky LED** is an identifier, then instances of **Controlling** have two identifiers and may be identified by either the **Button pin** attribute value or the **Led pin** attribute value. Since we intend to have a computer-based solution, the number of class instances is always finite and so the R1 function can be specified by simply enumerating the mapping between the instances of the two classes by supplying values for the **Button_pin** and **LED_pin** attributes. The instances of **Controlling** are the [Cartesian product](#) of the set of identifying attributes of the **Bouncy Button** and **Blinky LED** classes.

In the case of an association which is also a function, the association class (*i.e.* the class that forms the correlation between the relationship participants, **Controlling** in this example) can be eliminated by incorporating the referential attributes into one of the participating classes. For a one-to-one association, you are free to choose either class to hold the referential attributes. Usually, we

choose to place the referential attributes in the class associated with the subject of the verb phrase which represents the function. The previous class diagram may then be simplified to be as follows.

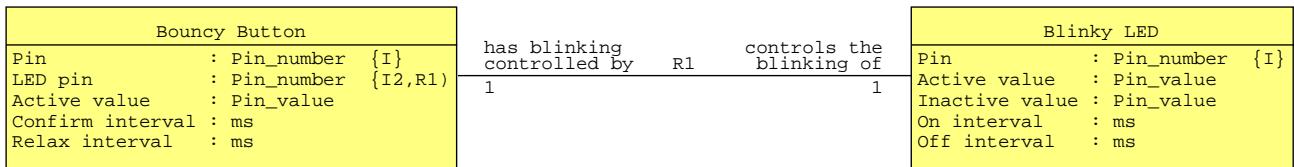


Figure 26. Simplified Relationship for LED Blinking Control

The simplified class diagram contains the same information as the one with the **Controlling** class. For example, given an instance of **Bouncy Button** you can find the related instance of **Blinky LED** by applying the function implied by R1. Applying R1 to an instance of **Bouncy Button** implies that we search the set of **Blinky LED** instances to find one where the value of the Pin attribute equals the value of the Led pin attribute of the instance. Conversely, given an instance of **Blinky LED** applying R1 means that we must find the instance of **Bouncy Button** whose Led_pin attribute value is the same as the attribute value of Pin of the **Blinky LED** instance. Note the data model does *not* specify how such searches are conducted. That information is in the realm of the implementation which varies depending upon the size and persistent storage specifics of the underlying execution mechanism.

In mathematics, functions are usually described by their properties with respect to fixed sets. For example, $f(x) = x^2$, can be defined as a function whose domain is the real numbers (\mathbb{R}) and whose codomain is also the real numbers. We can then examine the properties of the function, usually by examining its image in the codomain.

The usage here of associations as binary relations on sets takes a different point of view. The properties of the relationship, e.g. the singular and unconditional nature of a one-to-one association, is used to *constrain* the membership of the class instance sets. Class instances can possibly be created and deleted over time, but when the data is examined, the constraints implied by the class relationships *must* hold true. This allows us to make precise statements of constraints on the membership of the class instances sets. By using class names as nouns and associating verb phrases with the association, it is possible to create a precise, if somewhat stilted, series of statements about the program data that must hold true throughout the execution of the program.

Not all associations are total functions. Several variations of the *conditionality* and *multiplicity* of associations are *partial functions*, i.e. not all elements of the domain set have a corresponding mapping to the codomain set. Associations that are partial functions require the use of an association class to contain the mapping because, in general, there are a different number of instances of the association class than of the referring set.

There is one other type of relationship between classes called a *generalization*. Whereas an association is based on the Cartesian product of sets, the generalization is based on the *disjoint union* of sets. These two types of relationships are complementary and together form the basis of encoding the logic of real world associations. The generalization relationship is discussed in a later example.

Like class identifiers, relationships are a *formalism* of the modeling. Relationships between class instance sets provide a way of *executing* the functions implied by the relationship in a consistent manner with minimal rules. How the formalisms of relationships are implemented is another matter altogether. Again, we must emphasize the separation between the logic of the model and its implementation by a computing machine. For now, it is sufficient to understand that classes are logical predicates which consist of sets of class instances that are logical propositions and that classes participate in relationships which encode the semantic rules of real-world associations

between class instances.

There is much more to be said about classes and their attributes and the relationships between class instances. There are many ways to implement the formalism of our data management logic and the later chapters show the precise design and implementation of the mechanisms used to support this view of data for the reactive systems we target.

Sequencing Execution

The view of data described in the previous section can be seen as a “static” view of the system. That view is informative. It is possible to make precise statements about the domain predicates and their associations. As an information model it can be queried. But there is more to a reactive system domain than just data and no computation is described by such an information model. Some classes in a domain are passive, simply holding data for use elsewhere. For example, a class that holds design and operating characteristics of a pump may not have any computation associated with it. It serves simply as a set of values that can be used by other classes when controlling a particular pump according to its design characteristics.

In this section, we continue the button / LED example to illustrate how execution is sequenced over time. The primary mechanism for sequencing execution is by the actions of an extended Moore state model which models the life cycle of class instances. Not all classes have an interesting life cycle. Some classes come into existence when the program starts, go out of existence at program termination and otherwise have no behavior that varies in time or by preceding events. Classes whose action varies over time use a state model to encode the behavior. All the instances of a given class have the same behavior. But each instance operates independently of the others and has its own notion of its current state.

There are precise rules for how the state models operate, but we defer that discussion until later. For now, we give an informal overview of how state models sequence execution.

Consider the **Bouncy Button** state model below.

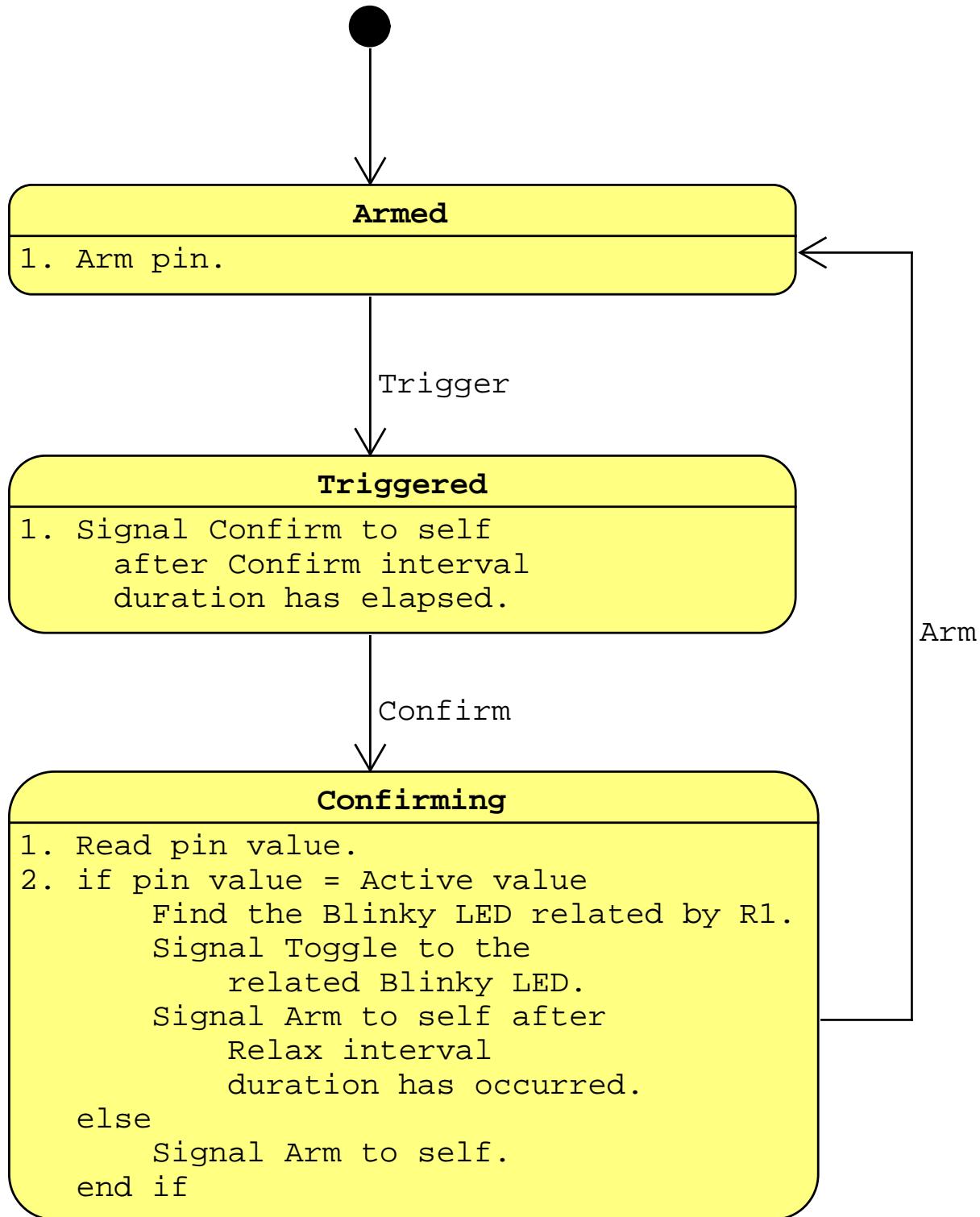


Figure 27. Bouncy Button State Model

In this diagram, states are represented as yellow boxes and events are represented by directed arrows connecting the states. The model consists of three states and the initial state is **Armed**. When any event is received in a current state of the machine, it causes a transition to a connected

state. When the new state is entered, the action listed in the state is executed. Here we have used a form of *structured English* to describe the actions. Actions are associated with states and are only executed upon the transition into a state (even when the transition is back to the same state). In this discussion, we omit any explanation of how events are generated and delivered and several other details of the transition mechanism. These are explained later when we take up execution sequencing in earnest.

The state model is cyclic and proceeds through its actions one step at a time. The *Signal* operations in the state actions are noteworthy. State models may receive events as the result of some happening in the system environment or as events signaled from other state models or even signaled to themselves. You may also specify a delay time for a signal and it is not signaled until at least the delay time has elapsed. Note that event signals are always directed at a specific instance of a class and *not* to the class as a whole. That instance is known either by context, *i.e.* the instance that is executing the action may be the target of an event signal, an instance of another class, or a happening in the environment that is bridged into the system. The most common way in which one class instance signals another class instance is to locate the target instance by traversing one or more relationships that exist between the signaling instance and target instance classes. The relationships have semantic significance and traversal is the computation of the binary relation that the relationships represents. In this example, the **Confirming** state action traverses **R1** to find a related instance of **Blinky LED**.

The following figure shows the state model for the **Blinky LED** class.

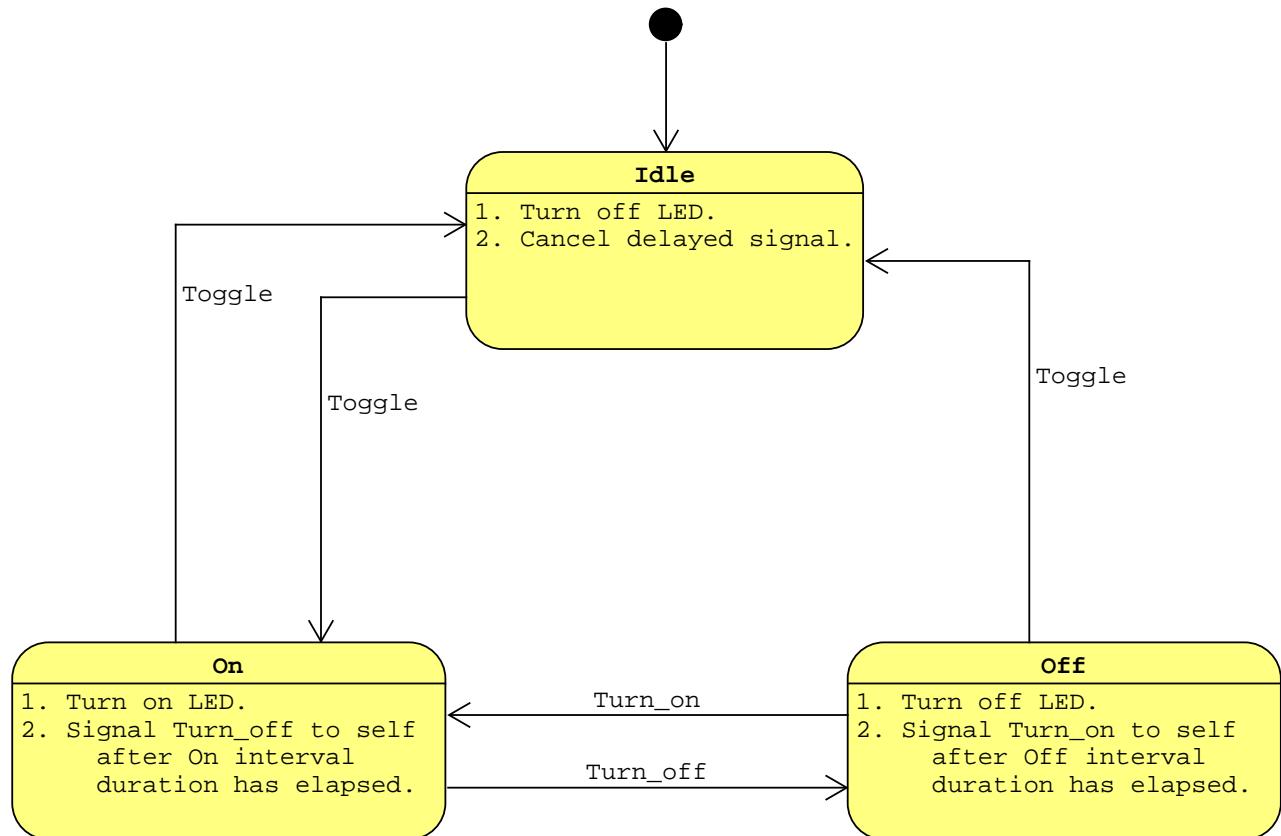


Figure 28. Blinky LED State Model

The two state models precisely define the sequence of computations for the **Bouncy Button** and **Blinky LED** class instances. Yet, it is not immediately apparent how these models interact to achieve the behavior we want. A sequence diagram is a good graphic to show how, over time,

control flows between the class instances.

The diagram below shows one particular sequence to illustrate how the **Bouncy Button** interacts with an input pin and a timer to generate the **Toggle** event to the **Blinky LED**.

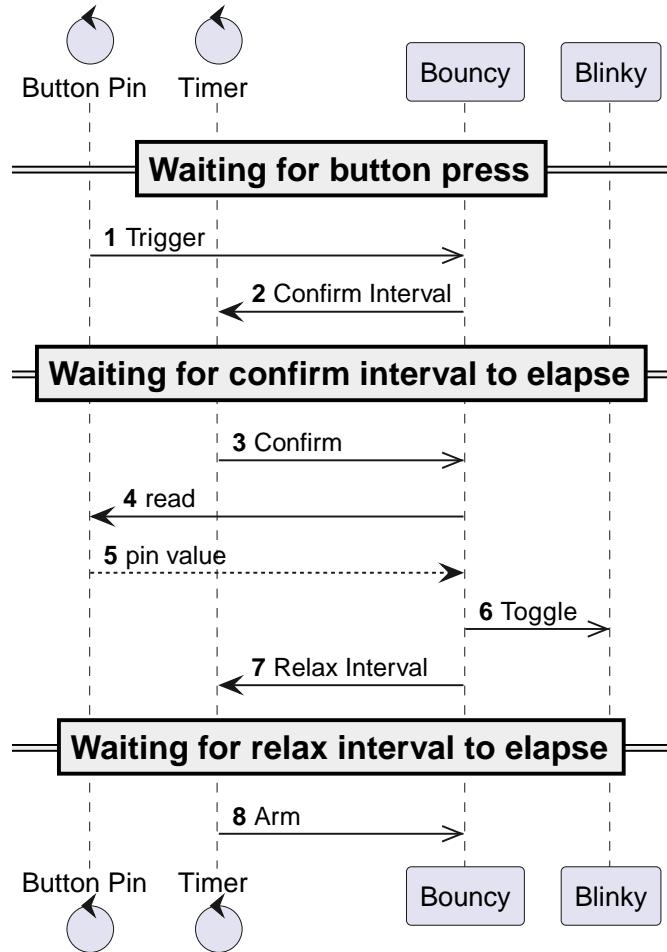


Figure 29. Sequence Diagram of Push Button

The following describes the interactions with the environment in terms already presented in Part I.

- When the program is started, the GPIO pin associated with the button was initialized to respond and the Background Notification Queue is used to signal the **Trigger** event to the **Bouncy Button** state machine. This event causes the transition into the **Triggered** state.
- The **Triggered** state action responds by using the timer queue to request notification when the **Confirm** interval time has elapsed.
- When the confirmation interval time has elapsed, the Background Notification Queue is used again to transport the notification into the background where it is used to signal the **Confirm** event to the **Bouncy Button** state machine.
- The **Confirm** event causes a transition into the **Confirming** state and the state action reads the value of the pin associated with the push button.
- The returned pin value is checked and, for this case, it is assumed to confirm the button press (*i.e.* this was not a glitch).
- The positive confirmation of the button press causes the logic of the **Confirming** action to find the related instance of **Blinky LED** and to signal the **Toggle** event to the state machine of that instance. The **Toggle** event is sent to the specific instance of the **Blinky LED** state machine.

that is associated with the specific instance of **Bouncy Button** that is executing as given by the R1 association.

7. The state action continues, using the timing queue to indicate when the relaxation interval elapses.
8. After the relaxation interval elapses, the timer queue uses the Background Notification Queue to indicate the elapsed relaxation time and this results in signaling the **Arm** event to the **Bouncy Button** state machine.

Note that all the actions originate from either a GPIO interrupt or a timer interrupt. The time intervals where the state machine is “waiting” the system may perform other computation including going into low power mode if there is no additional work to be done.

The connection between the button press and controlling the lighting of the LED happens via the **Toggle** event sent by the **Bouncy Button** state machine to the **Blinky LED** state machine. When the **Toggle** event is received by the **Blinky LED** state machine, the following sequence diagram shows what happens.

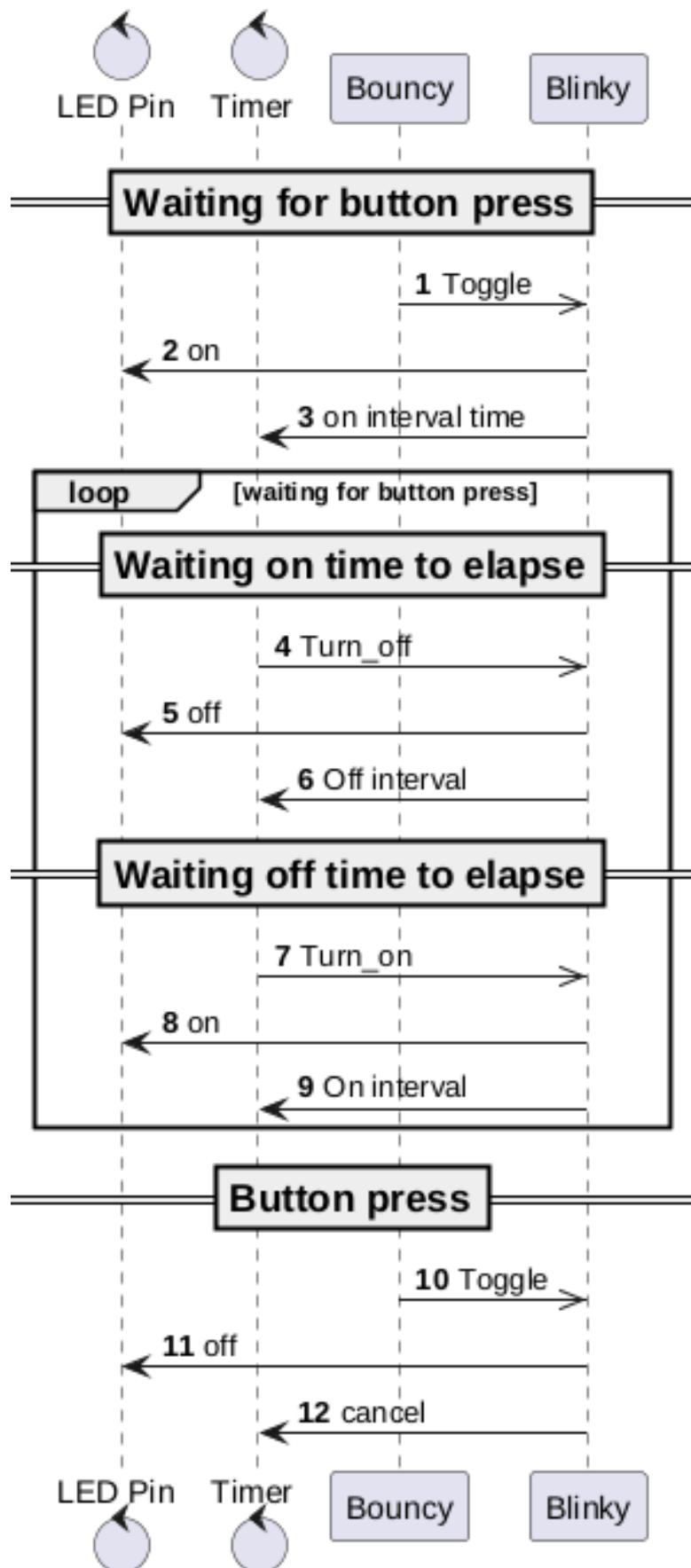


Figure 30. Sequence Diagram of Blinking LED

1. When the program is started, the GPIO pin to which the LED is connected is initialized so that the LED is off. The receipt of the **Toggle** event from the **Bouncy Button** state machine causes the transition to the **On** state whose action starts the blink sequence.
2. The LED pin is set to the active state to turn on the LED.
3. The state machine uses the timer queue to request notification when the amount of time the LED is to remain on has elapsed.
4. When the on time has elapsed, the notification is used to signal the **Turn_off** event to the **Blinky LED** machine. Receipt of the **Turn_off** event causes a transition to the **Off** state.
5. The activity executed upon entry into **Off** sets the LED pin to a value that turns the LED off.
6. The timer queue is used to notify the state machine when the amount of time that the LED is to remain off has elapsed.
7. When the off time has elapsed, the notification is used to signal the **Turn_on** event to the **Blinky LED** machine.
8. The **Turn_on** event causes a transition to the **On** state and the state action sets the LED pin to on.
9. Again, the timer queue is used to measure the amount of time the LED is to remain on. Steps 4 through 9 are repeated by virtue of the state transitions until the push button is pressed.
10. When the push button is pressed, assuming that a valid press is detected, the **Bouncy Button** machine again signals **Toggle** to the **Blinky LED** machine.
11. The LED is turned off, regardless of its current state.
12. Any potentially pending time out is cancelled. The **Blinky LED** machine is now idle and waits until it is signaled with another **Toggle** event.

There are more details of state model execution that have been omitted in this introduction and we cover those details in a separate section.

Taking a step back from the diagrams and descriptions, the computations can be categorized as:

- Synchronize with the system environment. Data and control flow across the system boundary and, for some domains, interact directly with the execution.
- Sequence execution of the state machine by signaling events and having those events dispatched to cause state machine transition.
- Access parametric data associated with the domain classes. This includes both reading attributes from the data model as well as updating attributes with calculated values.
- Create side effects in other domains. Similar to interacting with the system environment, a domain may interact with other domains. This is the essence of the *bridging* mechanism that is used to build the complete system from subject-matter decomposed domains.

The remainder of Part II is divided between the part of ReSEE that manages data and that which manages execution. We start with data management because without a robust foundation for containing the data, the execution portion has no information with which to calculate.

It is important to emphasize that our intent here is to build a software platform-dependent implementation of the execution model. The ReSEE embodies a set of implementation choices for the mechanisms which map the concepts of the execution model onto systems of the type we target. In keeping with our approach both in this part of the book and Part I, the engineering choices that are appropriate to the demands of the reactive applications we target are not necessarily universally good choices for other types of applications which necessarily have different computational demands on their target platforms. There are several places, which we point out later, where a particular implementation choice will force a particular construction be used for the

mapping from the execution model.

Socratic Questions

1. Isn't ReSEE just another kind of RTOS?

No. A primary concept of a typical RTOS is to maintain multiple, virtual processor contexts and provide a scheduler to decide which virtual processor context is run in the physical processor at any given time. Schedulers typically provide a variety of scheduling policies such as time based or priority based scheduling. In ReSEE, there is only one processor context and that is the one running in the single physical processor core.

2. So, there is no preemption of background processing by other background processing?

Correct. Background processing runs with interrupts enabled to allow the foreground processing to handle interactions with the system environment. But there is no preemption of background processing by other background processing. Arbitrary preemption of background processing adds non-deterministic behavior. It is a "run to completion" style of execution.

3. How do you characterize the execution sequencing scheme?

The ReSEE execution sequencing is a single stack coroutine scheme that is an implementation of the execution model and is better suited to the implementation targets. The twist in the coroutine scheme is that *yielding* is not directly invoked by the application code. Each state action executes as a coroutine and runs to completion. The end of the action results in a yield action and the next state transition is dispatched. For our targets, a single stack implementation scheme offers a number of advantages. There is only a single processor context and that is the one currently in the machine registers. Stack sizes are more predictable and it is not necessary to use additional memory for multiple virtual contexts and the stacks those contexts require. This minimizes the additional software required just to decide what is to be computed next.

4. How do you characterize the data management scheme?

One shortcoming of conventional RTOS approaches is that they are focused almost completely on scheduling the many virtual contexts onto the single processor. Secondarily, they provide facilities for inter-task communications and interfacing to the environment via interrupt handlers. With respect to data, you are on your own. In ReSEE, data and execution operate as complementary parts of the system solution.

5. Aren't state machines subject to "combinatorial explosions"?

Using a state model for an entire system or some large component of a system can result in a combinatorially large transition function since the number of events and states is trying to cover a large amount of logic. This execution model avoids large numbers of states and events by associating the state model to the life cycle of a class. Each class instance has its own state and when we talk about the state behavior of class instances we use the term, *state machine*. Experience shows that life cycle-based state models are usually quite small because behavior of a single class is well contained. We prefer small state models that derive their power by composing with other state models by signaling events between themselves. Large state models in this context are often a sign of poor decomposition in the analysis and that there may be multiple classes hiding in the model.

6. How about using hierarchical state models?

Hierarchical state models were conceived to reduce the combinatorial problems of applying state behavior to larger portions of a system. They are considerably more complex and have more rules of execution. This execution model does not use them. This approach avoids hierarchical formulations in favor of directed graph oriented approaches. The composition of one state machine signaling events to another state machine whose identity is determined by traversing relationships between the classes involved is essentially a directed graph view.

[1] arm-none-eabi-gcc (GNU Arm Embedded Toolchain 10.3-2021.07) 10.3.1 20210621 (release)

[2] The term, *compiler*, is used here in a broad sense to include the possibility of interpreters and other implementation

mechanisms which achieve the same goal of mapping language semantics to machine execution.

[3] <https://www.cs.utexas.edu/users/EWD/ewd04xx/EWD498.PDF>

[4] Although, there surely exists some obscure language where this is not true.

[5] For now, non-functional system requirements are not considered.

[6] Private communications from Paul Higham.

Data Management Fundamentals

The previous chapter shows an informal introduction to the ReSEE execution model that we spend the remainder of this book to explain further and implement. The examples are hopelessly simplistic for practical systems but provide the foundational concepts that we continue to explore. In this chapter, we show the implementation of the fundamental mechanisms used for managing data in ReSEE. In a future chapter, we show how these fundamental constructs are used in ReSEE to support our execution model.

There is no requirement for the data management mechanisms to be implemented strictly in relational algebraic terms. We stated in the core modeling concepts that the data management facet of a domain model is based upon the relational theory of data, but that does *not* imply that the only path forward for an implementation is to be based on relational algebraic operations. It is sufficient for the implementation to achieve the requirements allocated to it by whatever mechanisms are well suited to the class of target applications.

However, being able to base ReSEE data management directly on relational algebra reduces the cognitive burden on understanding how data is actually handled. The implementation difficulty arises from the fact that relational algebraic operations are *type generators*. Although the algebra is closed in the sense that operations take relational operands and produce relational results, the headings associated with the results may be different than those of the operands. Projection of attributes is an easy example. The project operation on a relation value results in another relation value that contains a subset of the attributes of the operand of project. For a statically typed implementation, this implies that we must generate both the data structures and type-specific code for these operations.

The type generation characteristics of relational algebra are challenging for many programming languages to meet. In the case of "C", one is reduced to extensive use of pre-processor macros, using an external code generator of some sort, or bearing the costs of manually coding and maintaining significant quantities of type-specific code. Although one could consider using a run-time or hybrid type system that could accommodate newly generated types, such an implementation is deemed too expensive for our application targets. This design yields a "pay for what you use" property rather than having to amortize a full featured run-time type system over the entire system.

This deficiency in "C" is one of the prime reasons that the implementation was done in Zig. Zig has the ability to generate new types at compile time and, in this execution model, all type information needed to produce new relational values is also known at compile time. As a counter example, a RDMS must have a run-time type system because new relation types can be created upon command. In this execution model, no new relational types are generated at run-time and all queries on the data are known at compile time. Because Zig has this capability, the design of the data management implementation chooses to build directly upon a relational algebra library.

The following steps describe how the relational algebra foundation is built and used to produce the application data management functions of ReSEE.

1. The next three sections present an unembellished relational algebra consisting of relation tuples, values and variables. Application data is stored in relation variables. Calculations on application data are performed on relation values.
2. Once the relational algebra foundations are in place, there are three sections discussing the principles and implementation of binary relations between relation variables. Binary relations give a concise way to specify functional relationships between the tuples in a relation variable. We show how those relationships between relation variables are implemented using an adjacency matrix to track the relations binary relation instances between relation variable tuples.

Further chapters then show how the foundations of relational data and binary relations are applied to yield all the necessary mechanism for storing and manipulating application data in accordance with our execution model.

The implementation is in the Zig language and we begin to see substantial amounts of compile-time meta-programming in the Zig code that provides the type generation needed by relational algebraic operations.

Relation Tuples

A Relation Tuple^[1] represents a single logical proposition about some aspect of a subject matter domain. They are the basic building blocks for encoding logic into data. A tuple consists of a heading and a corresponding set of values. The heading consists of a set of attributes names / attribute type pairs and represents a logical predicate with the attributes as the variables of the predicate. The attribute values are the predicate variable values and the tuple, with its attribute values, is considered to be a true proposition.

The attribute names form a set, implying that there are no duplicate attributes. At all times, every attribute of a tuple must have a valid value from the set of values of the attribute's data type. A tuple is immutable in the sense that no functions are provided to modify directly the attribute values in place. The attributes of a tuple heading have no inherent *order*. Tuples are type generators and new tuple headings are created by many of the tuple algebraic operations, e.g. project.

Tuples can hold three categories of values:

Scalar values

Scalar values are ordinary system-supplied values such as integers or floating point numbers. Scalar values may also have internal structure, which can be accessed according to the operations provided by the type of the value. This allows for user defined data types based on Zig structures. On this basis, we disallow certain Zig types as being suitable as tuple attributes. Many Zig-provided types are concerned with the operation of the language itself. We do not consider many native Zig data types as providing information of value to application logic. Essentially, scalar values are any values that are not either tuple values or relation values. Fundamentally, all scalars must support an equality operation^[2].

Tuple values

A tuple attribute may itself be a tuple type and hold a tuple value. We describe such attributes as *tuple-valued*. Tuple values provide an equality operation.

Relation values

A tuple attribute may hold a relation value. We describe such attributes as *relation-valued*. We define relation values in the [following section](#). This can be considered a form of nesting. It is rare to use deep, nested hierarchies as tuple values. With more than one or two levels as it becomes difficult to interpret the logical intent and is usually an indication a poor domain analysis. Relation values also provide an equality operation.

Representing a Tuple Value

We represent a tuple in Zig as a `struct` onto which additional information and functions are added at compile time. In the code, we refer to the basic Zig `struct` as a **Heading**. Tuples are generated based on a **Heading** which is verified for allowed data types. After verification, the **Heading** is treated as a **TupleBase**. This same usage will also show up when we discuss Relation Value types. Tuples do not allocate memory and are intended for use as simple Zig values.

```

<<relation tuple>>
/// `RelTuple` is a type function that creates a new Relation Tuple data type from
/// a basic Zig `struct` which it treats as the heading of a Tuple.
pub fn RelTuple(
    comptime Heading: type,
) type {
    validateTupleHeading(Heading);

    return struct {
        const Self = @This();

        <<tuple fields>>
        <<tuple declarations>>
        <<tuple public functions>>
        <<tuple private functions>>
    };
}

```

At compile time, the tuple heading is checked to ensure it does not have any inappropriate attribute data types for a relational tuple. There are many types in Zig which are used in language operations and which don't hold information useful for logical predicates, e.g. `noreturn`. Note also that optionals are not allowed. Zig optionals would correspond to a `NULL` value in a relational context. Much has been written about `NULL` in relational contexts and we do not repeat the arguments here. Rather, we simply state the rule that `NULL` values are not allowed because they have the following undesirable characteristics:

- Null values have do not have clear semantics. They are used to indicate optional information, missing information or the result of a partial function.
- Null values introduce [three-valued logic](#) which is an unwelcome complication and for which it is difficult to devise a scheme that handles all the edge cases well.
- Null values are not needed. This treatment of tuple (and relational) algebra contains all the necessary constructs to handle the case of partial functions. The other two semantic issues with Null values are considered design deficiencies.

```

<<tuple module private functions>>
/// The `validateTupleHeading` function examines `Heading` to determine
/// if it is suitably typed to be used as the base for a `RelTuple`.
/// A compiler error is generated for any structure fields that are not
/// allowed in a `RelTuple`.
fn validateTupleHeading(
    comptime Heading: type,
) void {
    if (@typeInfo(Heading) != .@"struct")
        @compileError("a tuple heading must be a struct type");

    inline for (meta.fields(Heading)) |heading_field| {
        // The only pointer and array types allowed are those which can be Zig strings.
        if (!trait.isZigString(heading_field.type)) {

```

```

const field_info = @typeInfo(heading_field.type);
switch (field_info) {
    // These types are deemed sufficient to encode application logic.
    // Note we allow `struct` types as a mechanism to have both
    // relation values and user-defined abstract data types
    // as tuple attributes.
    .void,
    .bool,
    .int,
    .float,
    .@"struct",
    .@"enum",
    .vector,
    => {},
}

// All other Zig types are not allowed as types for tuple attributes.
else => @compileError("field, '" ++ heading_field.name ++
    "', has type, '" ++ @tagName(field_info) ++
    "', which is not allowed in a RelTuple"),
}
}
}
}

```

The distinction between Zig types allowed as tuple types is complicated by the notion of a string in Zig. Zig has no specific string data type but treats strings as arrays of u8 values. Normally, arrays are not considered an appropriate data types for tuple attributes. We allow a special case for strings and consider any type which is allowed by the function, `isZigString` as acceptable. The `isZigString` function^[3] encapsulates the Zig language rules for the set of arrays and pointers that can be coerced into a slice of u8 values, which is the usual way that string data is handled.

We make a number of declarations for symbolic convenience.

```

<<tuple declarations>>=
/// The `TupleBase` is an identifier for the base Zig `struct` type used
/// to define the tuple.
pub const TupleBase = Heading;

```

The primary field for a `RelTuple` simply holds a `TupleBase` value.

```

<<tuple fields>>=
/// A holder for the base `struct` value.
reltuple: TupleBase,

```

It is convenient to have a `void` field that serves as a tag or marker that this `struct` is indeed a `RelTuple`.

```

<<tuple fields>>=
/// This field is used as a tag to ensure arguments passed in type-erased
/// contexts are actually `RelTuple` tuples.
comptime is_reltuple: void = {},

```

The test for whether a type is a tuple type simply checks for the presence of the `is_reltuple` field.

```
<<tuple module public functions>>=
/// The `isRelTupleType` returns `true` if the `TupleType` argument is
/// a type as returned from the `RelTuple` type function and `false` otherwise.
pub fn isRelTupleType(
    comptime TupleType: type,
) bool {
    return @typeInfo(TupleType) == .@"struct" and @hasField(TupleType, "is_reltuple");
}
```

So, a `RelTuple` is an embellished Zig struct with limitations on the data types allowed in the struct.

Attribute Identifiers

Although attributes of a tuple are given a string name, it is more convenient to refer to attributes of existing tuples using an enumeration. We make extensive use of the attribute enumeration and give the enumerated type for the attributes of a tuple the, `AttributeId` name. The `meta` module of the standard library has a function to create such an enumeration. Using the enumeration is tantamount to having a numerical “index” into the tuple, but note the values of the enumerators are completely determined by the compiler at compile time.

```
<<tuple declarations>>=
/// `AttributeId` is an enumerated data type where each variant has the same name
/// as a field in the `TupleBase` .
pub const AttributeId = meta.FieldEnum(TupleBase);
```

With this type definition, attributes can be referred to using an *enum literal*. For example, an attribute named, “weight”, can be indicated by the enum literal, `.weight`.

Several standard lib modules are needed for the type manipulations that follow. Since Relation Tuples are type generators, what follows contains a large amount of `comptime` meta-programming to construct new data types supporting the tuple algebra.

```
<<tuple module declarations>>=
const meta = std.meta;
const trait = @import("zigtrait.zig"); ①
const mem = std.mem;
const enums = std.enums;
```

① This module was preserved from previous releases of the Zig standard library. It does *not* appear in the current version.

Tuple properties

It is convenient to hold the *degree* of a tuple as a constant of the type.

```

<<tuple declarations>>=
/// The number of attributes in the tuple.
pub const tuple_degree = meta.fields(Heading).len;

```

There are also a number of introspection functions provided. They are simple syntactic sugar. These functions usually return values associated with tuple type but are provided with tuple value interfaces since that is often at hand.

```

<<tuple public functions>>=
/// The `degree` function returns the number of attributes in a tuple.
pub fn degree(
    _: Self,
) usize {
    return tuple_degree;
}

```

```

<<tuple public functions>>=
/// The `attributes` function returns a slice pointing to the attribute names.
pub fn attributes(
    _: Self,
) []const []const u8 {
    return meta.fieldNames(TupleBase);
}

```

```

<<tuple public functions>>=
/// The `attributeName` function returns the attribute name corresponding to `attr_id`.
pub fn attributeName(
    _: Self,
    comptime attr_id: AttributeId,
) []const u8 {
    return @tagName(attr_id);
}

```

```

<<tuple public functions>>=
/// The `AttributeType` function returns the data type corresponding to `attr_id`.
pub fn AttributeType(
    comptime attr_id: Self.AttributeId,
) type {
    return @FieldType(TupleBase, @tagName(attr_id));
}

```

Tuple creation

In conventional Zig initialization style, the provided `create` function returns a `RelTuple` value. There is no allocation as these values are intended to be value-oriented.

```
<<tuple public functions>>=
/// The `create` function returns a `RelTuple` value whose attributes
/// contain the values based on the `attr_values` argument. Tuples are
/// created by supplying values for the structure fields in `TupleBase`.
pub fn create(
    attr_values: TupleBase,
) Self {
    return .{
        .reltuple = attr_values,
    };
}
```

Tests

```
<<tuple module tests>>=
test RelTuple {
    // We start with a simple base structure.
    // Note it is not necessary to specify a separate structure as the
    // heading of the tuple.
    const TestTuple = RelTuple(struct { len: usize, weight: f32, name: []const u8 });

    const test_1 = TestTuple.create(.{
        .len = 32,
        .weight = 5.0,
        .name = "foobar",
    });

    try testing.expectEqual(@as(usize, 3), test_1.degree());
    try testing.expectEqual(@as(usize, 3), test_1.attributes().len);
    try testing.expectEqualStrings("name", test_1.attributeName(.name));
    try testing.expect(f32 == TestTuple.AttributeType(.weight));

    // Construct a tuple that contains a tuple-valued attribute.
    // For testing purposes, we use a direct construction.
    // It is more common in application code to use the `wrap` function.
    const PropertiesTuple = RelTuple(struct { len: usize, weight: f32 });
    const NestedTuple = RelTuple(struct { name: []const u8, properties: PropertiesTuple });

    const test_2 = NestedTuple.create(.{
        .name = "foobar",
        .properties = PropertiesTuple.create(.{
            .len = 32,
            .weight = 5.0,
        })
    });
}
```

```

        }),

    });

    try testing.expectEqual(@as(usize, 2), test_2.degree());
    try testing.expectEqualStrings("properties", test_2.attributeName(.properties));
    try testing.expect(PropertiesTuple == NestedTuple.AttributeType(.properties));

    // A tuple may have an empty set of attributes.
    // This is an important boundary case which shows up when discussing
    // relation values.
    // The empty tuple. A tuple with no attributes and a void value.
    const EmptyTuple = RelTuple(struct {});
    const empty = EmptyTuple.create(.{});
    try testing.expectEqual(@as(usize, 0), empty.degree());
    const empty_attributes = empty.attributes();
    try testing.expectEqual(@as(usize, 0), empty_attributes.len);

    // All empty tuples are equal.
    const empty_2 = EmptyTuple.create(.{});
    try testing.expect(empty.eql(empty_2));
}

```

Tuple Equality

Before showing the tuple algebra operations, we must first address tuple equality. Equality is a fundamental concept for Relation Tuples. Two tuples are equal if their headings are equal and their corresponding attribute values are equal. Two headings are equal if they are the same length, have the same attribute names, and the data types of corresponding attributes are the same. However, there is an additional complication for the implementation. In the relational model, attributes of a tuple heading have no inherent *order*. Relation Tuples are type generators and new tuple headings are created in most tuple or relation value operations. Clearly, the implementation must hold the tuple attributes in some order in memory since memory itself is ordered. But our computation of tuple heading equality must be independent of the order the attributes are stored in memory. Access to attribute values is by enum literal values named the same as the heading attribute names. It is important that the implementation keep track of the order in which things are held in memory, but `RelTuple` users must not assume some order based on the fashion that the implementation logic uses to store attributes in memory.

Tuple equality has two components: equivalent headings and corresponding equivalent values. First, we show the function that determines tuple heading equality. The implementation technique is simple, brute-force sequential search that runs at `comptime`.

```

<<tuple module public functions>>=
/// The `equivalentHeading` function tests two headings for equality.
/// It returns `true` if the headings are the same length, have the same
/// attributes, and the corresponding attributes have the same data type.
/// Otherwise, `false` is returned.
pub fn equivalentHeading(
    comptime OneBase: type,
    comptime OtherBase: type,
) bool {

```

```

const one_fields = meta.fields(OneBase);
const other_fields = meta.fields(OtherBase);
if (one_fields.len != other_fields.len) return false;

inline for (one_fields) |one_field| {
    var match: bool = false;
    inline for (other_fields) |other_field| {
        match = match or
            (comptime mem.eql(u8, one_field.name, other_field.name) or
            one_field.type == other_field.type);
    }
    if (!match) return false;
}

return true;
}

```

Once heading equality is established, it is necessary to compare values on an attribute-by-attribute basis to establish the tuples as the same. Because tuple attributes may be either tuple-valued or relation-valued, that must be taken into account when computing equality.

```

<<tuple module declarations>>=
const relvalue = @import("relvalue.zig");

```

```

<<tuple module public functions>>=
/// The `attributeEql` function compares two tuple attribute values
/// for equality. If the attributes are equal, the function returns
/// `true` and `false`, otherwise. If an attribute of the tuple is of
/// `@"struct` type and declares an `eql` function, then that function
/// is used to determine equality of the attributes.
pub fn attributeEql(
    one_value: anytype,
    other_value: anytype,
) bool {
    const OneType = @TypeOf(one_value);

    if (comptime trait.isZigString(OneType)) {
        const one_string: []const u8 = one_value;
        const other_string: []const u8 = other_value;
        return mem.eql(u8, one_string, other_string);
    } else {
        return switch (@typeInfo(OneType)) {
            .@"struct" => |_| if (meta.hasFn(OneType, "eql"))
                one_value.eql(other_value) ①
            else
                meta.eql(one_value, other_value),
            else => meta.eql(one_value, other_value), ②
        };
    }
}

```

```
}
```

① If there is an `eql` function, then use that. Both tuples and relation value define `eql` and user supplied abstract types should.

② For everything else, `meta.eql()` determines attribute equality.

We now have the pieces to put together an equality function.

```
<<tuple public functions>>=
/// The `eql` function compares two tuple values.
/// It returns `true` if the values are equal and `false` otherwise.
pub fn eql(
    self: Self,
    other: anytype,
) bool {
    const OtherTuple = @TypeOf(other);
    if (comptime !isRelTupleType(OtherTuple))
        @compileError("comparison value is not a tuple, found '" ++
            @typeName(OtherTuple) ++ "'");
    if (comptime !equivalentHeading(TupleBase, OtherTuple.TupleBase)) return false;

    const one_attributes = comptime meta.fieldNames(TupleBase);
    inline for (one_attributes) |attr_name| {
        const one_attr_value = @field(self.reltuple, attr_name);
        const other_attr_value = @field(other.reltuple, attr_name);
        if (!attributeEql(one_attr_value, other_attr_value)) return false;
    }

    return true;
}
```

Tests

```
<<tuple module tests>>=
test "tuple equal" {
    // Equivalent tuples with different order of attributes in the base declaration.
    const OneHeading = struct { len: usize, weight: f32, name: []const u8 };
    const OtherHeading = struct { weight: f32, name: []const u8, len: usize };

    const OneTuple = RelTuple(OneHeading);
    const OtherTuple = RelTuple(OtherHeading);

    // Note the order of the fields in the `struct` literal does not matter.
    // The compiler orders fields properly.
    const test_1 = OneTuple.create(.{
        .name = "foobar",
        .len = 32,
```

```

        .weight = 5.0,
    });
    const test_2 = OtherTuple.create(.{
        .weight = 5.0,
        .name = "foobar",
        .len = 32,
    });
}

// Test of order independence.
try testing.expect(test_1.eql(test_2));

// Check that tuple comparisons detect the different addresses.
const Address = RelTuple(
    struct { number: usize, street: []const u8, city: []const u8 },
);
const Person = RelTuple(struct {
    name: []const u8,
    height: f32,
    age: u8,
    address: Address,
});
}

// Check that tuple equality works. p1 and p2 are constructed as equal.
const p1 = Person.create(.{
    .name = "Tom Jones",
    .height = 5.8,
    .age = 76,
    .address = Address.create(.{
        .number = 100,
        .street = "Elm Street",
        .city = "Our Town",
    }),
});
const p2 = Person.create(.{
    .name = "Tom Jones",
    .height = 5.8,
    .age = 76,
    .address = Address.create(.{
        .number = 100,
        .street = "Elm Street",
        .city = "Our Town",
    }),
});
try testing.expect(p1.eql(p2));

// This time Tom moved to Pine Street
const p3 = Person.create(.{
    .name = "Tom Jones",
    .height = 5.8,
    .age = 76,
    .address = Address.create(.{
        .number = 100,

```

```

        .street = "Pine Street",
        .city = "Our Town",
    )),
});
try testing.expect(!p1.eql(p3));

// Unable to test relation-valued attributes here. They are tested later, in
// a later section.
}

```

Tuple Algebra

There is an well-defined yet open-ended algebra for Relation Tuples. Most of the tuple operations are used to support directly the algebra of *Relation Values*.

Except for those operations which return a scalar value, e.g. `degree`, the operations are closed and return a new tuple value. Functions for the following algebraic operations are provided:

extract	Obtain the value of an attribute.
update	Obtain a new tuple with the same heading but with one attribute value changed.
updateMany	Obtain a new tuple with the same heading but with zero or more attribute values changed.
project	Obtain a new tuple which has a subset of attributes.
rename	Change the name of attributes.
extend	Add one attribute to a tuple.
extendMany	Add zero or more attributes to a tuple.
wrap	Create a tuple-valued attribute from a subset of attributes.
unwrap	Remove a tuple-valued attribute placing the unwrapped attributes into the result.
join	Possibly obtain a new tuple that is the equi-join of two tuples. The heading of the new tuple is the union of the headings of the operands, i.e. the heading contains all the attributes of the first tuple and all the attributes of the second tuple except for any like-named attributes. If the like-named attributes are equal or the headings are disjoint, then the joined tuple is returned. Otherwise, <code>null</code> is returned.

Tuple Extract

A common operation on a tuple is to *extract* (or “get”) the value of one of its attributes. It is the only operation provided to access the value of an attribute.

<<tuple public functions>>=

```

/// The `extract` function returns the value of the attribute given by, `attr_id`.
pub fn extract(
    self: Self,
    comptime attr_id: AttributeId,
) AttributeType(attr_id) {
    return @field(self.reltuple, tagName(attr_id));
}

```

Tests

```

<<tuple module tests>>=
test "tuple extract" {
    const TestTuple = RelTuple(struct { len: usize, weight: f32, name: []const u8 });

    const test_1 = TestTuple.create(.{
        .name = "foobar",
        .len = 32,
        .weight = 5.0,
    });

    // Note that the attribute identifiers are enum literals whose name
    // matches the attribute name.
    try testing.expectEqual(@as(usize, 32), test_1.extract(.len));
    try testing.expectEqual(@as(f32, 5.0), test_1.extract(.weight));
    try testing.expectEqualStrings("foobar", test_1.extract(.name));

    // Should the tuple Heading have default values defined,
    // attribute values for those fields may be omitted when the tuple
    // is created.
    const AnotherHeading = struct {
        size: usize = 42,
        width: u16 = 14,
    };
    const AnotherTuple = RelTuple(AnotherHeading);

    const test_2 = AnotherTuple.create(.{});
    try testing.expectEqual(@as(usize, 42), test_2.extract(.size));
    try testing.expectEqual(@as(u16, 14), test_2.extract(.width));
}

```

Tuple Update

The opposite of extract is update. Note however, that tuple values are immutable and the update function returns a *new* tuple value with the specified attribute updated.

Two functions are provided. The first one updates a single value. This is the most common case and singularity makes the ergonomics of the interface simpler.

```

<<tuple public functions>>=
/// The `update` function returns a new tuple value where the attribute given by,
/// `attr_id`, has the value given by, `new_value`.
pub fn update(
    self: Self,
    comptime attr_id: AttributeId,
    new_value: AttributeType(attr_id),
) Self {
    var new_tuple = self;
    @field(new_tuple.reltuple, @tagName(attr_id)) = new_value;

    return new_tuple;
}

```

There are occasions when multiple attributes need to be updated. In this case, the interface requires that you build a list of attribute name / attribute value pairs.

```

<<tuple public functions>>=
/// The `updateMany` function returns a new tuple value where zero or more attributes
/// have updated values.
/// The `av_pairs` argument is expected to be an list of attribute name /
/// attribute value pairs.
pub fn updateMany(
    self: Self,
    av_pairs: anytype,
) Self {
    var new_tuple = self;
    inline for (av_pairs) |av_pair| {
        assert(av_pair.len == 2);
        const attr_id: AttributeId = av_pair.@"0";
        const value: AttributeType(attr_id) = av_pair.@"1";
        @field(new_tuple.reltuple, @tagName(attr_id)) = value;
    }

    return new_tuple;
}

```

Tests

```

<<tuple module tests>>=
test "tuple update" {
    const TestTuple = RelTuple(struct { len: usize, weight: f32, name: []const u8 });

    const test_1 = TestTuple.create(.{
        .name = "foobar",
        .len = 32,
        .weight = 5.0,
    });
}

```

```

});  
  

const new_name = test_1.update(.name, "baz");  

try testing.expectEqualStrings("baz", new_name.extract(.name));  
  

const new_len = test_1.update(.len, 100);  

try testing.expectEqual(@as(usize, 100), new_len.extract(.len));  
  

const new_weight = test_1.update(.weight, 17.3);  

try testing.expectEqual(@as(f32, 17.3), new_weight.extract(.weight));  
  

const several_new = test_1.updateMany(&.{  

    .{ .len, 200 },  

    .{ .name, "fup" },  

});  

try testing.expectEqual(@as(usize, 200), several_new.extract(.len));  

try testing.expectEqualStrings("fup", several_new.extract(.name));  

}

```

Tuple Projection

A Relation Tuple may be projected to obtain a new tuple that has a subset of the attributes of the original tuple. In this function, we see the beginnings of a pattern in the subsequent implementation. The previously presented tuple functions did not change the tuple heading. Projection and most of the other tuple functions return a tuple that has a different heading. The design strategy is to create a type function to produce the modified heading and then use the main body of the function to transfer the values from the original tuple to the resulting tuple. The type functions that produce new tuple headings are also used when we take up operations on relation values, since relation value headings are also based on tuple headings. The tuple type generation functions are used to build sets of those tuples for Relation Values.

Since we frequently need to create new structure types, we do some name shortening of the data type used for structure fields.

```

<<tuple module declarations>>=
const StructField = std.builtin.Type.StructField;

```

We start the implementation of tuple projection with a function that creates a new tuple heading type containing a subset of the attributes taken from another tuple heading type. The design strategy is to create a set from the projection attributes. Then, the new type is created by copying over only the structure field type information for the projected attributes. Note that we don't test for duplicated projection attribute names — the compiler will reject any structure with duplicated fields. We also don't test that the projection attribute ids refer to the heading because the compiler would catch the type error for the enum literal.

```

<<tuple module public functions>>=
/// The `ProjectHeading` function is a type function that creates a
/// new tuple heading type consisting of only the attributes given by,
/// `projAttrs`.
pub fn ProjectHeading(
    comptime Heading: type,
    comptime projAttrs: []const meta.FieldEnum(Heading),

```

```

) type {
    const heading_fields = meta.fields(Heading);
    if (heading_fields.len < proj_attrs.len)
        @compileError("attempt to project more attributes than are available");

    var proj_fields: [proj_attrs.len]StructField = undefined;
    inline for (proj_attrs, 0..) |proj_attr_id, proj_index| {
        proj_fields[proj_index] = heading_fields[@intFromEnum(proj_attr_id)]; ①
    }

    return @Type(.{
        .@"struct" = .{
            .layout = .auto,
            .backing_integer = null,
            .fields = &proj_fields,
            .decls = &.{},
            .is_tuple = false,
        },
    });
}

```

① One of the benefits of the field enumeration approach is that the integer value of the enumeration variant can be used as an index in the type heading fields. This technique is used frequently in this module.

With the projected heading in hand, the `project` function copies the attribute values from the given tuple to the appropriate attributes in the projected tuple.

```

<<tuple public functions>>=
/// The `project` function returns a new tuple whose heading contains only
/// those attributes given in the `projAttrs` slice.
pub fn project(
    self: Self,
    comptime projAttrs: []const AttributeId,
) RelTuple(ProjectHeading(TupleBase, projAttrs)) {
    const Projection = ProjectHeading(TupleBase, projAttrs);

    var proj_tuple: Projection = undefined;
    const proj_field_names = comptime meta.fieldNames(Projection);
    inline for (proj_field_names) |field_name| {
        @field(proj_tuple, field_name) = @field(self.reltuple, field_name);
    }
    return RelTuple(Projection).create(proj_tuple);
}

```

Some tuple algebras contain an `eliminate` function which is the counterpart of `project`, i.e. the new tuple has the heading of the given tuple minus the set of attributes given in the argument^[4]. That function is not provided here based on its limited usage and that experience shows it is subject to causing more errors when tuple headings change. Using `project` is a direct statement of the desired attributes and the result is a tuple with a heading matching the projected attributes.

An interesting edge case is the *nullary* projection, i.e. `t.project(&.{}).` This means that the resulting tuple has the empty set as its heading. It's rare to need such a projection, but it does

come up and ensures the algebra is consistent at its edge cases.

Tests

```
<<tuple module tests>>=
test "tuple project" {
    const Color = enum { Red, Green, Blue };
    const TestTuple = RelTuple(struct {
        name: []const u8,
        len: usize,
        weight: f32,
        color: Color,
    });
    const test_1 = TestTuple.create(.{
        .name = "foo",
        .len = 22,
        .weight = 50.0,
        .color = .Blue,
    });

    const proj_1 = test_1.project(&.{ .len, .color });
    try testing.expectEqual(@as(usize, 2), proj_1.degree());
    try testing.expectEqual(@as(usize, 22), proj_1.extract(.len));
    try testing.expectEqual(Color.Blue, proj_1.extract(.color));

    const proj_2 = test_1.project(&.{.name});
    try testing.expectEqual(@as(usize, 1), proj_2.degree());
    try testing.expectEqualStrings("foo", proj_2.extract(.name));

    // The nullary projection, i.e. the tuple with an empty set of attributes
    // is a required edge case.
    const proj_3 = test_1.project(&.{});
    try testing.expectEqual(@as(usize, 0), proj_3.degree());
}
```

Tuple Rename

There are circumstances where it is necessary to change the name of an attribute. For example, the *join* operation, and others, presented later perform a simple equi-join on equal-named attributes. It is not unusual for attributes to have different names even though they record the same semantic information. This results from attribute naming conventions that name the meaning of the attribute from different semantic perspectives.

Following the previous pattern, a type function to produce the required heading is shown first.

```
<<tuple module public functions>>=
/// The `RenameHeading` function returns a new tuple heading type that has the
/// same number of attributes as the original tuple, but where zero or more
```

```

/// of the attributes is given a new name.
/// The `renamings` argument is expected to be a list of pairs.
/// Each pair consists of:
/// 1. An attribute identifier of the field to be renamed.
/// 2. The new name, as a string, to be given to the attribute.
pub fn RenameHeading(
    comptime Heading: type,
    comptime renamings: anytype,
) type {
    const heading_fields = meta.fields(Heading);
    var renamed_fields: [heading_fields.len]StructField = undefined;
    @memcpy(&renamed_fields, heading_fields);

    // Iterate over the renamings, changing the requested names.
    const HeadingAttributeId = meta.FieldEnum(Heading);
    inline for (renamings) |renaming| { ①
        assert(renaming.len == 2);
        const attr_id: HeadingAttributeId = renaming.@"0";
        const field_index = @intFromEnum(attr_id);
        renamed_fields[field_index].name = renaming.@"1";
    }

    return @Type(.{
        .@"struct" = .{
            .layout = .auto,
            .fields = &renamed_fields,
            .decls = &.{},
            .is_tuple = false,
        },
    });
}

```

- ① Note that an empty `renamings` argument will result in returning the same heading, *i.e.* the function is a no-op for empty `renamings`. The case is so infrequent that it is not deemed worth a test to short circuit the processing.

```

<<tuple public functions>>=
/// The `rename` function returns a tuple where the number of attributes and
/// their values are the same, but with zero or more of the attributes having
/// new names. The `remainings` argument is taken to be a list of previous
/// attribute id / new attribute name pairs. The pairs map an attribute id
/// for the given tuple to a new string name to which the attribute's name
/// is set.
pub fn rename(
    self: Self,
    comptime renamings: anytype,
) RelTuple(RenameHeading(TupleBase, renamings)) {
    const Renamed = RenameHeading(TupleBase, renamings);

    comptime var rename_map = enums.EnumMap(AttributeName, [:0]const u8){}; ②

```

```

    inline for (renamings) |renaming| {
        comptime rename_map.put(renaming.@"0", renaming.@"1");
    }

    var renamed: Renamed = undefined;
    const heading_ids = comptime enums.values(AttributeId);
    inline for (heading_ids) |attr_id| {
        const attr_name = @tagName(attr_id);
        const maybe_rename = comptime rename_map.get(attr_id);
        const rename_name = maybe_rename orelse attr_name;
        @field(renamed, rename_name) = @field(self.reltuple, attr_name);
    }

    return RelTuple(Renamed).create(renamed);
}

```

- ① The strategy is to create a mapping from previous attribute id to new name as given by the renamings. While copying the values over to the new tuple, the map is used to direct which old attribute value is given to the newly named attribute.

Tests

```

<<tuple module tests>>=
test "tuple rename" {
    const Color = enum { Red, Green, Blue };
    const TestTuple = RelTuple(struct {
        name: []const u8,
        len: usize,
        weight: f32,
        color: Color,
    });
    const test_1 = TestTuple.create(.{
        .name = "foo",
        .len = 22,
        .weight = 50.0,
        .color = .Blue,
    });

    const rename_1 = test_1.rename(.{
        .{ .len, "size" },
        .{ .color, "rgb" },
    });
    try testing.expectEqual(@as(usize, 22), rename_1.extract(.size));
    try testing.expectEqual(@as(Color, .Blue), rename_1.extract(.rgb));

    // It is allowed to rename nothing and the rename operation is a no-op.
    const rename_2 = test_1.rename(.{});
    try testing.expect(rename_2.eql(test_1));
}

```

Tuple Extend

Extending a tuple creates a new tuple by adding attributes to an existing tuple. Two functions are provided.

- The `extend` function extends a tuple by a single attribute. This is a common use case and the interface to the function is simpler.
- The `extendMany` function extends a tuple by zero or more attributes. Its interface requires lists of lists to specify the new attributes and is more complicated to use.

```
<<tuple module public functions>>=
/// The `ExtendHeadingSingle` function returns a new tuple heading type
/// by adding the attribute name / attribute type as given by `attr_name`
/// and `AttrType` arguments, respectively. Optionally, a default value
/// can be specified for the new attribute.
pub fn ExtendHeadingSingle(
    comptime Heading: type,
    comptime attr_name: [:0]const u8,
    comptime AttrType: type,
    comptime opt_default: ?AttrType,
) type {
    const heading_fields = meta.fields(Heading);
    const extended_field_count = heading_fields.len + 1;

    // Copy the original heading structure fields to the extended heading fields.
    var extended_heading_fields: [extended_field_count]StructField = undefined;
    @memcpy(extended_heading_fields[0..heading_fields.len], heading_fields);

    extended_heading_fields[extended_field_count - 1] = .{
        .name = attr_name,
        .type = AttrType,
        .default_value_ptr = if (opt_default) |default| ①
            @as(?*const anyopaque, @ptrCast(&default))
        else
            null,
        .is_comptime = false,
        .alignment = if (@sizeOf(AttrType) == 0) 1 else @alignOf(AttrType),
    };

    return @Type(.{
        .@"struct" = .{
            .layout = .auto,
            .fields = &extended_heading_fields,
            .decls = &.{},
            .is_tuple = false,
        },
    });
}
```

① This incantation was taken from `enums.zig` in the standard library.

```

<<tuple public functions>>
/// The `extend` function creates a new `RelTuple` value that has
/// one additional attribute. This function is provided as a more
/// convenient interface for the common case of extending a tuple by
/// only one attribute. All the attributes and values of the original
/// tuple are preserved. The `attr_name` argument gives the name of
/// the attribute to be added to the tuple. The `AttrType` argument
/// gives the type of the added attribute. The value of the extended
/// attribute is given by either by the `attr_value` or `opt_default`
/// arguments. It is not allowed for both `attr_value` and `opt_default`
/// to be `null`. If `opt_default` is not `null`, then it establishes
/// a default value for the extended attribute. If `attr_value` is not
/// `null`, then it is the value to which the extended attribute is set.
/// If both are not `null`, the `attr_value` is taken as the value for
/// the extended attribute, but the default value for the extended tuple
/// is also established. This flexibility allows new tuple types to be
/// established by invoking `ExtendHeadingSingle` with a default value for
/// the extended attribute, making it possible to create tuples without
/// having to specify every attribute value. The returned tuple has the
/// same heading as the given tuple with the addition of the `attr_name`
/// attribute which is always set to some value as previously described.

pub fn extend(
    self: Self,
    comptime attr_name: [:0]const u8,
    comptime AttrType: type,
    attr_value: ?AttrType,
    comptime opt_default: ?AttrType,
) RelTuple(ExtendHeadingSingle(TupleBase, attr_name, AttrType, opt_default)) {
    const Extended = ExtendHeadingSingle(TupleBase, attr_name, AttrType, opt_default);

    var extended_base: Extended = undefined;
    const heading_ids = comptime enums.values(AttributeId);
    inline for (heading_ids) |orig_attr_id| {
        const orig_attr_name = @tagName(orig_attr_id);
        @field(extended_base, orig_attr_name) = @field(self.reltuple, orig_attr_name);
    }

    if (attr_value) |extend_value| {
        @field(extended_base, attr_name) = extend_value;
    } else {
        const field_index = comptime meta.fieldIndex(Extended, attr_name).?;
        const ExtendAttributeId = comptime meta.FieldEnum(Extended);
        const extend_attr_id: ExtendAttributeId = @enumFromInt(field_index);
        const default_value = comptime meta.fieldInfo(
            Extended,
            extend_attr_id,
        ).default_value_ptr;
        if (default_value) |default_ptr| {
            const default = @as(*align(1) const AttrType, @ptrCast(default_ptr))..*;
            @field(extended_base, attr_name) = default;
        }
    }
}

```

```

    } else {
        panic(
            "must specify an attribute value or there must be a default value\n",
            .{},
        );
    }
}

return RelTuple(Extended).create(extended_base);
}

```

Tests

```

<<tuple module tests>>=
test "tuple extend" {
    const TestTuple = RelTuple(struct {
        name: []const u8,
        len: usize,
        weight_lb: f32,
    });
    const test_1 = TestTuple.create(.{
        .name = "foo",
        .len = 22,
        .weight_lb = 50.0,
    });

    const extended_1 = test_1.extend(
        "weight_kg",
        f32,
        test_1.extract(.weight_lb) / 2.2,
        null,
    );
    try testing.expect((50.0 / 2.2) == extended_1.extract(.weight_kg));

    const extended_2 = test_1.extend(
        "weight_oz",
        f32,
        test_1.extract(.weight_lb) * 16.0,
        null,
    );
    try testing.expect((50.0 * 16.0) == extended_2.extract(.weight_oz));

    // Default values means the attribute value of the extension is optional.
    const extended_opt = test_1.extend(
        "weight_oz",
        f32,
        null,
        16.0,
    );
    try testing.expect(16.0 == extended_opt.extract(.weight_oz));
}

```

```

// extending composes
const extended_3 = test_1
    .extend("weight_kg", f32, test_1.extract(.weight_lb) / 2.2, null)
    .extend("weight_oz", f32, 16.0, null);
try testing.expect(50.0 / 2.2 == extended_3.extract(.weight_kg));
try testing.expect(16.0 == extended_3.extract(.weight_oz));
}

```

The extendMany operation on tuples returns a new tuple that includes multiple additional attributes.

```

<<tuple module public functions>>=
/// The `ExtendHeadingMulti` function returns a new tuple heading type by adding the
/// attributes given in the `extensions` argument.
/// The `extensions` argument is taken as a list of extension specifications.
/// Each extension specification is itself a three element list.
///
/// 1. The first element is the name of a new attribute as a string.
/// 2. The second element is the data type of the new attribute.
/// 3. The third element is an optional default value that is given to the
///     extended attribute if a value is not specified for it.
pub fn ExtendHeadingMulti(
    comptime Heading: type,
    comptime extensions: anytype,
) type {
    const heading_fields = meta.fields(Heading);
    const extended_field_count = heading_fields.len + extensions.len;

    // Copy the original heading structure fields to the extended heading fields.
    var extended_heading_fields: [extended_field_count]StructField = undefined;
    @memcpy(extended_heading_fields[0..heading_fields.len], heading_fields);

    // Create the extension attribute fields by appending to the accumulating
    // field array.
    inline for (extensions, heading_fields.len..) |extension, ext_index| {
        assert(extension.len == 3);
        const AttrType = extension.@"1";
        const opt_default: ?AttrType = extension.@"2";
        extended_heading_fields[ext_index] = .{
            .name = extension.@"0",
            .type = AttrType,
            .default_value_ptr = if (opt_default) |default|
                @as(?*const anyopaque, @ptrCast(&default))
            else
                null,
            .is_comptime = false,
            .alignment = if (@sizeOf(AttrType) == 0) 1 else @alignOf(AttrType),
        };
    }
}

```

```

return @Type(.{
    .@"struct" = .{
        .layout = .auto,
        .fields = &extended_heading_fields,
        .decls = &.{},
        .is_tuple = false,
    },
});
}

```

```

<<tuple public functions>>=
/// The `extendMany` function creates a new `RelTuple` value that has zero
/// or more additional attributes. All the attributes and values of the
/// original tuple are preserved. The `extendAttrs` argument is taken as
/// a list of triples. Each triple contains:
/// 1. attribute name
/// 2. attribute type
/// 3. optional attribute default value
/// See `ExtendHeadingMulti` for more information.
///

/// The `values` argument is taken as a list of pairs. Each pair contains:
/// 1. the attribute id of an extended attribute
/// 2. the value to which the extended attribute is to be set
///

/// Any extended attributes that are not contained in the `values` list
/// are assigned their respective default value. It is not allowed for an
/// extended attribute to have no value specified in the `values` argument
/// and to have no default value specified in the `extendAttr` argument.
pub fn extendMany(
    self: Self,
    comptime extendAttrs: anytype,
    values: anytype,
) RelTuple(ExtendHeadingMulti(TupleBase, extendAttrs)) {
    const Extended = ExtendHeadingMulti(TupleBase, extendAttrs);

    var extended_base: Extended = undefined;
    const heading_ids = comptime enums.values(AttributeId);
    inline for (heading_ids) |attr_id| {
        const attr_name = @tagName(attr_id);
        @field(extended_base, attr_name) = @field(self.reltuple, attr_name);
    }

    const ExtendedAttributeId = meta.FieldEnum(Extended); ①
    comptime var extended_attr_ids = enums.EnumSet(ExtendedAttributeId).initEmpty();
    inline for (extendAttrs) |extend_attr| {
        const attr_name: [:0]const u8 = extend_attr.@"0";
        const field_index = comptime meta.fieldIndex(Extended, attr_name).?;
        const attr_id: ExtendedAttributeId = @enumFromInt(field_index);
        comptime extended_attr_ids.insert(attr_id);
    }
}

```

```

    }

    inline for (values) |value| { ❷
        const attr_id: ExtendedAttributeId = value@"0";
        assert(extended_attr_ids.contains(attr_id));
        const attr_name = @tagName(attr_id);
        const AttrType = @FieldType(Extended, attr_name);
        const attr_value: AttrType = value@"1";
        @field(extended_base, attr_name) = attr_value;
        comptime extended_attr_ids.remove(attr_id);
    }

    inline while (comptime extended_attr_ids.count() != 0) { ❸
        const bit = comptime extended_attr_ids.bits.findFirstSet().?;
        const attr_id: ExtendedAttributeId = @enumFromInt(bit);
        const attr_name = @tagName(attr_id);
        const AttrType = @FieldType(Extended, attr_name);
        const default_value = comptime meta.fieldInfo(Extended, attr_id).default_value_ptr;
        if (default_value) |default_ptr| {
            const default = @as(*align(1) const AttrType, @ptrCast(default_ptr))..*;
            @field(extended_base, attr_name) = default;
        } else {
            @compileError("no value or default value specified for: '" ++
                fmt.comptimePrint("{s}\n", attr_name));
        }
        comptime extended_attr_ids.remove(attr_id);
    }

    return RelTuple(Extended).create(extended_base);
}

```

- ❶ To ensure that every extended attribute is set to a value, we build an enumeration set that contains the attribute ids of the extension attributes. As each extended attribute is considered, we test that the attribute is contained in the extension set and after its value is assigned then the attribute id is removed from the set.
- ❷ Set the values that are specified in the `values` argument. Note the attribute name is checked and after it is set, the attribute id is removed from the enum set.
- ❸ Anything that remains in the enum set is assigned its default value. Note that it is not possible to use the iterator provided by the enum set because it cannot be resolved at `comptime`. So, we just find the first bit set and after it is used it is removed from the enum set. It's a little wordier, but resolves at `comptime`.

Tests

```

<<tuple module tests>>=
test "tuple extendMany" {
    const TestTuple = RelTuple(struct {
        name: []const u8,
        len: usize,
        weight_lb: f32,

```

```

});  

const test_1 = TestTuple.create(.{  

    .name = "foo",  

    .len = 22,  

    .weight_lb = 50.0,  

});  

// New values for the extended attributes are fully specified.  

const extended = test_1.extendMany(.{  

    .{ "weight_kg", f32, null },  

    .{ "weight_oz", f32, null },  

}, .{  

    .{ .weight_oz, test_1.extract(.weight_lb) * 16.0 },  

    .{ .weight_kg, test_1.extract(.weight_lb) / 2.2 },  

});  

try testing.expect(50.0 == extended.extract(.weight_lb));  

try testing.expect((50.0 / 2.2) == extended.extract(.weight_kg));  

try testing.expect((50.0 * 16.0) == extended.extract(.weight_oz));  

// Default values are used to set the extended attribute values.  

const extend_2 = test_1.extendMany(.{  

    .{ "len_in", usize, 22 * 12 },  

    .{ "len_meter", usize, 22 * 100 },  

}, .{});  

try testing.expect(22 == extend_2.extract(.len));  

try testing.expect((22 * 12) == extend_2.extract(.len_in));  

try testing.expect((22 * 100) == extend_2.extract(.len_meter));  

// Part specified values, part default values.  

const extended_3 = test_1.extendMany(.{  

    .{ "weight_oz", f32, 50.0 * 16.0 },  

    .{ "weight_kg", f32, null },  

}, .{  

    .{ .weight_kg, test_1.extract(.weight_lb) / 2.2 },  

});  

try testing.expect((50.0 / 2.2) == extended_3.extract(.weight_kg));  

try testing.expect((50.0 * 16.0) == extended_3.extract(.weight_oz));  

}

```

Tuple Wrap

The wrap operation on a tuple combines zero or more attributes together to form a tuple-valued attribute. This operation is the primary way to construct tuple-value attributes.

It is useful to view tuple wrapping using a tabular display^[5]. Consider the following tuple:

RelTuple				
name	number	street	city	state
[]const u8	usize	[]const u8	[]const u8	[]const u8

Rhonda	1050	Elm Street	San Francisco	CA
--------	------	------------	---------------	----

If we wrap the `number` and `street` attributes into a tuple-valued attribute named `address`, we get:

RelTuple				
name	city	state	address	
[]const u8	[]const u8	[]const u8	RelTuple	
			+-----+ number street	
			+-----+	
			usize []const u8	
-----+ Rhonda	San Francisco CA	1050	Elm Street	
-----+				

We can go a step farther and wrap the `address`, `city`, and `state` attributes into a tuple-valued `location` attribute and obtain:

RelTuple				
name	location			
[]const u8	RelTuple			
	+-----+ address	city	state	
	+-----+ RelTuple	[]const u8	[]const u8	
	+-----+ number street			
	+-----+ usize []const u8			
-----+ Rhonda	1050	Elm Street	San Francisco CA	
-----+				

This is a visual indication that Relation Tuples can hold just scalars or tuple-valued attributes and tuple-valued attributes can hold other tuple-valued attributes. Tuples may also hold relation-valued attributes and we show that in the next section.

```
<<tuple module public functions>>=
/// The `WrapHeading` type function generates a new `RelTuple` type.
/// The new tuple heading contains all the attributes of the original
/// heading minus those attributes in the `wrap_attrs` slice plus a new
```

```

/// attribute named, `tuple_valued_attr`. The type of `tuple_valued_attr`  

/// is a `RelTuple` that contains the attributes given by the `wrap_attrs`  

/// slice.  

pub fn WrapHeading(  

    comptime Heading: type,  

    comptime tuple_valued_attr: [:0]const u8,  

    comptime wrapAttrs: []const meta.FieldEnum(Heading),  

) type {  

    const HeadingAttributeId = meta.FieldEnum(Heading); ①  

    const wrapped_set = enums.EnumSet(HeadingAttributeId).initMany(wrapAttrs);  
  

    const heading_fields = meta.fields(Heading);  

    // +1 -- for the tuple-valued attribute  

    const wrapped_field_count = heading_fields.len - wrapAttrs.len + 1;  

    var wrapped_heading_fields: [wrapped_field_count]StructField = undefined;  

    var wrapped_field_counter: usize = 0;  

    for (heading_fields, 0...) |heading_field, field_index| {  

        if (!wrapped_set.contains(@enumFromInt(field_index))) {  

            wrapped_heading_fields[wrapped_field_counter] = heading_field;  

            wrapped_field_counter += 1;
        }
    }
}  
  

const WrappedTuple = RelTuple(ProjectHeading(Heading, wrapAttrs)); ②  

wrapped_heading_fields[wrapped_field_counter] = .{  

    .name = tuple_valued_attr,  

    .type = WrappedTuple,  

    .defaultValue_ptr = null,  

    .is_comptime = false,  

    .alignment = if (@sizeOf(WrappedTuple) == 0) 1 else @alignOf(WrappedTuple),
};  
  

return @Type(.{
    .@("struct") = .{
        .layout = .auto,
        .fields = &wrapped_heading_fields,
        .decls = &.{},
        .is_tuple = false,
    },
});
}
}

```

- ① The strategy is to use a set initialized by the `wrapAttrs` to determine which field information to copy from the given heading to form the new heading.
- ② By projecting out the `wrapAttrs` we can create a tuple-valued attribute whose field information is appended to the end field information for the new heading.

```

<<tuple public functions>>=
/// The `wrap` function creates a new `RelTuple` value consisting
/// of the attributes in the given tuple plus a `tuple_valued_attr`.

```

```

/// The `tuple_valued_attr` is tuple-valued and contains the attributes
/// that are in the `wrap_attrs` argument. The net effect is that
/// `wrap()` creates a new tuple that has a nested tuple named,
/// `tuple_valued_attr`, and consisting of attributes that were previously
/// contained in the given tuple.
pub fn wrap(
    self: Self,
    comptime tuple_valued_attr: [:0]const u8,
    comptime wrapAttrs: []const AttributeId,
) RelTuple(WrapHeading(TupleBase, tuple_valued_attr, wrapAttrs)) {
    const Wrapped = WrapHeading(TupleBase, tuple_valued_attr, wrapAttrs);
    const wrapped_fields = meta.fields(Wrapped);

    var wrapped: Wrapped = undefined;
    inline for (wrapped_fields) |wrapped_field| {
        const attr_name = wrapped_field.name;
        if (comptime mem.eql(u8, attr_name, tuple_valued_attr)) { ❶
            const WrappedAttrType = wrapped_field.type;

            var wrap_value: WrappedAttrType.TupleBase = undefined;
            const wrap_attr_ids = comptime enums.values(WrappedAttrType.AttributeId);
            inline for (wrap_attr_ids) |wrap_attr_id| {
                const wrap_attr_name = @tagName(wrap_attr_id);
                @field(wrap_value, wrap_attr_name) = @field(self.reltuple, wrap_attr_name);
            }
            @field(wrapped, attr_name) =
                RelTuple(WrappedAttrType.TupleBase).create(wrap_value);
        } else {
            @field(wrapped, attr_name) = @field(self.reltuple, attr_name);
        }
    }

    return RelTuple(Wrapped).create(wrapped);
}

```

- ❶ There are two cases to consider. The tuple-valued attribute has to be created from values of the given tuple. For all other attributes, the values from the given tuple are simply copied.

Tests

```

<<tuple module tests>>=
test "tuple wrap" {
    const TestTuple = RelTuple(struct {
        name: []const u8,
        number: usize,
        street: []const u8,
        city: []const u8,
        state: []const u8,
    });
    const test_1 = TestTuple.create.{
```

```

        .name = "Rhonda",
        .number = 1050,
        .street = "Elm Street",
        .city = "San Francisco",
        .state = "CA",
    });

// Wrap .number and .street into a tuple-valued attributed named "address"
const test_2 = test_1.wrap("address", &{ .number, .street });
const address_2 = test_2.extract(.address);
try testing.expectEqual(@as(usize, 1050), address_2.extract(.number));
try testing.expectEqualStrings("Elm Street", address_2.extract(.street));

// Wrapped attributes may be tuple-valued where one of the wrapped is itself
// a tuple.
const test_3 = test_2.wrap("location", &{ .address, .city, .state });
try testing.expectEqualStrings("Rhonda", test_3.extract(.name));
try testing.expectEqualStrings("CA", test_3.extract(.location).extract(.state));

// Rather than perform the two wrappings from above in separate steps,
// the wrap operation composes nicely. The order of composition is
// important.
const composed = test_1
    .wrap("address", &{ .number, .street })
    .wrap("location", &{ .address, .city, .state });

try testing.expectEqual(@as(usize, 2), composed.degree());
try testing.expectEqualStrings(
    "San Francisco",
    composed.extract(.location).extract(.city),
);

// Nullary wrapping. It is possible to wrap the empty set.
// Not very useful, but edge cases matter.
const test_5 = test_1.wrap("empty", &{});
try testing.expectEqual(@as(usize, 6), test_5.degree());
try testing.expectEqual(@as(usize, 0), test_5.extract(.empty).degree());
}

```

Tuple Unwrap

The inverse of wrap is unwrap. This operation undoes a wrap result by taking a tuple-valued attribute and mixing in its attributes with the other attributes of the tuple. The tuple-valued attribute itself is discarded. Unwrapping has the potential of causing a duplicate attribute should the tuple-valued attribute have attributes with the same name as other attributes of the given tuple. This is a compile error.

```

<<tuple module public functions>>=
/// The `UnwrapHeading` type function returns a new `RelTuple` heading
/// where the attributes of the `attr` attribute are added to the

```

```

/// result heading. Note that unwrapping a tuple-valued attribute may
/// result in a duplicate attribute name which results in a compile
/// time error.
pub fn UnwrapHeading(
    comptime Heading: type,
    comptime unwrap_attr_id: meta.FieldEnum(Heading),
) type {
    const unwrapAttrInfo = meta.fieldInfo(Heading, unwrap_attr_id);
    const UnwrapAttrType = unwrapAttrInfo.type;
    if (!isRelTupleType(UnwrapAttrType))
        @compileError("unwrap attribute '" ++ unwrapAttrInfo.name ++
                      "' is not a RelTuple");

    const unwrap_fields = meta.fields(UnwrapAttrType.TupleBase);
    // Check for duplicate field names between Heading and the wrapped attribute fields.
    inline for (unwrap_fields) |unwrap_field| {
        if (meta.fieldIndex(Heading, unwrap_field.name) != null)
            @compileError("duplicate attribute '" ++ unwrap_field.name ++
                          "' when unwrapped");
    }

    const heading_fields = meta.fields(Heading);
    // -1 since the attribute being unwrapped is not in the result heading.
    const unwrapped_field_count = heading_fields.len + unwrap_fields.len - 1;
    var unwrapped_heading_fields: [unwrapped_field_count]StructField = undefined;
    var unwrapped_field_index: usize = 0;
    inline for (heading_fields, 0..) |heading_field, field_index| {
        if (field_index == @intFromEnum(unwrap_attr_id)) { ①
            inline for (unwrap_fields) |unwrap_field| {
                unwrapped_heading_fields[unwrapped_field_index] = unwrap_field;
                unwrapped_field_index += 1;
            }
        } else {
            unwrapped_heading_fields[unwrapped_field_index] = heading_field;
            unwrapped_field_index += 1;
        }
    }

    return @Type(.{
        .@struct" = .{
            .layout = .auto,
            .fields = &unwrapped_heading_fields,
            .decls = &.{},
            .is_tuple = false,
        },
    });
}

```

- ① Much like the wrap case, there are two cases in obtaining attribute values for the unwrapped tuple. For the tuple-valued attributes, we “raise” its type information to the resulting header. For all other attributes, the type information is simply copied to the result.

```

<<tuple public functions>>=
/// The `unwrap` function returns a new tuple value whose heading is
/// the union of attributes in the original tuple and the attributes of
/// the tuple-valued `wrapped_attr`, minus the `wrapped_attr` itself.
/// This function has the effect of *flattening* a tuple-valued attribute
/// back into the resulting tuple thus inverting any previous `wrap`
/// operation.
pub fn unwrap(
    self: Self,
    comptime wrapped_attr: AttributeId,
) RelTuple(UnwrapHeading(TupleBase, wrapped_attr)) {
    const Unwrapped = UnwrapHeading(TupleBase, wrapped_attr);
    const unwrapped_fields = meta.fields(Unwrapped);

    const unwrap_attr_value = self.extract(wrapped_attr).reltuple;

    var unwrapped: Unwrapped = undefined;
    inline for (unwrapped_fields) |unwrapped_field| {
        const unwrapped_name = unwrapped_field.name;
        if (comptime meta.fieldIndex(TupleBase, unwrapped_name) != null) {
            @field(unwrapped, unwrapped_name) = @field(self.reltuple, unwrapped_name);
        } else if (comptime meta.fieldIndex(Unwrapped, unwrapped_name) != null) {
            @field(unwrapped, unwrapped_name) = @field(unwrap_attr_value, unwrapped_name);
        } else {
            @compileError("lost wrapped attribute '" ++ unwrapped_name ++ "'");
        }
    }

    return RelTuple(Unwrapped).create(unwrapped);
}

```

Tests

```

<<tuple module tests>>=
test "tuple unwrap" {
    const AddressTuple = RelTuple(struct {
        number: usize,
        street: []const u8,
        city: []const u8,
        state: []const u8,
    });

    const addr_1 = AddressTuple.create(.{
        .number = 1050,
        .street = "Elm Street",
        .city = "San Francisco",
        .state = "CA",
    });
}

```

```

const PersonTuple = RelTuple(struct {
    name: []const u8,
    address: AddressTuple,
});

const person_1 = PersonTuple.create(.{
    .name = "Sally",
    .address = addr_1,
});

const unwrapped_1 = person_1.unwrap(.address);
try testing.expectEqual(@as(usize, 5), unwrapped_1.degree());
try testing.expectEqualStrings("Sally", unwrapped_1.extract(.name));
try testing.expectEqualStrings("CA", unwrapped_1.extract(.state));
}

```

Tuple Join

The concept behind a join of two tuples is to obtain a result that has a heading that contains:

1. The set of like-named attributes (possibly empty).
2. The remaining attributes of the first tuple.
3. The remaining attributes of the second tuple.

Concisely, we want the union of the two headings.

If the like-named attributes *do not have* equal values, then the join is empty. If there are no like-named attributes, *i.e.* the two headings are disjoint, then the join is successful since no comparison between like-named attributes fails.

To compute equality on the like-named attributes, we compute the intersection of the two headings. This design uses a `StaticStringMap` as a set of attribute names.

```

<<tuple module public functions>>=
/// Returns the intersection of the two tuple headings
/// as a string map containing the attribute names of the intersection.
pub fn IntersectFields(
    comptime OneHeading: type,
    comptime OtherHeading: type,
) std.StaticStringMap(void) {
    const one_fields = meta.fields(OneHeading);
    const other_fields = meta.fields(OtherHeading);
    // Worst case is that the tuple attribute name sets are subsets.
    const total_field_count = @min(one_fields.len, other_fields.len);

    var map_kvs_list: [total_field_count]struct { []const u8 } = undefined;
    var attr_index: usize = 0;
    inline for (one_fields) |one_field| {
        inline for (other_fields) |other_field| {
            if (mem.eql(u8, one_field.name, other_field.name)) {
                if (one_field.type != other_field.type)
                    @compileError("types for join attributes '" ++ one_field.name ++

```

```

        "' are not the same");
    map_kvs_list[attr_index] = .{one_field.name};
    attr_index += 1;
}
}

return std.StaticStringMap(void).initComptime(map_kvs_list[0..attr_index]);
}

```

With the heading intersection, the attributes of the joined heading can be computed.

```

<<tuple module public functions>>=
/// The `JoinHeadings` type function creates a new `RelTuple` type that contains the
/// join of the attributes in the `One` and `Other` arguments.
/// The new tuple heading consists of those attribute in common plus those attributes
/// from the argument headings that are not part of the common set of attributes.
pub fn JoinHeadings(
    comptime OneHeading: type,
    comptime OtherHeading: type,
) type {
    const one_fields = meta.fields(OneHeading);
    const other_fields = meta.fields(OtherHeading);

    const IntersectAttrMap = comptime IntersectFields(OneHeading, OtherHeading);
    const joined_field_count = one_fields.len + other_fields.len; // worst case
    var joined_fields: [joined_field_count]StructField = undefined;

    // All the fields of OneHeading are included.
    @memcpy(joined_fields[0..one_fields.len], one_fields);
    var joined_field_index: usize = one_fields.len;

    // All the fields of OtherHeading that are _not_ in the intersection
    // are included.
    inline for (other_fields) |other_field| {
        if (!IntersectAttrMap.has(other_field.name)) {
            joined_fields[joined_field_index] = other_field;
            joined_field_index += 1;
        }
    }

    return @Type(.{
        .@("struct" = .{
            .layout = .auto,
            .backing_integer = null,
            .fields = joined_fields[0..joined_field_index],
            .decls = &.{},
            .is_tuple = false,
        },
    });
}

```

```
}
```

```
<<tuple public functions>>=
/// The `join` function joins two tuples together. The header of the
/// resulting tuple is the union of the attributes of the given tuples.
/// It consists of those attribute in common between the two tuples
/// plus those attributes from each tuple that are not part of the set of
/// common attributes. The join of tuples with equal headers results in
/// the same tuple. The join of tuples with disjoint headers results in
/// a tuple with a heading that consists of all the attributes of both
/// tuples, i.e. the Cartesian product. Note that the attributes values
/// of any common attributes must be equal, otherwise the join is empty
/// and the returned result is `null`. For the case of disjoint headings,
/// the join is always non-empty as there are no common attributes that
/// would fail to be equal.

pub fn join(
    self: Self,
    joiner: anytype,
) ?RelTuple(JoinHeadings(TupleBase, @TypeOf(joiner).TupleBase)) {
    const Joiner = @TypeOf(joiner);
    if (comptime !isRelTupleType(Joiner))
        @compileError("type '" ++ @typeName(Joiner) ++ "' is not a RelTuple");

    const Joined = JoinHeadings(TupleBase, Joiner.TupleBase);
    const IntersectAttrMap = comptime IntersectFields(TupleBase, Joiner.TupleBase);
    const self_attribute_names = comptime meta.fieldNames(TupleBase);

    // Perform the join attribute equality evaluation first so as not to
    // complicate the attribute copy code. There are typically few common
    // attributes and we can bail out here if there is no match.
    inline for (self_attribute_names) |attr_name| {
        // Force comptime determination.
        const is_join_attr = comptime IntersectAttrMap.has(attr_name);
        if (is_join_attr and !attributeEql(
            @field(self.reltuple, attr_name),
            @field(joiner.reltuple, attr_name),
        )) {
            return null;
        }
    }

    const joined_attribute_names = comptime meta.fieldNames(Joined);
    var joined: Joined = undefined;
    inline for (joined_attribute_names) |attr_name| {
        // Converting the attribute name string into an AttributeId effectively tests
        // whether the attribute is part of the tuple.
        if (comptime meta.fieldIndex(TupleBase, attr_name) != null) {
            @field(joined, attr_name) = @field(self.reltuple, attr_name);
        } else if (comptime meta.fieldIndex(Joiner.TupleBase, attr_name) != null) {
            @field(joined, attr_name) = @field(joiner.reltuple, attr_name);
        }
    }
}
```

```

    } else {
        @compileError("can't find attribute '" + attr_name + "' in a tuple join");
    }
}

return RelTuple(Joined).create(joined);
}

```

```

<<tuple module tests>>=
test "tuple join" {
    // One common attribute, "name".
    const T1Tuple = RelTuple(struct { name: []const u8, age: u8 });
    const T2Tuple = RelTuple(struct { name: []const u8, height: u8 });

    const t1 = T1Tuple.create({ .name = "Bob", .age = 42 });
    const t2 = T2Tuple.create({ .name = "Bob", .height = 72 });

    const joined = t1.join(t2).?;
    try testing.expectEqual(@as(usize, 3), joined.degree());
    try testing.expectEqualStrings("Bob", joined.extract(.name));

    // Joining where the common attributes are unequal is empty.
    const t3 = T2Tuple.create({ .name = "Jane", .height = 72 });
    try testing.expect(t1.join(t3) == null);

    // Joining a tuple with itself or an equal tuple results in
    // the same tuple.
    const join_same = t1.join(t1).?;
    try testing.expectEqual(@as(usize, 2), join_same.degree());
    try testing.expectEqualStrings("Bob", join_same.extract(.name));
    try testing.expectEqual(@as(u8, 42), join_same.extract(.age));

    // Joining tuples with disjoint headings is the Cartesian product.
    const T4Tuple = RelTuple(struct { nomen: []const u8, height: u8 });
    const t4 = T4Tuple.create({ .nomen = "Bob", .height = 72 });

    const dis_joint = t1.join(t4).?;
    try testing.expectEqual(@as(usize, 4), dis_joint.degree());
    try testing.expectEqualStrings("Bob", dis_joint.extract(.name));
    try testing.expectEqualStrings("Bob", dis_joint.extract(.nomen));
    try testing.expectEqual(@as(u8, 42), dis_joint.extract(.age));
    try testing.expectEqual(@as(u8, 72), dis_joint.extract(.height));
}

```

Tuple to Relation Value

A Relation Tuple always can be transformed into a Relation Value of cardinality one without any additional information. To make this transformation, we need the reference to the `RelValue` type function. We discuss Relation Values in the next section. For now it is sufficient to say that `RelValue` is a type function much like `RelTuple` in that it takes a heading and returns a Relation Value type

matching the heading.

```
<<tuple module declarations>>=
const RelValue = relvalue.RelValue;
```

```
<<tuple public functions>>=
/// The `relValueFromTuple` function returns a relation value of cardinality 1
/// that contains the given tuple value.
pub fn relValueFromTuple(
    self: Self,
) RelValue(TupleBase) {
    return RelValue(TupleBase).create(&.self.reltuple), 100);
}
```

Testing of this function is deferred to the next section when `RelValue` types are introduced.

Formatted Tuple Output

During development of relational algebraic expressions, it is essential to be able to obtain a tabular representation of the intermediate results. In this section, we present a `format` function that is suitable for use with the standard library `format` function. This function outputs a character-based tabular representation of a `RelTuple`. We saw examples of it earlier in the discussion of the `wrap` operation.

There is a [separate](#) section holding the design and code for formatting the tabular output.

```
<<tuple module declarations>>=
const reformat = @import("reformat.zig");
const RelFormatter = reformat.RelFormatter;
const relvalue_allocator = relvalue.relvalue_allocator;
```

```
<<tuple public functions>>=
/// The `format` function is provided for use by the standard library
/// output formatting when specialized for `RelTuple` values.
/// Note that no formatting options are used and no format string is
/// accepted.
pub fn format(
    self: Self,
    writer: *std.Io.Writer,
) std.Io.Writer.Error!void {
    var formatter = RelFormatter.init(relvalue_allocator) catch
        return std.Io.Writer.Error.WriteFailed;
    defer formatter.deinit();
    try writer.writeAll(formatter.formatRelTuple(self)) catch
        return std.Io.Writer.Error.WriteFailed;
}
```

Testing is deferred until the section on formatted output.

Code layout

```
<<reltuple.zig>>
///! The `reltuple.zig` module implements the concept of a *Relation Tuple*.
///! A Relation Tuple, not to be confused with a Zig tuple,
///! is a component of the relational model of data. They are the
///! building blocks for *Relation Values*. A tuple represents a single
///! logical proposition about some aspect of a subject matter domain.
///! A tuple consists of a heading and a corresponding set of values.
///! The heading consists of a set of attributes names / attribute type
///! pairs. The attribute names form a set, implying that there are
///! no duplicate attributes. At all times, every attribute of a tuple
///! must have a valid value from the set of values for its data type.
///! A tuple is immutable in the sense that no functions are provided to
///! modify directly the attribute values and users of this module are
///! admonished to use only the functions provided here. The attributes
///! of a tuple heading have no inherent _order_. Relation Tuples are
///! type generators and new tuple headings are created in a many
///! circumstances. The notion of tuple heading equality is independent
///! of the order in memory that the implementation chooses to store the
///! heading attributes. Users of this module are also admonished not
///! to make assumptions about the apparent order of attributes since it
///! is subject to change as the implementation finds convenient.
<<edit warning>>
<<copyright info>>

const std = @import("std");
const builtin = @import("builtin");
const assert = std.debug.assert;
const panic = std.debug.panic;
const testing = if (builtin.os.tag == .freestanding)
    @import("resee_testing")
else
    std.testing;
const fmt = std(fmt);

pub const version = "1.0.0-a1";

<<tuple module declarations>>

<<relation tuple>>

<<tuple module public functions>>

<<tuple module private functions>>

<<tuple module tests>>
```

Relation Values

A Relation Value is a set of Relation Tuples, all of which have the same heading. Since all the tuples have the same heading, it is convenient to talk about a relation value as having that same heading. Since the relation value is a set and set members are known only by their identity, the implication is that no two relation tuples in a relation value may have the same attribute values for all the corresponding attributes in the tuples.

Like relation tuples, the heading of a relation value represents a logical predicate where the attributes are variables to that predicate. Since a relation value is a set of tuples, it represents a set of propositions which are considered **true**. Also similar to tuples, there is no inherent *order* among the tuples of a relation value as well as no order among the attributes.

Relation values are also *type generators* in that every relation value with a distinct heading represents a different static data type. In this algebra, we allow for both tuple-valued attributes and relation-valued attributes. Such “nesting” construct allow for much simpler interfaces in some circumstances than insisting upon “flat” tuples.

Memory Management

Since the subtitle of this book is, “A Microcontroller from the Bottom Up”, we must confront the problem of dynamic memory allocation and its management for relation values. It is conventional wisdom that programs running on a microcontroller with limited memory resources should not perform dynamic memory allocation. The reasoning behind this view is that these are most often long-running applications and that memory can become fragmented. Fragmentation of the system heap can lead to situations where it is not possible to obtain the required memory for a critical operation. Also as fragmentation proceeds, the time required to allocate memory is not deterministic.

Whether or not there is empirical evidence for this view is debatable. However, there are observations worth making.

- With deterministic requirements, it is often possible to allocate memory on a worst case basis. After all, if the requirements say that you must support six *gronkolator controllers* [6] all the required storage space can be allocated at compile time with no penalty for having done so. It is frequently the case that all the gronkolator controllers may not be active at the same time, but that doesn’t change the fact that they all may be active at some time. Some concept of a memory pool with an upper bound set at compile time is a reasonable design in many cases. If a request to acquire a seventh gronkolator controller is made, then there is a serious problem and, depending upon the consequences, may be denied or cause a panic, *i.e.* the request can be tossed up or shorted to ground. Even in cases where a maximum is not specified by requirement, practical upper bounds can usually be found. For example, there is no reason to insist upon having 10,000 instances of some resource when there is insufficient compute power to service them all.
- Most of the concerns over dynamic memory allocation in limited resource environments stem from using a single system heap for the allocation. Since “C” is the predominate language for this type of software, the focus is on the behavior of `malloc` and `free` with little or no considerations for other types of memory management beyond a single system heap.
- Some software engineers in this application area advocate performing allocation from the system heap at initialization time and then forbidding it afterwards. This mainly seems to be an attempt to avoid placing bounds on the data structures in the code. Allowing heap allocation within some specific time is a narrow, difficult line to draw and with `malloc` in the global namespace finding such rule violations is difficult to detect and enforce. This is similar to the problem encountered when development policies restrict the manner in which an

implementation language may be used by creating subsets of language features that are considered as “safe”. Checklists do not replace understanding and skill with the specific design circumstances.

- More troubling is that some system designs do not have an effective scheme to control the lifetimes of data instances and dynamic allocation becomes a free-for-all in the code. With unclear ownership and hand-off rules, coupled with pre-emptive threading, it is no wonder that mysterious failures due to incorrect memory management happen with disturbing regularity.

For the particular case of handling Relation Values, we must find a scheme that accommodates the fact that the values require memory for storage and the amount of memory required is not generally known at compile time. Unlike Relation Tuples, which have only one set of attributes, the tuple set of a Relation Value varies with the algebraic operation. However, the worst case size of a Relation Value is known at runtime and we can take advantage of the fact that Relation Values are ephemeral and do not have lifetimes that span the entire runtime of the program. To manage the memory for Relation Values, we make the following rules:

- All relation value expressions are evaluated in a session that must be started by invoking `relExprBegin` and terminated by `relExprEnd`. Such expression evaluation sessions may **not** be nested, i.e. there is at most one such ongoing session at any time. Any code may be executed between the invocations of `relExprBegin` and `relExprEnd`.
- An arena allocator is used during the evaluation of relation value expressions. At the end of the expression evaluations, the memory of the arena is discarded. This is triggered by `relExprEnd`.
- If any results of the relation value expressions are to be preserved, they must be stored elsewhere before the invocation of `relExprEnd`. As we will see in a later section, the results are often stored in a *Relation Variable*.
- Internally, the algebraic operations allocate memory for any result relation values plus any other memory needed for the operation itself. All allocated memory for which there is a well-defined lifetime is returned to the allocator when it is no longer needed. Also, any memory allocated that is not returned as a relation value is returned to the allocator. For example, if an error occurs and an algebraic operation does not return a relation value, any memory it may have allocated for the result is returned to the allocator. This rule is intended to reduce the pressure on the allocator and to prevent an extended sequence of errors from consuming all the available memory.
- Finally, and this rule is contrary to the usual conventions in Zig for memory allocation, failure to allocate memory results in a panic. The rational for this decision is:
 - Since relation value algebra is closed, we want to be able to compose the operations together from left to right. The usual idiom of using the Zig `try` mechanism is clumsy in this circumstance. Since `try` is a prefix operation, parentheses are required to group the closed operations sequentially.
 - In a freestanding environment, a fixed buffer is used for the child allocator of the arena. It is conventional in this environment to use system tests to ensure that memory requests do not overflow the fixed buffer and then add some margin. This does *not* guarantee that there is never a panic, but the probability is low and the approach provides visibility into the amount of memory used which is usually limited resource. This is similar to the way that memory is allocated for stack space. A freestanding application could allocate the arena on the stack, but it is helpful to keep stack usage small, consistent, and oriented to the flow of execution. Moving relation value memory allocation off the stack provides better insight to overall system memory usage and allows for independent sizing of the stack and relation value arena.
 - In an OS environment, we have choices for the child allocator of the arena. In this implementation, the testing allocator is used during tests otherwise the standard library general purpose allocator is used.

In the end, this design insists that relation value expressions be evaluated in a context where a dedicated arena allocator is used and the allocator is effectively a garbage collection mechanism. Relation value expressions do involve dynamic memory allocation, but in a manner that ameliorates concerns of a limited memory environment.

```
<<relvalue module declarations>>=
/// The allocator used to allocate all dynamic memory needed
/// during evaluation of a relational expression.
pub const relvalue_allocator = allocator_instanceallocator();
/// Use the dedicated testing allocator during testing.
var allocator_instance = heap.ArenaAllocator.init(
    if (builtin.is_test) testing.allocator else child_allocator,
);
/// Otherwise, use a fixed buffer allocator if we are freestanding
/// and the general purpose allocator otherwise.
var child_allocator = if (builtin.os.tag == .freestanding) fblk: {
    var alloc_buf: [0x4000]u8 = undefined; // 16 KiB -- needs a config option
    break :fblk heap.FixedBufferAllocator.init(&alloc_buf);
}
else
    heap.GeneralPurposeAllocator({});
```

With the memory allocation design completed, we show the code for controlling the relation value expression session lifetime.

```
<<relvalue module public functions>>=
/// The `relExprBegin` function must be invoked at the beginning of a
/// sequence of Relation Value expression evaluations.
pub fn relExprBegin() void {
    const capacity = allocator_instance.queryCapacity();
    if (capacity != 0) {
        panic("beginning relation value expression with an occupied arena: '{d} bytes'\n",
              .{capacity});
    }
}
```

```
<<relvalue module public functions>>=
/// The `relExprEnd` function is invoked to end a sequence of Relation Value
/// expression evaluations. This function returns any memory allocated during
/// the evaluations of relation value expressions.
pub fn relExprEnd() void {
    if (!allocator_instance.reset(
        if (builtin.is_test) .free_all else .retain_capacity)) { ①
        panic("arena allocator reset failed\n", .{});
    }
}
```

```
}
```

- ① For testing, we must free everything to keep the testing allocator from producing errors. Otherwise, we want to take advantage of the "pre-heating" that the `.retain_capacity` option offers.

For internal operations that need temporary memory, a more convenient interface to the `relvalue_allocator` is provided.

```
<<relvalue module private functions>>=
/// The `relExprAlloc` function allocates memory from the relvalue_allocator.
/// Failure to allocate the requested memory creates a `panic` condition.
fn relExprAlloc(
    comptime T: type,
    n: usize,
) []T {
    return relvalue_allocator.alloc(T, n) catch |err|
        panic("failed to allocate {d} bytes during a relation expression: {s}\n",
            .{n, @errorName(err)});
}
```

Representing a Relation Value

The storage required for a relation value is composed of that memory required to store the tuple set and that used used for an index to ensure tuple identity. This section shows the main functions used to define a relation value and the design decisions behind the data structures used for that purpose.

Relation Value types are constructed by the `RelValue` function which specifies the heading of the relation value is provided as an argument.

```
<<relation value>>=
/// `RelValue` is a type function that creates a new Relation Value data
/// type from a `Heading`. A `Heading` is a basic, unadorned Zig `struct`
/// that defines the attributes and data types for the relation value.
/// The `Heading` argument must be a valid heading as required by the
/// `RelTuple` function. A `RelValue` type contains a set of `Tuples`
/// all of which have the same `Heading`.
pub fn RelValue(
    comptime Heading: type,
) type {
    return struct {
        const Self = @This();

        <<relation fields>>
        <<relation declarations>>
        <<relation public functions>>
        <<relation private functions>>
    };
}
```

```
}
```

We make a number of declarations for symbolic shorthand.

```
<<relation declarations>>=
/// The base structure of a tuple for this relation value.
pub const TupleBase = Heading;
/// The type of a tuple for this relation value.
pub const Tuple = RelTuple(TupleBase);
/// The enumerated type representing tuple attributes.
pub const AttributeId = Tuple.AttributeId;
```

Since Relation Values are sets of Relation Tuples, we need access to the `reltuple.zig` module.

```
<<relvalue module declarations>>=
const reltuple = @import("reltuple.zig");
const RelTuple = reltuple.RelTuple;
const isRelTupleType = reltuple.isRelTupleType;
```

The design of the storage for the set of `RelTuple` instances must account for:

- `RelValue` values are immutable. Once created, they are not modified.
- The worst case set size is known at runtime but **not** at compile time.
- Access to tuples in the `RelValue` is most often sequential.

Given the simplicity of the required memory management, we simply allocate worst case space from the arena allocator and manage it as a slice of `Tuple`.

```
<<relation declarations>>=
const TupleStore = []Tuple;
```

```
<<relation fields>>=
/// Storage for the set of tuple values.
storage: TupleStore,
/// Index of the next available location in the storage slice.
next: usize,
```

Appending is the sole storage operation required internally to construct relation values.

```
<<relation private functions>>=
/// The `append` function adds a single `tuple` to the given relation value.
fn append(
    self: *Self,
    tuple: Tuple,
) void {
    self.storage[self.next] = tuple;
    self.next += 1;
```

```
}
```

As with Relation Tuples, we need a convenient way to determine if a given type is indeed a RelValue. We use the same device of a void field.

```
<<relation fields>>=
/// A field used to easily determine if a given `struct` is a Relation Value.
comptime is_relvalue: void = {},
```

A simple function is provided by the module to test types for being a RelValue.

```
<<relvalue module public functions>>=
/// The `isRelValueType` function returns `true` if the type given by the
/// `RelValueType` argument can be identified as resulting from an invocation
/// of the `RelValue` type function.
pub fn isRelValueType(
    comptime RelValueType: type,
) bool {
    return @typeInfo(RelValueType) == .@["struct" and @hasField(RelValueType,
"is_relvalue");
}
```

Enforcing Identity Constraints

A RelValue is a **set** of tuples and sets do not have duplicates. Each tuple in a RelValue may not equal any other tuple in the RelValue as defined by tuple equality. It is frequently the case that when constructing a RelValue from another RelValue that there are tuples in common that must be found and excluded from the result. Sometimes duplicated tuples are silently ignored, e.g. union. Other times duplicated tuples are an error, e.g. disjoint union.

A hash map is a clear choice to meet these requirements. Based on the memory management strategy, we choose the unmanaged flavor of a HashMap from the standard library as the basis for ensuring tuple uniqueness. As with storage, the size of the hash map does not change once created and we are able to perform a single allocation when the RelValue is created.

```
<<relation declarations>>=
const default_load_factor: u64 = 90; ①

/// The specialization of the `HashMapUnmanaged` used with this
/// particular `Tuple`.
const Index = HashMapUnmanaged(
    Tuple,
    void, ②
    TupleHashContext(Tuple, enums.values(AttributeId)),
    default_load_factor,
);
```

- ① A loading of 80% is usual. We clamp down on memory usage here. This number may need to be revisited when some characterization has been done on this module.
- ② It's a set. We are only interested in the keys and the entire Tuple is the key.

```
<<relation fields>>=
/// A specialized `HashMapUnmanaged` used to ensure tuple uniqueness.
index: Index,
```

Tuple Hashing

The standard library `HashMap` uses a *context* as a means of providing the two functions needed for hashing:

- a. a function to compute a hash value of the key and
- b. a function to determine equality of two keys.

There are multiple places where an hash map is useful in this module. But to be generally useful, the functions of the hash context need to be applied to different subsets of tuple attributes. We use a type function to specialize the hash map context for a particular tuple type and attribute subset.

The `reltuple` module has done the heavy lifting for determining attribute equality.

```
<<relvalue module declarations>>=
const attributeEql = reltuple.attributeEql;
```

```
<<relvalue module private functions>>=
/// The `TupleHashContext` type function returns a context suitable for
/// use with the standard library `HashMap`. The generated context can
/// be used to create a `HashMap` for a tuple of type, `TupleType`, considering
/// only the attributes given by `attr_ids`.
fn TupleHashContext(
    comptime TupleType: type,
    comptime attr_ids: []const TupleType.AttributeId,
) type {
    return struct {
        const CtxSelf = @This();

        pub fn hash(
            self: CtxSelf,
            tup: TupleType,
        ) u64 {
            _ = self;
            var hasher = std.hash.Fnv1a_64.init();
            inline for (attr_ids) |attr_id| {
                hasher.update(mem.asBytes(&tup.extract(attr_id)));
            }
            return hasher.final();
        }
    };
}
```

```

pub fn eql(
    self: CtxSelf,
    a: TupleType,
    b: TupleType,
) bool {
    _ = self;
    inline for (attr_ids) |attr_id| {
        if (!attributeEql(a.extract(attr_id), b.extract(attr_id))) ①
            return false;
    }
    return true;
}
};
}

```

① We already know how to compute attribute equality.

Relation Value Errors

There are a number of errors that can arise from the operations. We define an error set for them.

```

<<relvalue module declarations>>=
pub const RelValueError = error{
    DuplicateTuple,           // Attempt to insert duplicate tuple.
    RequiresSingularRelation, // The operation requires a relation of cardinality 1.
};

```

Creating Relation Values

Since a primary target for this module is microcontroller-based systems, this design limits the maximum size of a Relation Value tuple set to be the maximum value of a `u16` type. This is a conservative approach to limit total memory consumption. This value will be reconsidered once the module is characterized in terms of its performance and memory usage. For the use case envisioned, 64 Ki-tuples in a relation value is a large population. Note that because some algebraic operations can result in the product of the number of tuples of each term, e.g. join, in some cases the maximum number of tuples in a relation value term may be limited to 256.

```

<<relvalue module declarations>>=
/// The upper bound of this type determines the maximum number of tuples
/// that can be held in a `RelValue`.
pub const RelationCapacity = u16;

```

A single function is provided to create relation values from a set of tuples.

```

<<relation public functions>>=

```

```

/// The `create` function creates a new `RelValue` from the tuples supplied
/// by the `tuples` argument slice. The tuples slice must contain distinct
/// tuple values.
pub fn create(
    tuples: []const TupleBase,
) Self {
    var new_value = createCapacity(tuples.len);
    for (tuples) |tuple| {
        const inserted = new_value.insertTuple(Tuple.create(tuple));
        assert(inserted);
    }
    return new_value;
}

```

A boundary case for creating relation values is the relation value that contains no tuples.

```

<<relation public functions>>=
/// The `createEmpty` function creates a `RelValue` that contains
/// no tuples.
pub fn createEmpty() Self {
    return createCapacity(0);
}

```

There are interesting cases when a relation value has an empty heading.

```

<<relvalue module declarations>>=
/// The `RelValue` type that has an empty heading.
pub const EmptyHeadingRelValue = RelValue(struct {});

```

There are two interesting relation values that have empty headings.

- The relation value with an empty heading and no tuples is usually called “DUM”. DUM is the relational analog of `false` and acts as an annihilator when used in a relation join.
- The relation value with an empty heading and one tuple is usually called “DEE”. DEE is the relational analog of `true` and acts as an identity when used in a relation join. Note that there are only two possible relation values that have an empty header. There is only one possible tuple, the tuple whose value is empty, that can be inserted into a relation value with an empty heading.

These two relation values are not used frequently, but they elegantly solve the edge cases that do sometimes arise.

```

<<relvalue module public functions>>=
/// The `dum` function creates a `RelValue` that has an empty heading
/// and cardinality of zero.

```

```

pub fn dum() EmptyHeadingRelValue {
    return EmptyHeadingRelValue.createEmpty();
}

```

```

<<relvalue module public functions>>=
/// The `dee` function creates a `RelValue` that has an empty heading
/// and cardinality of one.
pub fn dee() EmptyHeadingRelValue {
    return EmptyHeadingRelValue.create(&.{{}});
}

```

Although relation values are immutable after construction, construction of the value itself starts with allocating the space for the tuple set and then inserting the tuples on-by-one. So for internal usage, we have several functions that are used by the algebraic functions to construct a relation value result.

```

<<relation private functions>>=
/// The `createCapacity` function creates a `RelValue` that can contain
/// at most `capacity` tuples. No the tuples are inserted into the
/// `RelValue` and the intended usage is by algebraic functions that
/// construct and return `RelValue` types.
fn createCapacity(
    capacity: usize,
) Self {
    assert (capacity <= math.maxInt(RelationCapacity));
    return .{
        .storage = relvalue_allocator.alloc(Tuple, capacity) catch |err|
            panic(
                "failed to allocate relvalue tuple storage: '{s}'\n",
                .{@errorName(err)},
            ),
        .next = 0,
        .index = iblk: {
            var index: Index = .{};
            index.ensureTotalCapacity(
                relvalue_allocator,
                @intCast(capacity),
            ) catch |err|
                panic(
                    "failed to allocate relation index: '{s}'\n",
                    .{@errorName(err)},
                );
            break :iblk index;
        },
    };
}

```

There are some cases where the result of an algebraic function contains all the tuples of one operand, e.g. union. For that use case, a specialized function is supplied.

```
<<relation private functions>>=
/// The `cloneCapacity` function creates a `RelValue` value that can
/// contain at most `capacity` tuples and inserts all the tuples of the
/// given `RelValue` into the returned `RelValue`. The intended usage
/// is by algebraic functions that construct and return `RelValue`
/// types and for which one operand is a subset of the returned value.
fn cloneCapacity(
    self: Self,
    capacity: usize,
) Self {
    var clone = createCapacity(capacity);

    for (self.body()) |self_tuple| {
        const inserted = clone.insertTuple(self_tuple);
        // All the tuples in the given relation must insertTuple since the clone
        // relation value starts empty. Note that if the requesting capacity
        // is too small to hold the given relation value, then `insertTuple`
        // will assert.
        assert(inserted);
    }

    return clone;
}
```

The functions provided so far to create relation values are primarily designed for local module usage and are convenient when a tuple set can be provided as a single argument. For some use cases, it is more convenient if the relation value creation can be done one tuple at a time, allowing other computations between each tuple insertion. The following construct decouples the building of the tuple set to allow other computation. This interface was designed specifically for the Relation Variable module. For that case, a `Builder` type is provided to allow single tuple insertions but not expose the internal mechanisms for storing the relation value tuple set. Creating a relation value using this builder concept is designed as a three step process.

1. Initialize a builder with the ultimate capacity.
2. Insert tuples one at a time.
3. Finalize the result, returning the newly created relation value.

Although the interface was created to satisfy a particular use case, other use cases can use the mechanism within its design constraints to build relation values one tuple at a time.

```
<<relation declarations>>=
/// The `Builder` type is provided with functions that help
/// construct relation values one tuple at a time.
pub const Builder = struct {
    /// Location where the relation value is stored during its creation time.
```

```

value: Self = undefined,
```

```

/// The `init` function is used first to set the ultimate
/// size of the relation value tuple set.
pub fn init(
    capacity: usize,
) Builder {
    return .{ .value = createCapacity(capacity) };
}

/// The `insert` function is used to insert a tuple into the
/// growing relation value.
pub fn insert(
    builder: *Builder,
    tuple: Tuple,
) bool {
    return builder.value.insertTuple(tuple);
}

/// The `finalize` function returns the newly created relation value.
pub fn finalize(
    builder: *Builder,
) Self {
    defer builder.value = undefined;
    return builder.value;
}
};

```

Despite the usage of an arena allocator as a garbage collector for evaluating a relational expression, a `destroy` function is provided for internal use. This is typically used when the lifetime of the memory is well defined or an error condition means that an allocated relation variable is not returned. In those cases, functions invoke `destroy` to return memory that is never returned to the caller. This has the effect of lowering the pressure on the allocator. This is especially useful given that the arena allocator for relation expressions is reset using the `.retain_capacity` option. We do not want error situations to distort the amount of memory retained for the next relation expression evaluation.

```

<<relation private functions>>=
/// The `destroy` function releases any storage for a `RelValue`.
/// It is intended for internal use where the lifetime of the is predictable
/// or where an error condition prevents returning previously allocated values.
fn destroy(
    self: *Self,
) void {
    relvalue_allocator.free(self.storage);
    self.index.deinit(relvalue_allocator);
    self.* = undefined;
}

```

All tuples in a relation value must be “inserted” into the value. The primary purpose of insertion is

to manage the uniqueness constraints on tuples by using the hash map for the relation value.

```
<<relation private functions>>=
fn insertTuple(
    self: *Self,
    tuple: Tuple,
) bool {
    const index_result = self.index.getOrPutAssumeCapacity(tuple); ①

    return if (index_result.found_existing)
        false
    else tblk: {
        self.append(tuple);
        break :tblk true;
    };
}
```

① Hash first to see if the tuple can be inserted.

Tests

A simple RelValue is used in many of the test cases below and a function to create it is provided for common use.

```
<<relvalue module tests>>=
// setting up test data
const RelTestType = struct {
    name: []const u8,
    value: usize,
};
const TestRelation = RelValue(RelTestType);

fn create_test_relation() TestRelation {
    return TestRelation.create(&.{.
        .{ .name = "foo", .value = 0 },
        .{ .name = "foo", .value = 1 },
        .{ .name = "foo", .value = 2 },
        .{ .name = "bar", .value = 0 },
        .{ .name = "bar", .value = 1 },
        .{ .name = "bar", .value = 2 },
    });
}
```

```
<<relvalue module tests>>=
test "create relation value" {
    relExprBegin(); defer relExprEnd();
```

```

const test_relation = create_test_relation();

try testing.expect(test_relation.cardinality() == 6);

try testing.expectEqual(@as(usize, 0), dum().degree());
try testing.expectEqual(@as(usize, 0), dum().cardinality());

try testing.expectEqual(@as(usize, 0), dee().degree());
try testing.expectEqual(@as(usize, 1), dee().cardinality());
}

```

Relation Value Introspection

As with Relation Tuples, a set of introspection functions is provided for Relation Values. Most of these are simple one line functions that provide some convenient syntactic sugar and are shown with little additional commentary.

```

<<relation public functions>>=
/// The `cardinality` function returns the number of tuples in the
/// given `RelValue`.
pub fn cardinality(
    self: *const Self,
) usize {
    return self.next;
}

```

```

<<relation public functions>>=
/// The `degree` function returns the number of attributes in the
/// heading of the given `RelValue`.
pub fn degree(
    self: *const Self,
) usize {
    _ = self;
    return Tuple.tuple_degree;
}

```

```

<<relation public functions>>=
/// The `attributes` function returns a slice containing the string names
/// of the attributes of the given `RelValue`.
pub fn attributes() []const []const u8 {
    return meta.fieldNames(TupleBase);
}

```

```

<<relation public functions>>=
/// The `attributeName` function returns a string name
/// of the attribute of the given `RelValue` given by the `attr_id` argument.
/// Recall, that the `attr_id` can be conveniently given as an enum literal.
pub fn attributeName(
    attr_id: AttributeId,
) []const u8 {
    return Tuple.attributeName(attr_id);
}

```

```

<<relation public functions>>=
/// The `AttributeType` function is a type function that returns the data type of
/// the attribute identified by, `attr_id`.
pub fn AttributeType(
    comptime attr_id: AttributeId,
) type {
    return @FieldType(TupleBase, @tagName(attr_id));
}

```

Relation Value Algebra

In this section, we describe the algebraic operations on `RelValue` variables provided by this algebra. Recall that relational algebra has an open-ended set of operations. There is a set of primitive operations that can be used to derive the other operations, but the operations described here are given because of their utility and frequency of use. Not every conceivable relational operation is given here and more may be added as need presents itself. The set of operations is rather large and suffers from what large feature sets frequently do: most applications use only a few of the provided operations, but over a large set of applications, all the operations get used.

The following defines some groups of related operations:

Direct value access	Operations that allow direct access to relation value attributes.
Relation value comparisons	Operations on the set characteristics of relation values, e.g. a test for set membership.
Aggregate operations	Computations across all the attributes of a relation value, e.g. <code>sum</code> or <code>max</code> .
Set operations	Traditional set operations, such as <code>union</code> , extended to relation values.
Projection	Creating relation values with a subset of attributes.
Restriction	Creating relation values with a subset of tuples.
Join	Creating relation values whose heading is a union of two headings.
Extending	Creating relation values by adding new attributes.
Update	Creating new relation values that are modifications to

	existing values.
Rename	Creating new relation values whose attribute names have been changed.
Wrapping	Creating relation values containing tuple-valued attributes.
Grouping	Creating relation values containing relation-valued attributes.
Transitive closure	Creating relation values containing the transitive closure of a starting relation value.
Iteration	Visiting the tuples in a relation value in a specified order.

Direct Value Access

There are only a few ways to directly access the tuples or attribute values in a `RelValue`.

Attribute Slices

It is sometimes useful to extract from a relation value all the values of a given attribute. The following function returns the values of attributes as an array in a specified order.

```
<<relation public functions>>=
/// The `attributeSlice` function returns a slice pointing to an array
/// whose size is the cardinality of the given relation value and whose
/// values are those values of the `attr_id` attribute. The relation
/// value is conceptually sorted by the `sortAttrs` in the sorting order
/// given by `order`. The returned array values are then placed in the
/// array in the order of the sorted tuples. If no `sortAttrs` are given,
/// i.e. `sortAttrs.len == 0`, then the order of the returned attribute
/// values is indeterminate. The memory for the array is allocated from
/// the relation values arena allocator. This implies that any returned
/// array values must be used within the expression or, if the values
/// are intended to survive the scope of the relational expression,
/// stored in a variable whose lifetime is longer than that of the
/// relational expression. Note that the values in the returned array
/// do *not* necessarily form a set and there may be duplicated values. To
/// ensure a set, you can project the relation value on the desired
/// attribute id before retrieving the array.
pub fn attributeSlice(
    self: *const Self,
    comptime attr_id: AttributeId,
    comptime order: IterationOrder,
    comptime sortAttrs: []const AttributeId,
) []const AttributeType(attr_id) {
    const ResultType = AttributeType(attr_id);
    var resultArray: []ResultType = relExprAlloc(ResultType, self.cardinality());

    if (comptime sortAttrs.len != 0) {
```

```

// Initialize a sorting map to the identity mapping. ①
var sort_map: []RelationCapacity =
    relExprAlloc(RelationCapacity, self.cardinality());
defer relvalue_allocator.free(sort_map);

var index: RelationCapacity = 0;
while (index < sort_map.len) : (index += 1) {
    sort_map[index] = index;
}

sort.heap(
    RelationCapacity,
    sort_map,
    RelSortContext{ .tuples = self.body() },
    SortLessThanFn(order, sort_attrs),
);

const self_body = self.body();
for (0..sort_map.len) |sort_map_index| {
    resultArray[sort_map_index] =
        self_body[sort_map[sort_map_index]].extract(attr_id);
}
} else {
    for (self.body(), 0..) |self_tuple, tuple_index| {
        resultArray[tuple_index] = self_tuple.extract(.attr_id);
    }
}

return resultArray;
}

```

① Sorting is discussed [below](#).

```

<<relvalue module tests>>=
test "array from relation value" {
    relExprBegin(); defer relExprEnd();

    const test_relation = create_test_relation();
    const values = test_relation.project(&{.value});
    const value_array = values.attributeSlice(.value, .desc, &{.value});
    try testing.expectEqual(@as(usize, 3), value_array.len);
    try testing.expectEqual(@as(usize, 2), value_array[0]);
    try testing.expectEqual(@as(usize, 1), value_array[1]);
    try testing.expectEqual(@as(usize, 0), value_array[2]);
}

```

Tuple Access

Functions are provided for two special cases. Often, you are dealing with relation values containing a single tuple. For a singular `RelValue`, its tuple can be returned without any loss of information.

This is the counterpart to a Relation Tuple that can be converted to a Relation Value of cardinality one.

```
<<relation public functions>>=
/// The `tupleFromRelValue` function returns the tuple of a `RelValue`
/// of cardinality one. It is an error if the given `RelValue` is not
/// of cardinality one.
pub fn tupleFromRelValue(
    self: *const Self,
) RelValueError!Tuple {
    return if (self.cardinality() == 1)
        self.storage[0]
    else
        RelValueError.RequiresSingularRelation;
}
```

Attribute Access

Given a singular relation, a convenience function to obtain the value of an attribute is provided.

```
<<relation public functions>>=
/// The `extract` function returns the attribute value associated with
/// `attr_id` from the given `RelValue`. It is an error to attempt to
/// `extract` an attribute value from a relation value whose cardinality
/// is not one.
pub fn extract(
    self: *const Self,
    comptime attr_id: AttributeId,
) RelValueError!AttributeType(attr_id) {
    return (try self.tupleFromRelValue()).extract(attr_id);
}
```

Body

Direct access to the tuples of a relation value is provided as a slice pointing into the tuple storage. This function is used internally by algebraic operations to access the tuple set of relation values.

```
<<relation public functions>>=
/// The `body` relation returns a slice pointing to the tuples in the
/// given `RelValue`. Iterating across the slice gives the tuples in
/// an unspecified order. Recall that tuples in a Relation Value have
/// no inherent order. If an order is required, an `iterator` is provided
/// to visit the tuples based on a sorting order.
pub fn body(
    self: *const Self,
```

```
) []const Tuple {
    return self.storage[0..self.next];
}
```

Relation Value Comparisons

Since relation values are sets of tuples, we can determine if the sets are equal, subsets, or supersets of each other.

Contains

Set membership is computed using the `contains` function. This name was chosen to be familiar in a Zig context as it is the same function name used in various standard library functions that test for set membership.

```
<<relation public functions>>=
/// The `contains` function returns `true` if `tuple` is a member of the
/// set of tuples of the given relation value and `false` otherwise.
pub fn contains(
    self: *const Self,
    tuple: Tuple,
) bool {
    return self.index.contains(tuple);
}
```

Empty

Variations on the `contains` idea yield operations to determine if a relation value contains any or no tuples.

```
<<relation public functions>>=
/// The `empty` function returns `true` if the cardinality of the
/// given `RelValue` is zero and `false` otherwise.
pub fn empty(
    self: *const Self,
) bool {
    return self.cardinality() == 0;
}
```

Not Empty

```
<<relation public functions>>=
/// The `notEmpty` function returns `true` if the cardinality of the
/// given `RelValue` is *not* zero and `false` otherwise.
```

```

pub fn notEmpty(
    self: *const Self,
) bool {
    return !self.empty();
}

```

Equality

Next we consider relation value equality. There are two components to relation value equality

- Heading equality and
- Tuple set equality

The `reltuple` module defines a function to determine if two headings are equal and we use it here.

```

<<relvalue module declarations>>=
const equivalentHeading = reltuple.equivalentHeading;

```

Recall that headings may be equal even if the order of storage in the implementation is different. However, the implementation of the set operations need to know if two relation values have the same implementation storage order. If so, then tuples may be moved around between the two sets without any consequence based on the order. If the relation values have a different ordering to the tuples, then it is necessary to reorder one of the tuples to make its type match that of the another one.

We divide this complication into two parts.

- Determine if reordering is necessary.
- Perform the reordering of the tuple attributes.

```

<<relation private functions>>=
/// The `tupleNeedsReordering` function determines if the implementation
/// storage order of two `RelTuple` types is the same. It returns `true`
/// if the memory order is different, i.e. one of the tuples would have
/// to be reordered to make the types the same, and `false` if the
/// orders are the same and the reordering operation is not required.
fn tupleNeedsReordering(
    comptime OneTupleType: type,
    comptime OtherTupleType: type,
) bool {
    if (!isRelTupleType(OneTupleType))
        @compileError("one argument is not a tuple, found '" ++
                     @typeName(OneTupleType) ++ "'");
    if (!isRelTupleType(OtherTupleType))
        @compileError("other argument is not a tuple, found '" ++
                     @typeName(OtherTupleType) ++ "'");
    if (!equivalentHeading(OneTupleType.TupleBase, OtherTupleType.TupleBase))
        @compileError("tuple headings are not equivalent");
}

```

```

    return OneTupleType.TupleBase != OtherTupleType.TupleBase; ①
}

```

- ① It is the case that a tuple needs to be reordered even though it has an equivalent heading. The fact that the headings are equivalent means the names and types are the same. However, the physical order of attributes in memory is only guaranteed if the types are the same. Sometimes, the order is the same but the two types are different. This can result because of the names given to anonymous structures. Reordering in that case amount to casting one tuple type to a new tuple type. Note that `@bitCast()` does not work on structures that don't have a defined memory layout.

```

<<relation private functions>>=
/// The `reorderTuple` function returns a `RelTuple` value of type, `TupleType`,
/// whose attribute values are the same as those from `reorder_tuple`.
/// This function does not modify the attribute values only the memory order
/// of the implementation is affected.
fn reorderTuple(
    comptime TupleType: type,
    reorder_tuple: anytype,
) TupleType {
    const ReorderTupleType = @TypeOf(reorder_tuple);           ①
    var reordered: TupleType = undefined;
    const reorder_field_names = comptime meta.fieldNames(ReorderTupleType.TupleBase);
    inline for (reorder_field_names) |reorder_field_name| {
        @field(reordered.reltuple, reorder_field_name) =
            @field(reorder_tuple.reltuple, reorder_field_name);
    }
    return reordered;
}

```

- ① There is no need for type checks here. The `reorderTuple` function is only called after first invoking `tupleNeedsReordering` and all the relevant type checks are done there.

With these two functions in hand, the `eql` function is shown below.

```

<<relation public functions>>=
/// The `eql` function returns `true` if the `one` relation value is equal
/// to the `other` relation value. Equality is determined by the two relation
/// values having equal headings and containing the same set of tuples.
pub fn eql(
    one: Self,
    other: anytype,
) bool {
    const OtherRelType = @TypeOf(other);
    const needs_reordering = comptime tupleNeedsReordering(Tuple, OtherRelType.Tuple);

    if (one.cardinality() != other.cardinality()) return false; ①
}

```

```

    if (needs_reordering) {
        for (one.body()) |one_tuple| {
            const reordered = reorderTuple(OtherRelType.Tuple, one_tuple);
            if (!other.contains(reordered)) return false;
        }
    } else {
        for (one.body()) |one_tuple| {
            if (!other.contains(one_tuple)) return false;
        }
    }

    return true;
}

```

- ① Different sized sets can never be equal.

```

<<relvalue module tests>>=
test "relation value equality" {
    relExprBegin(); defer relExprEnd();

    const test_relation = create_test_relation();
    try testing.expect(test_relation.eql(test_relation));

    // Construct an equivalent relation but with different heading order.
    // Checking if reordering works properly.
    const OtherTestBase = struct {
        value: usize,
        name: []const u8,
    };
    const OtherTestRelation = RelValue(OtherTestBase);
    const other_relation = OtherTestRelation.create(&.{.
        .{ .name = "foo", .value = 0 },
        .{ .name = "foo", .value = 1 },
        .{ .name = "foo", .value = 2 },
        .{ .name = "bar", .value = 0 },
        .{ .name = "bar", .value = 1 },
        .{ .name = "bar", .value = 2 },
    });
    try testing.expect(test_relation.eql(other_relation));

    // Build a relation value with the same heading but different values
    // for one attribute.
    // Check that differences are detected.
    const AnotherTestRelation = RelValue(OtherTestBase);
    const another_relation = AnotherTestRelation.create(&.{.
        .{ .name = "foo", .value = 0 },
        .{ .name = "foo", .value = 1 },
        .{ .name = "foo", .value = 2 },
    });
}

```

```

        .{ .name = "bar", .value = 0 },
        .{ .name = "bar", .value = 1 },
        .{ .name = "bar", .value = 3 }, // different tuple
    });

    try testing.expect(!test_relation.eql(another_relation));
}

```

Supersets

```

<<relation public functions>>=
/// The `superSetOf` function returns `true` if the set of tuples of the
/// `one` relation value is a superset of the tuples in the `other` relation value
/// and `false` otherwise.
pub fn superSetOf(
    one: Self,
    other: anytype,
) bool {
    if (one.cardinality() < other.cardinality()) return false; ①

    const OtherRelType = @TypeOf(other);
    const needs_reordering = comptime tupleNeedsReordering(Tuple, OtherRelType.Tuple);

    if (needs_reordering) {
        for (other.body()) |other_tuple| {
            const reordered = reorderTuple(Tuple, other_tuple);
            if (!one.contains(reordered)) return false;
        }
    } else {
        for (other.body()) |other_tuple| {
            if (!one.contains(other_tuple)) return false;
        }
    }

    return true;
}

```

① One can never be a superset of other if it has fewer tuples.

```

<<relvalue module tests>>=
test "relation value superset" {
    relExprBegin(); defer relExprEnd();

    const test_relation = create_test_relation();
    try testing.expect(test_relation.superSetOf(test_relation));

    const OtherTestBase = struct {
        value: usize,

```

```

        name: []const u8,
    };
    const OtherTestRelation = RelValue(OtherTestBase);
    const other_relation = OtherTestRelation.create(&.{.
        .{ .name = "foo", .value = 0 },
        .{ .name = "foo", .value = 1 },
        .{ .name = "foo", .value = 2 },
        .{ .name = "bar", .value = 0 },
        .{ .name = "bar", .value = 1 },
        .{ .name = "bar", .value = 2 },
    });
}

try testing.expect(test_relation.superSetOf(other_relation));

const AnotherTestRelation = RelValue(OtherTestBase);
const another_relation = AnotherTestRelation.create(&.{.
    .{ .name = "foo", .value = 0 },
    .{ .name = "foo", .value = 1 },
    .{ .name = "foo", .value = 2 },
    .{ .name = "bar", .value = 0 },
    .{ .name = "bar", .value = 1 },
    .{ .name = "bar", .value = 2 },
    .{ .name = "bar", .value = 3 }, // additional tuple
});
}

try testing.expect(!test_relation.superSetOf(another_relation));
}

```

```

<<relation public functions>>=
/// The `properSuperSetOf` function returns `true` if the `one` relation value
/// is a proper superset of the `other` relation value and `false` otherwise.
pub fn properSuperSetOf(
    one: Self,
    other: anytype,
) bool {
    if (one.cardinality() <= other.cardinality()) return false;

    return one.superSetOf(other);
}

```

```

<<relvalue module tests>>=
test "relation value proper superset" {
    relExprBegin(); defer relExprEnd();

    const test_relation = create_test_relation();
    // A set is always a superset of itself but never a proper superset of itself.
    try testing.expect(!test_relation.properSuperSetOf(test_relation));
}

```

```

const OtherTestBase = struct {
    value: usize,
    name: []const u8,
};

const OtherTestRelation = RelValue(OtherTestBase);
const other_relation = OtherTestRelation.create(&.{.
    .{ .name = "foo", .value = 0 },
    .{ .name = "foo", .value = 1 },
    .{ .name = "foo", .value = 2 },
    .{ .name = "bar", .value = 0 },
    .{ .name = "bar", .value = 1 },
    // missing: .name = "bar", .value = 2
});

try testing.expect(test_relation.properSuperSetOf(other_relation));
}

```

Subsets

```

<<relation public functions>>=
/// The `subSetOf` function returns `true` if the `one` relation value is a
/// subset of the `other` relation value and `false` otherwise.
pub fn subSetOf(
    one: Self,
    other: anytype,
) bool {
    if (one.cardinality() > other.cardinality()) return false; ①

    const OtherRelType = @TypeOf(other);
    const needs_reordering = comptime tupleNeedsReordering(Tuple, OtherRelType.Tuple);

    if (needs_reordering) {
        for (one.body()) |one_tuple| {
            const reordered = reorderTuple(OtherRelType.Tuple, one_tuple);
            if (!other.contains(reordered)) return false;
        }
    } else {
        for (one.body()) |one_tuple| {
            if (!other.contains(one_tuple)) return false;
        }
    }

    return true;
}

```

① A set with more elements cannot be a subset of the other set.

```

<<relvalue module tests>>=
test "relation value subset" {
    relExprBegin(); defer relExprEnd();

    const test_relation = create_test_relation();
    // A set is always a subset of itself.
    try testing.expect(test_relation.subSetOf(test_relation));
    try testing.expect(TestRelation.createEmpty().subSetOf(test_relation));

    const OtherTestBase = struct {
        value: usize,
        name: []const u8,
    };
    const OtherTestRelation = RelValue(OtherTestBase);
    const other_relation = OtherTestRelation.create(&.{.
        .{ .name = "foo", .value = 0 },
        .{ .name = "foo", .value = 1 },
        .{ .name = "foo", .value = 2 },
        .{ .name = "bar", .value = 0 },
        .{ .name = "bar", .value = 1 },
        .{ .name = "bar", .value = 2 },
    });
    try testing.expect(test_relation.subSetOf(other_relation));
}

```

```

<<relation public functions>>=
/// The `properSubSetOf` function returns `true` if the `one` relation value is a
/// proper subset of the `other` relation value and `false` otherwise.
pub fn properSubSetOf(
    one: Self,
    other: anytype,
) bool {
    if (one.cardinality() >= other.cardinality()) return false; ①

    return one.subSetOf(other);
}

```

① A proper subset of another set can never contain the same or more elements.

```

<<relvalue module tests>>=
test "relation value proper subset" {
    relExprBegin(); defer relExprEnd();

    const test_relation = create_test_relation();
    // A set is not a proper subset of itself.
    try testing.expect(!test_relation.properSubSetOf(test_relation));
}

```

```

const OtherTestBase = struct {
    value: usize,
    name: []const u8,
};

const OtherTestRelation = RelValue(OtherTestBase);
const other_relation = OtherTestRelation.create(&.<
    .{ .name = "foo", .value = 0 },
    .{ .name = "foo", .value = 1 },
    .{ .name = "bar", .value = 0 },
    .{ .name = "bar", .value = 1 },
    .{ .name = "bar", .value = 2 },
});

try testing.expect(other_relation.properSubSetOf(test_relation));
}

```

Aggregate Operations

It is typical to provide functions to compute values based on all the values of a particular attribute. Technically, this is not part of the relational algebra itself, but provides needed computations that would otherwise require explicit iteration in the application code. These functions are only defined for numeric attribute types. The following operations are provided:

sum

Compute the sum of all the values for a given attribute. The sum of an empty relation is zero.

average

Compute the average of the values for a given attribute. The average of an empty relation is undefined since division by zero is undefined.

max

Compute the maximum of the values for a given attribute. The maximum of an empty relation is the minimum value of the attribute's data type.

min

Compute the minimum of the values for a given attribute. The minimum of an empty relation is the maximum value of the attribute's data type.

Note a “count” operation is not provided, cardinality serves that purpose.

These function operate on an *accumulator* semantic. You can think of the functions as starting with an initial value and then the operation is performed on the initial value accumulating the result. This approach allows for empty relations by simply returning the initial value given to the accumulator.

Sum

```

<<relation public functions>>=
/// The `sum` function computes the sum of the values of the attribute
/// given by, `attr_id`. The function is only defined for attribute
/// types for which addition is defined.
pub fn sum(
    self: *const Self,

```

```

    comptime attr_id: AttributeId,
) AttributeType(attr_id) {
    const SumType = AttributeType(attr_id);

    var result_sum: SumType = @as(SumType, 0);
    for (self.body()) |self_tuple| {
        result_sum += self_tuple.extract(attr_id);
    }

    return result_sum;
}

```

Average

```

<<relation public functions>>=
/// The `average` function computes the sum of the values of the
/// attribute given by, `attr_id` divided by the cardinality of
/// the given relation. This function is not defined for empty relations.
pub fn average(
    self: *const Self,
    comptime attr_id: AttributeId,
) AttributeType(attr_id) {
    if (self.cardinality() == 0)
        panic("attempt average an empty relation value\n", .{});

    const SumType = AttributeType(attr_id);

    const result_div: SumType = switch (@typeInfo(SumType)) {
        .int => iblk: {
            const item_count: SumType = @intCast(self.cardinality());
            break :iblk item_count;
        },
        .float => fblk: {
            const item_count: SumType = @floatFromInt(self.cardinality());
            break :fblk item_count;
        },
        else => @compileError("attribute '" ++ @tagName(attr_id) ++
            "' is not a numeric type"),
    };

    return self.sum(attr_id) / result_div;
}

```

Maximum

```

<<relation public functions>>=
/// The `max` function computes the maximum value of any of the attributes
/// given by `attr_id`. The maximum of an empty relation is the minimum
/// value of the attribute's data type.
pub fn max(
    self: *const Self,

```

```

    comptime attr_id: AttributeId,
) AttributeType(attr_id) {
    const MaxType = AttributeType(attr_id);

    var result_max: MaxType = switch (@typeInfo(MaxType)) {
        .int => math.minInt(MaxType),
        .float => -math.inf(MaxType),
        else => @compileError("attribute '" ++ @tagName(attr_id) ++
            "' does not support the max aggregate operation"),
    };

    for (self.body()) |self_tuple| {
        result_max = @max(result_max, self_tuple.extract(attr_id));
    }

    return result_max;
}

```

Minimum

```

<<relation public functions>>=
/// The `min` function computes the minimum value of any of the attributes
/// given by `attr_id`. The minimum of an empty relation is the maximum
/// value of the attribute's data type.
pub fn min(
    self: *const Self,
    comptime attr_id: AttributeId,
) AttributeType(attr_id) {
    const MinType = AttributeType(attr_id);

    var result_min: MinType = switch (@typeInfo(MinType)) {
        .int => math.maxInt(MinType),
        .float => math.inf(MinType),
        else => @compileError("attribute '" ++ @tagName(attr_id) ++
            "' does not support the min aggregate operation"),
    };

    for (self.body()) |self_tuple| {
        result_min = @min(result_min, self_tuple.extract(attr_id));
    }

    return result_min;
}

```

Relation Set Operations

Relational algebra borrows operations from sets such as union and intersection. However, in relational algebra we must place additional constraints on the nature of the operands of the set operation. Relation set operations are defined only for relation values that have equal headings, *i.e.* the same set of attributes (including equivalence of the corresponding data types). This condition is usually referred to as, *union-compatible*. It is analogous to the compatibility requirements for matrix

multiplication.

The Cartesian product is also defined differently than in set theory. Set theory defines it using the notion of ordered pairs. Relational theory does not use ordered pairs and defines an extended Cartesian product as yielding a set of tuples having the union of the headings of the arguments. This algebra implementation does not define a Cartesian product operation. Rather, the `join` operation described below produces the Cartesian product when the attributes of the argument tuple headings are disjoint.

Union

The classic set operation is union. It is the relational analog of Boolean *OR*. The returned relation value contains tuples that are present in either of the operands. Note the name here is `unionRelValue` since `union` is Zig keyword. This was deemed easier to write than `@"union"`, however the `@"union"` synonym is also given.

```
<<relation public functions>>
/// The `unionRelValue` function creates a new relation value that contains
/// the tuples that are present in the `one` `RelValue` or the `other`
/// `RelValue` or both. The resulting relation value is a guaranteed
/// to be a set and duplicated tuples are omitted from the result.
/// The `unionRelValue` function is commutative and associative and distributes
/// over intersection.
pub fn unionRelValue(
    one: Self,
    other: anytype,
) Self {
    // Since `tupleNeedsReordering` checks its inputs arguments, we place
    // it at the beginning of the function.
    const OtherRelType = @TypeOf(other);
    const needs_reordering = comptime tupleNeedsReordering(Tuple, OtherRelType.Tuple);

    // Worst case size is that all tuples of both relation values end up in the union,
    // i.e. the two relations are disjoint sets.
    const union_capacity = one.cardinality() + other.cardinality();
    var union_rel = one.cloneCapacity(union_capacity); ①

    if (needs_reordering) {
        for (other.body()) |other_tuple| {
            const reordered = reorderTuple(Tuple, other_tuple);
            _ = union_rel.insertTuple(reordered);
        }
    } else {
        for (other.body()) |other_tuple| {
            // Duplicate tuples are silently ignored.
            _ = union_rel.insertTuple(other_tuple);
        }
    }

    return union_rel;
}
```

```

pub inline fn @"union"(  

    one: Self,  

    other: anytype,  

) Self {  

    return one.unionRelValue(other);  

}

```

- ① We start with all the tuples from the `one` value and then discard any duplicates that appear in the `other` value.

Tests

```

<<relvalue module tests>>=  

test "union relation value" {  

    relExprBegin(); defer relExprEnd();  
  

    const test_relation = create_test_relation();  
  

    // The union of a set with itself always yields the given set,  

    // i.e. A union A == A.  

    var union_relation = test_relation.unionRelValue(test_relation);  

    try testing.expect(union_relation.cardinality() == 6);  

    try testing.expect(union_relation.eql(test_relation));  
  

    // Test where reordering is necessary.  

    const OtherTestBase = struct {  

        value: usize,  

        name: []const u8,  

    };  

    const OtherTestRelation = RelValue(OtherTestBase);  
  

    const other_relation = OtherTestRelation.create(&.{  

        .{ .value = 0, .name = "baz" },  

        .{ .value = 1, .name = "baz" },  

        .{ .value = 2, .name = "baz" },  

        .{ .value = 2, .name = "bar" },      // common tuple
    });
    const other_union = test_relation.unionRelValue(other_relation);  
  

    // 9 ==> 6 from test_relation, 3 from other_relation,  

    // common tuple appears only once.  

    try testing.expect(other_union.cardinality() == 9);  
  

    const AnotherTestRelation = RelValue(OtherTestBase);  

    const another_relation = AnotherTestRelation.create(&.{  

        .{ .name = "baz", .value = 0 },  

        .{ .name = "baz", .value = 1 },  

        .{ .name = "baz", .value = 2 },
    });
}

```

```

// Union is associative, i.e. (A union B) union C == A union (B union C).
const lhs = test_relation.unionRelValue(other_relation).unionRelValue(
another_relation);
const rhs = test_relation.unionRelValue(other_relation.unionRelValue(
another_relation));
try testing.expect(lhs.eql(rhs));
}

```

Intersection

Relation value intersection is the relational analog of Boolean *AND*. The result contains only those tuples that are found in both relation values.

```

<<relation public functions>>=
/// The `intersect` function returns a relation value that contains all
/// the tuples that are present in both the `one` relation value and in the
/// `other` relation value. The `intersect` function is commutative and associative
/// and it distributes over the union.
pub fn intersect(
    one: Self,
    other: anytype,
) Self {
    const OtherRelType = @TypeOf(other);
    const needs_reordering = comptime tupleNeedsReordering(Tuple, OtherRelType.Tuple);

    // Worst case is the maximum number of tuples from one of the relations,
    // i.e. the given relation value is a subset of the other or vice versa.
    const intersect_capacity = @max(one.cardinality(), other.cardinality());
    var intersect_rel = createCapacity(intersect_capacity);

    if (needs_reordering) {
        for (one.body()) |self_tuple| {
            const reordered = reorderTuple(OtherRelType.Tuple, self_tuple);
            if (other.contains(reordered)) {
                const inserted = intersect_rel.insertTuple(self_tuple);
                assert(inserted);
            }
        }
    } else {
        for (one.body()) |self_tuple| {
            if (other.contains(self_tuple)) {
                const inserted = intersect_rel.insertTuple(self_tuple);
                assert(inserted);
            }
        }
    }

    return intersect_rel;
}

```

Tests

```
<<relvalue module tests>>=
test "intersect relation value" {
    relExprBegin(); defer relExprEnd();

    const test_relation = create_test_relation();

    // Like the union, the intersection of a set with itself just
    // yields the original set.
    var intersect_relation = test_relation.intersect(test_relation);

    try testing.expect(intersect_relation.cardinality() == 6);

    const OtherTestBase = struct {
        value: usize,
        name: []const u8,
    };
    const OtherTestRelation = RelValue(OtherTestBase);

    const other_relation = OtherTestRelation.create(&.{.
        .{ .value = 0, .name = "baz" },
        .{ .value = 1, .name = "baz" },
        .{ .value = 2, .name = "baz" },
        .{ .value = 2, .name = "bar" },      // only tuple in common
    });

    const other_intersect = test_relation.intersect(other_relation);
    try testing.expect(other_intersect.cardinality() == 1);
}
```

Minus

Set difference is another fundamental operation.

```
<<relation public functions>>=
/// The `minus` operation returns a relation value that contains all the
/// tuples of the `minuend` that are *not* also contained in the
/// `subtrahend`. The `minus` operation is *not* commutative.
pub fn minus(
    minuend: Self,
    subtrahend: anytype,
) Self {
    const OtherRelType = @TypeOf(subtrahend);
    const needs_reordering = comptime tupleNeedsReordering(Tuple, OtherRelType.Tuple);

    // Worst case is the subtrahend is empty.
    const capacity = minuend.cardinality();
```

```

var difference = createCapacity(capacity);

if (needs_reordering) {
    for (minuend.body()) |minuend_tuple| {
        const reordered = reorderTuple(OtherRelType.Tuple, minuend_tuple);
        if (!subtrahend.contains(reordered)) {
            const inserted = difference.insertTuple(minuend_tuple);
            assert(inserted);
        }
    }
} else {
    for (minuend.body()) |minuend_tuple| {
        if (!subtrahend.contains(minuend_tuple)) {
            const inserted = difference.insertTuple(minuend_tuple);
            assert(inserted);
        }
    }
}

return difference;
}

```

Tests

```

<<relvalue module tests>>=
test "minus relation value" {
    relExprBegin(); defer relExprEnd();

    const test_relation = create_test_relation();

    const diff_relation = test_relation.minus(test_relation);
    // Subtracting from yourself leaves nothing.
    try testing.expect(diff_relation.empty());

    const OtherTestBase = struct {
        value: usize,
        name: []const u8,
    };
    const OtherTestRelation = RelValue(OtherTestBase);

    const other_relation = OtherTestRelation.create(&.{.
        .{ .value = 0, .name = "baz" },
        .{ .value = 1, .name = "baz" },
        .{ .value = 2, .name = "baz" },
        .{ .value = 2, .name = "bar" },      // one tuple in common
    });

    const other_diff = test_relation.minus(other_relation);
    // Common tuple is now removed.
    try testing.expectEqual(@as(usize, 5), other_diff.cardinality());
}

```

```
}
```

Symmetric Difference

Since `minus` is not commutative, the symmetric difference is defined as the union of the two possible set differences, i.e. $(A \text{ minus } B) \cup (B \text{ minus } A)$. It is also true that the symmetric difference is equal to the union of the two relation values minus the intersection, i.e. $(A \cup B) \text{ minus } (A \cap B)$. The symmetric difference is the relational analog of Boolean *exclusive OR*. The tuples of the result are those contained in either relation but not both. This operation is sometimes called the *disjunctive union* because the result is the disjoint set of the relation values. Where the disjoint union requires the two relation values to be disjoint before performing the union, the symmetric difference forces the result to be the disjoint union by discarding any common tuples.

```
<<relation public functions>>=
/// The `symmetricDifference` function returns a relation value that
/// is the symmetric difference of the `one` and `other` relation
/// values. The result contains all the tuples in the difference between
/// the `one` relation value and the `other` relation value union with
/// the difference between `other_relation` and the `one` relation value.
/// Concisely, the result is:
/// (`one` minus `other`) union (`other` minus `one`).
pub fn symmetricDifference(
    one: Self,
    other: anytype,
) Self {
    const OtherRelType = @TypeOf(other);
    const needs_reordering = comptime tupleNeedsReordering(Tuple, OtherRelType.Tuple);

    // Worst case is the sets are disjoint and the result degenerates to
    // the union.
    const symdiff_capacity = one.cardinality() + other.cardinality();
    var symdiff_rel = createCapacity(symdiff_capacity);

    if (needs_reordering) {
        for (one.body()) |one_tuple| {
            const reordered = reorderTuple(OtherRelType.Tuple, one_tuple);
            if (!other.contains(reordered)) {
                const inserted = symdiff_rel.insertTuple(one_tuple);
                assert(inserted);
            }
        }
        for (other.body()) |other_tuple| {
            const reordered = reorderTuple(Tuple, other_tuple);
            if (!one.contains(reordered)) {
                const inserted = symdiff_rel.insertTuple(reordered);
                assert(inserted);
            }
        }
    } else {
}
```

```

        for (one.body()) |one_tuple| {
            if (!other.contains(one_tuple)) {
                const inserted = symdiff_rel.insertTuple(one_tuple);
                assert(inserted);
            }
        }
        for (other.body()) |other_tuple| {
            if (!one.contains(other_tuple)) {
                const inserted = symdiff_rel.insertTuple(other_tuple);
                assert(inserted);
            }
        }
    }

    return symdiff_rel;
}

```

Tests

```

<<relvalue module tests>>=
test "symmetric difference relation value" {
    relExprBegin(); defer relExprEnd();

    const test_relation = create_test_relation();

    const symdiff_relation = test_relation.symmetricDifference(test_relation);
    // Symmetric difference with yourself leaves nothing.
    try testing.expect(symdiff_relation.cardinality() == 0);

    const OtherTestBase = struct {
        value: usize,
        name: []const u8,
    };
    const OtherTestRelation = RelValue(OtherTestBase);

    const other_relation = OtherTestRelation.create(&.{.
        .{ .value = 0, .name = "baz" },
        .{ .value = 1, .name = "baz" },
        .{ .value = 2, .name = "baz" },
        .{ .value = 2, .name = "bar" },      // only tuple in common
    });

    const symdiff_other = test_relation.symmetricDifference(other_relation);
    // Common tuple is removed leaving 5 from test_relation plus 3 from other_relation.
    try testing.expectEqual(@as(usize, 8), symdiff_other.cardinality());

    // To show that the symmetric difference gives a disjoint union,
    // separate the sets and show that the disjoint union of the two
    // separated sets equals the symmetric difference.
    const sep_one = symdiff_other.restrictAtLeastOneEqual(.{

```

```

    .{ .name, "foo" },
    .{ .name, "bar" },
);
const sep_other = symdiff_other.restrictEqual(.name, "baz");
const disjoint_of_seps = try sep_one.disjointUnion(sep_other);
try testing.expect(symdiff_other.eql(disjoint_of_seps));
}

```

Disjoint Union

The disjoint union is similar to union except that the two relation operands must be disjoint sets. Any attempt to compute the disjoint union of two relation values that have tuples in common results in an error. Note that this behavior has “insert” semantics, *i.e.* it is an error to attempt to insert a duplicate tuple. The result relation value appears to have the other relation value inserted into it and fails on any attempt to insert a duplicate.

```

<<relation public functions>>=
/// The `disjointUnion` function returns a new relation value that contains
/// all the tuples from the `one` relation value and all the tuples from
/// the `other`. It is an error to attempt to compute the disjoint
/// union from relation values whose tuple sets contain common tuples.
/// The `disjointUnion` function is commutative and associative.
pub fn disjointUnion(
    one: Self,
    other: anytype,
) RelValueError!Self {
    const OtherRelType = @TypeOf(other);
    const needs_reordering = comptime tupleNeedsReordering(Tuple, OtherRelType.Tuple);

    // Since this operation insists that the relation values be disjoint,
    // the disjoint union must have the sum of the tuples in both relation values.
    const union_capacity = one.cardinality() + other.cardinality();
    var union_rel = one.cloneCapacity(union_capacity);
    errdefer union_rel.destroy();

    if (needs_reordering) {
        for (other.body()) |other_tuple| {
            const reordered = reorderTuple(Tuple, other_tuple);
            const inserted = union_rel.insertTuple(reordered);
            if (!inserted) return RelValueError.DuplicateTuple;
        }
    } else {
        for (other.body()) |other_tuple| {
            const inserted = union_rel.insertTuple(other_tuple);
            if (!inserted) return RelValueError.DuplicateTuple;
        }
    }

    return union_rel;
}

```

Tests

```
<<relvalue module tests>>=
test "disjoint union relation value" {
    relExprBegin(); defer relExprEnd();

    const test_relation = create_test_relation();

    const OtherTestBase = struct {
        value: usize,
        name: []const u8,
    };
    const OtherTestRelation = RelValue(OtherTestBase);

    const other_relation = OtherTestRelation.create(&.{.
        .{ .value = 0, .name = "baz" },
        .{ .value = 1, .name = "baz" },
        .{ .value = 2, .name = "baz" },
        .{ .value = 2, .name = "bar" },      // is a duplicate
    });
}

try testing.expectError(
    RelValueError.DuplicateTuple,
    test_relation.disjointUnion(other_relation),
);
}
```

Projection

Projection is a fundamental operation in relational algebra and computes a relation value containing the specified subset of attributes. Projection can be viewed as a vertical partitioning of the relation value.

```
<<relvalue module declarations>>=
const ProjectHeading = reltuple.ProjectHeading;
```

```
<<relation public functions>>=
/// The `project` function returns a new relation value whose heading
/// contains only those attributes given in the `proj_attrs` slice.
/// Note that because the resulting relation value is a set, the
/// cardinality of the projection result may be less than the cardinality
/// of the original relation value. This happens when the projected
/// attributes happen to have the same value in multiple tuples.
pub fn project(
    self: *const Self,
    comptime proj_attrs: []const AttributeId,
) RelValue(ProjectHeading(TupleBase, proj_attrs)) {
```

```

const Projection = RelValue(ProjectHeading(TupleBase, proj_attrs));

// Worst case is that all the projected attribute values form a set,
// i.e. the relation value can be identified by the projected attributes.
var projection = Projection.createCapacity(self.cardinality());
for (self.body()) |self_tuple| {
    const projected = self_tuple.project(proj_attrs);
    // Projected tuples do not guarantee a set, so duplicates are ignored.
    _ = projection.insertTuple(projected);
}

return projection;
}

```

Tests

```

<<relvalue module tests>>=
test "project relation value" {
    relExprBegin(); defer relExprEnd();

    const test_relation = create_test_relation();

    const value_relation = test_relation.project(&{.value});
    // There are 3 distinct values of the `value` attribute.
    try testing.expectEqual(@as(usize, 3), value_relation.cardinality());

    const name_relation = test_relation.project(&{.name});
    // There are 2 distinct values of the `name` attribute.
    try testing.expectEqual(@as(usize, 2), name_relation.cardinality());

    // The nullary projection of a non-empty relation value yields DEE.
    const nullary = test_relation.project(&{});
    try testing.expect(nullary.eql(dee()));

    // The nullary projection of an empty relation value yields DUM.
    const empty = TestRelation.createEmpty();
    const nullary_of_empty = empty.project(&{});
    try testing.expect(nullary_of_empty.eql(dum()));
}

```

Restriction

Restriction, often called “selection”, produces a new relation value that contains a subset of the operand’s tuples. Using the term, restriction, connotes a mental image of throwing out some of the tuples, where selection connotes an image of picking out favored tuples. Whichever way you think about the operation, it is fundamental and can be viewed as a horizontal partitioning of the relation value.

Restriction operates by using a “selector” function which evaluates to a Boolean result and determines if a tuple is placed in the result. This function takes a tuple value and any other

provided arguments and returns a Boolean value. If the selector returns `true`, the tuple is included in the restricted result. Otherwise, the tuple is not placed in the result.

Callers may supply any function matching the selector interface. Below, selector functions are supplied for some common cases.

```
<<relation public functions>>=
/// The `restrict` function returns a relation value that contains a
/// subset of the given relation value. Which tuples are included is
/// determined by the function pointed to by `selector`. The `selector`
/// function is invoked for each tuple in the relation value and is
/// passed the `selector_args` argument. An invocation of `selector`
/// that returns `true` for some tuple causes that tuple to be included
/// in the resulting relation value.
pub fn restrict(
    self: *const Self,
    selector: *const fn(tuple: Tuple, args: anytype) bool,
    selector_args: anytype,
) Self {
    // Worst case is that the restrictor always returns true.
    var restriction = createCapacity(self.cardinality());

    for (self.body()) |self_tuple| {
        if (selector(self_tuple, selector_args)) {
            const inserted = restriction.insertTuple(self_tuple);
            assert(inserted);
        }
    }

    return restriction;
}
```

The following functions satisfy the selector interface for common use cases.

```
<<relvalue module public functions>>=
/// The `identitySelector` selector function always returns `true`.
pub fn identitySelector(
    tuple: anytype,
    args: anytype,
) bool {
    _ = tuple;
    _ = args;
    return true;
}
```

```

<<relvalue module public functions>>=
/// The `annihilateSelector` selector function always returns `false`.
pub fn annihilateSelector(
    tuple: anytype,
    args: anytype,
) bool {
    _ = tuple;
    _ = args;
    return false;
}

```

```

<<relvalue module public functions>>=
pub fn equalitySelector(
/// The `equalitySelector` selector function returns whether a given
/// attribute is equal to a given value. The `av_pair` argument is
/// expected to be an attribute id / attribute value pair.
    tuple: anytype,
    av_pair: anytype,
) bool {
    assert(av_pair.len == 2);
    return attributeEql(tuple.extract(av_pair.@"0"), av_pair.@"1");
}

```

```

<<relvalue module public functions>>=
/// The `conjunctiveEqualitySelector` selector function returns the result
/// of computing the logical *AND* of the equality between multiple
/// attributes and values. The `av_pairs` argument is expected to be a
/// list of attribute id / attribute value pairs. The returned result is
/// `true` if all the attribute id / value pairs in `av_pairs` are equal.
pub fn conjunctiveEqualitySelector(
    tuple: anytype,
    av_pairs: anytype,
) bool {
    var match: bool = true;
    inline for (av_pairs) |av_pair| {
        match = match and equalitySelector(tuple, av_pair);
    }
    return match;
}

```

```

<<relvalue module public functions>>=
/// The `disjunctiveEqualitySelector` selector function returns the
/// result of computing the logical *OR* of the equality between multiple
/// attributes and values. The `av_pairs` argument is expected to be a

```

```

/// list of attribute id / attribute value pairs. The returned result is
/// `true` if at least one of the attribute id / value pairs in `av_pairs`
/// is equal.
pub fn disjunctiveEqualitySelector(
    tuple: anytype,
    av_pairs: anytype,
) bool {
    var match: bool = false;
    inline for (av_pairs) |av_pair| {
        match = match or equalitySelector(tuple, av_pair);
    }
    return match;
}

```

The following three functions provide convenient interfaces to using the previous selector functions. Essentially, these function “curry” the `restrict` operation to use the appropriate selector function.

```

<<relation public functions>>=
/// The `restrictEqual` function returns a new relation value that contains those
/// tuples where the value of the `attr_id` attribute is equal to the
/// value of the `attr_value` argument.
pub fn restrictEqual(
    self: *const Self,
    comptime attr_id: AttributeId,
    attr_value: AttributeType(attr_id),
) Self {
    return self.restrict(equalitySelector, .{ attr_id, attr_value });
}

```

```

<<relation public functions>>=
/// The `restrictAllEqual` function returns a new relation value that contains those
/// tuples where all of the attribute id / attribute value pairs of the `av_pairs`
/// argument are true. The `av_pairs` argument is assumed to be a list of
/// attribute id / value pairs.
pub fn restrictAllEqual(
    self: *const Self,
    av_pairs: anytype,
) Self {
    return self.restrict(conjunctiveEqualitySelector, av_pairs);
}

```

```

<<relation public functions>>=
/// The `restrictAtLeastOneEqual` function returns a new relation value that contains
/// those tuples where at least one of the attribute id / attribute value pairs of

```

```

/// the `av_pairs` argument are true. The `av_pairs` argument is assumed to be a list
/// of attribute id / value pairs.
pub fn restrictAtLeastOneEqual(
    self: *const Self,
    av_pairs: anytype,
) Self {
    return self.restrict(disjunctiveEqualitySelector, av_pairs);
}

```

The other common usage of restriction is to find tuples that match some comparison operation. Note that all attributes can be evaluated for equality, but some attribute types do not support the usual comparison operators such as less than or greater than. Note also that comparison of strings is a complicated subject. The comparison for Zig strings here is a simplistic comparison of the constituent memory bytes. Note also that the range of possible string operations is quite large. For example, there are no functions provided that deal with pattern or regular expression matching, which can be useful in some contexts.

```

<<relvalue module public functions>>=
/// The `comparativeSelector` selector function compares the value of an attribute
/// of `tuple` to a given value.
/// The `args` arguments is taken as a three element list:
///
/// 1. The attribute id of the attribute which participates in the comparison.
/// 2. An enum literal of type `std.math.CompareOperator`.
/// 3. The value against which the comparison is made.
///
/// The function returns the Boolean result of the comparison.
pub fn comparativeSelector(
    tuple: anytype,
    args: anytype,
) bool {
    assert(args.len == 3);
    return math.compare(tuple.extract(args@"0"), args@"1", args@"2"); ①
}

```

① Ultimately, we can compare those things that `math.compare` can handle.

```

<<relvalue module public functions>>=
/// The `conjunctiveComparativeSelector` selector function returns the result
/// of computing the logical *AND* of the comparisons between multiple
/// attributes and values. The `exprs` argument is expected to be a
/// list of triples. The triples are taken to be as described in the
/// `comparativeSelector` function. The returned result is
/// `true` if all comparison given by `exprs` are true.
pub fn conjunctiveComparativeSelector(
    tuple: anytype,
    exprs: anytype,
) bool {

```

```

var match: bool = true;
inline for (exprs) |expr| {
    match = match and comparativeSelector(tuple, expr);
}
return match;
}

```

```

<<relvalue module public functions>>=
/// The `disjunctiveComparativeSelector` selector function returns the
/// result of computing the logical *OR* of the comparisons between multiple
/// attributes and values. The `exprs` argument is expected to be a
/// list triples. The triples are taken to be as described in the
/// `comparativeSelector` function. The returned result is
/// `true` if at least one of the comparisons given by `exprs` is true.
/// is `true`.
pub fn disjunctiveComparativeSelector(
    tuple: anytype,
    exprs: anytype,
) bool {
    var match: bool = false;
    inline for (exprs) |expr| {
        match = match or comparativeSelector(tuple, expr);
    }
    return match;
}

```

Following the same pattern as with equality selector functions, we provide simpler interfaces which make using the comparative selector function convenient.

```

<<relation public functions>>=
/// The `restrictComparison` function returns a new relation value where
/// the value of the attribute given by `attr_id` when compared to
/// the `attr_value` by the `op` operation is true.
pub fn restrictComparison(
    self: *const Self,
    comptime attr_id: AttributeId,
    op: math.CompareOperator,
    attr_value: AttributeType(attr_id),
) Self {
    return self.restrict(comparativeSelector, .{ attr_id, op, attr_value });
}

```

```

<<relation public functions>>=
/// The `restrictAllComparison` function returns a new relation value where

```

```

/// the logical *AND* of all the comparisons given by `exprs` are true.
/// The `exprs` argument is taken as a list of triples.
/// Each triple is as described for the `comparativeSelector` function.
pub fn restrictAllComparison(
    self: *const Self,
    exprs: anytype,
) Self {
    return self.restrict(conjunctiveComparativeSelector, exprs);
}

```

```

<<relation public functions>>=
/// The `restrictAtLeastOneComparison` function returns a new relation value where
/// the logical *OR* of all the comparisons given by `exprs` are true.
/// The `exprs` argument is taken as a list of triples.
/// Each triple is as described for the `comparativeSelector` function.
pub fn restrictAtLeastOneComparison(
    self: *const Self,
    exprs: anytype,
) Self {
    return self.restrict(disjunctiveComparativeSelector, exprs);
}

```

Tests

```

<<relvalue module tests>>=
test "restrict relation value" {
    relExprBegin(); defer relExprEnd();

    const test_relation = create_test_relation();
    const all = test_relation.restrict(identitySelector, .{});
    try testing.expectEqual(@as(usize, 6), all.cardinality());

    const foo_named = test_relation.restrictEqual(.name, "foo");
    try testing.expectEqual(@as(usize, 3), foo_named.cardinality());

    const foo_and_0 = test_relation.restrictAllEqual(.{
        .{ .name, "foo" }, // .name == "foo" and
        .{ .value, 0 },   // .value == 0
    });
    try testing.expectEqual(@as(usize, 1), foo_and_0.cardinality());

    const foo_or_0 = test_relation.restrictAtLeastOneEqual(.{
        .{ .name, "foo" }, // .name == "foo" or
        .{ .value, 0 },   // .value == 0
    });
    try testing.expectEqual(@as(usize, 4), foo_or_0.cardinality());
}

```

```

const value_lt_2 = test_relation.restrictComparison(.value, .lt, 2);
try testing.expectEqual(@as(usize, 4), value_lt_2.cardinality());

const value_1_2_interval = test_relation.restrictAllComparison(.{
    .{ .value, .gte, 1},
    .{ .value, .lte, 2},
});
try testing.expectEqual(@as(usize, 4), value_1_2_interval.cardinality());
}

```

Join

Relational algebra defines several different join operations. In this algebra, two join implementation are provided:

natural join

The natural join yields a relation value whose heading is the union of the operand headings and whose body contains the set of tuples where like-named attributes between the tuples are equal.

grouped join

The grouped join yields a relation value where the joined tuples are placed in a relation-valued attribute. This type of join gives the same information as the so-called, “outer join”.

Natural Join

For the natural join, the returned heading is the union of the headings of the operands and returned tuple set contains those tuples where the values of like-named attributes are the same. If the headings of the two operands are disjoint, the result is the extended Cartesian product. Indeed, join is sometimes defined as the extended Cartesian product of the operands followed by a restriction on those tuples where the value of the like-named attributes are the same (and usually a little clutter of renaming and projection to keep the attribute naming correct).

```

<<relvalue module declarations>>=
const JoinHeadings = reltuple.JoinHeadings;

```

```

<<relation public functions>>=
/// The `join` function computes the natural join of the `joiner` and
/// `joinee` arguments. The returned relation value has a heading that is
/// the union of the headings of `joiner` and `joinee` and the returned
/// relation value has a tuple set containing those tuples where the
/// like-named attributes are equal. The `join` operation is commutative.
pub fn join(
    joiner: Self,
    joinee: anytype,
) RelValue(JoinHeadings(TupleBase, @TypeOf(joinee).TupleBase)) {
    const Joined = RelValue(JoinHeadings(TupleBase, @TypeOf(joinee).TupleBase));

    // Worst case is the two tuple headings are disjoint and the
    // result degenerates to the Cartesian product.
}

```

```

const joined_cardinality = joiner.cardinality() * joinee.cardinality();

var joined = Joined.createCapacity(joined_cardinality);
for (joiner.body()) |joiner_tuple| {
    for (joinee.body()) |joinee_tuple| {
        if (joiner_tuple.join(joinee_tuple)) |joined_tuple| {
            const inserted = joined.insertTuple(joined_tuple);
            assert(inserted);
        }
    }
}

return joined;
}

```

Tests

```

<<relvalue module tests>>=
test "join relation value" {
    relExprBegin(); defer relExprEnd();

    // This test uses three relation values which we describe
    // and populate first. The population is just fake data,
    // but the conclusions are true for the data values given.

    // In our world, an Author has a name and an affiliation with
    // some organization.
    const Author = struct {
        name: []const u8,
        affiliation: []const u8,
    };
    const authors = RelValue(Author).create(&.{.
        .name = "John Smith", .affiliation = "IniCom" },
        .name = "Jane Doe", .affiliation = "IGH" },
        .name = "Kevin James", .affiliation = "WannaBeU" },
    ));

    // Our world also includes Books, presumably written by some author.
    const Book = struct {
        title: []const u8,
        isbn: []const u8,
        sold: usize,
        price: f32, // just a test, don't use floats to hold real money
    };
    const books = RelValue(Book).create(&.{.
        .title = "The Good", .isbn = "32-78043", .sold = 200, .price = 19.95 },
        .title = "The Bad", .isbn = "62-18034", .sold = 50, .price = 29.95 },
        .title = "The Ugly", .isbn = "99-32053", .sold = 42, .price = 54.76 },
        .title = "The Cosi Cosi", .isbn = "10-38093", .sold = 79, .price = 8.99 },
    });
}

```

```

// Authors write books, possibly many but at least one.
// More than one author may help in writing a given book.
// We keep track of who did what in an Authorship relation.
const Authorship = struct {
    name: []const u8,
    title: []const u8,
};

const authorship = RelValue(Authorship).create(&.<
    .{ .name = "John Smith", .title = "The Good" },
    .{ .name = "Kevin James", .title = "The Good" },
    .{ .name = "Jane Doe", .title = "The Cosi Cosi" },
    .{ .name = "Jane Doe", .title = "The Bad" },
    .{ .name = "John Smith", .title = "The Ugly" },
);

// Consider John Smith. By restricting to obtain a
// singular relation value, we can join across authorship and
// books to obtain the data on John's work.
const author_john = authors.restrictEqual(.name, "John Smith");
const books_by_john = author_john.join(authorship);
const john_book_info = books_by_john.join(books);

try testing.expectEqual(@as(u32, 2), books_by_john.cardinality());
try testing.expectEqual(@as(u32, 2), john_book_info.cardinality());
try testing.expectEqual(@as(u32, 242), john_book_info.sum(.sold));
try testing.expectEqual(@as(u32, 200), books.max(.sold));
try testing.expectEqual(@as(u32, 42), books.min(.sold));
try testing.expectEqual(@as(f32, 54.76), books.max(.price));
try testing.expectEqual(@as(f32, 8.99), books.min(.price));

const avg_price = books.average(.price); // 28.41
const avg_match = math.approxEqRel(f32, avg_price, @as(f32, 28.41), @as(f32, 0.01));
try testing.expect(avg_match);

// join is commutative
const books_by_john_2 = authorship.join(author_john);
try testing.expect(books_by_john_2.eql(books_by_john));

// DUM works as an annihilator.
// Joining with DUM yields the empty relation.
try testing.expect(@TypeOf(authors).createEmptyeql(authors.join(dum())));
try testing.expectEqual(@as(u32, 0), authors.join(dum()).cardinality());
// DEE works as an identity.
try testing.expect(authors.join(dee()) .eql(authors));
}

```

Grouped Join

When two relation values are joined, any tuples where the like-named attributes are not equal are not included in the result. This is part of the definition of a join. Sometimes, what is needed is for all the tuples to be included and have some indication which ones had unmatched join attributes. In

conventional relation algebra formulations, this is usually accomplished using an “outer join” algebraic operation. The conventional outer join uses a NULL value for those attributes where there is no match.

Since we do not have NULL values in this algebra, we provide a relational form of the outer join that provides the information we need without relying on NULL values. We call this operation, “grouped join”. The essential difference is to replace NULL values with a relation-valued attribute that contains the tuples that are joined. That relation-valued attribute will be empty if none of the like-named attribute values match. Note the grouped join is **not** commutative.

There are two other functions needed to implement a grouped join. Like the other algebraic operations, we need a function to form a new heading for the grouped join. For other operations, we have used the functions provided by the `reltuple` module. Since we are about to build structures at `comptime`, we do some name shortening on a structure field.

```
<<relvalue module declarations>>=
const StructField = std.builtin.Type.StructField;
```

Since joining requires us to know the intersection of the tuple fields, we’ll use the function provided by the `reltuple.zig` module.

```
<<relvalue module declarations>>=
const IntersectFields = reltuple.IntersectFields;
```

```
<<relvalue module private functions>>=
/// The `GroupedJoinHeading` function creates a new heading structure
/// suitable for use in creating a grouped join relation value.
/// The returned heading consists of those attributes of `OneHeading`
/// plus an attribute named, `join_attr`. The `join_attr` is relation-valued
/// and consists of all the attributes of `OtherHeading` minus any
/// attributes that have the same name as those in `OneHeading`.
fn GroupedJoinHeading(
    comptime OneHeading: type,
    comptime OtherHeading: type,
    comptime join_attr: [:0]const u8,
) type {
    const one_fields = meta.fields(OneHeading);
    const other_fields = meta.fields(OtherHeading);

    const IntersectAttrMap = comptime IntersectFields(OneHeading, OtherHeading);
    const joined_field_count = one_fields.len + 1; // +1 for join_attr
    var joined_fields: [joined_field_count]StructField = undefined;

    // All the fields of the "One" heading are included.
    @memcpy(joined_fields[0..one_fields.len], one_fields);

    // All the fields of the "Other" heading minus the intersection attributes
    // are included in the relation-valued attribute named, "join_attr".
    var rva_fields: [other_fields.len]StructField = undefined;
    var rva_field_index: usize = 0;
    inline for (other_fields) |other_field| {
        if (!IntersectAttrMap.has(other_field.name)) {
```

```

        rva_fields[rva_field_index] = other_field;
        rva_field_index += 1;
    }

const RelValueAttr = RelValue(@Type(.{
    .@"struct" = .{
        .layout = .auto,
        .backing_integer = null,
        .fields = rva_fields[0..rva_field_index],
        .decls = &.{},
        .is_tuple = false,
    },
}));
```

// Place the relation-valued attribute into the result heading.

```

joined_fields[one_fields.len] = .{
    .name = join_attr,
    .type = RelValueAttr,
    .default_value_ptr = null,
    .is_comptime = false,
    .alignment = @alignOf(RelValueAttr),
};

return @Type(.{
    .@"struct" = .{
        .layout = .auto,
        .backing_integer = null,
        .fields = &joined_fields,
        .decls = &.{},
        .is_tuple = false,
    },
});
```

}

The second helper function required computes whether two tuples can be joined. Two tuples may be joined if their like-named attributes have the same value. Note that tuples with disjoint heading sets may be joined in the extended Cartesian product sense.

```

<<relvalue module private functions>>=
fn joinable(
    one: anytype,
    other: anytype,
) bool {
    const One = @TypeOf(one);
    if (comptime !reltuple.isRelTupleType(One))
        @compileError("type '" ++ @typeName(One) ++ "' is not a RelTuple() type");

    const Other = @TypeOf(other);
    if (comptime !reltuple.isRelTupleType(Other))
        @compileError("type '" ++ @typeName(Other) ++ "' is not a RelTuple() type");
```

```

const IntersectAttrMap = comptime IntersectFields(One.TupleBase, Other.TupleBase);
const one_attribute_names = comptime meta.fieldNames(One.TupleBase);

// To join, the tuples must have like-named attributes and
// no comparison failures of the corresponding values.
inline for (one_attribute_names) |attr_name| {
    const is_join_attr = comptime IntersectAttrMap.has(attr_name);
    if (is_join_attr and !attributeEql(
        @field(one.reltuple, attr_name),
        @field(other.reltuple, attr_name),))
    {
        return false;
    }
}
// N.B. no else clause, a null IntersectMap => attributes are disjoint.
// Disjoint tuples are "joinable" in the extended Cartesian product sense.
// Tuples are not joinable if they have common attributes which are not equal.

return true;
}

```

With the ability to create the grouped join heading and determine if two tuples may be joined, the `groupedJoin` operation is shown as follows.

```

<<relation public functions>>=
/// The `groupedJoin` function is a relational form of the outer
/// join that does not require NULL attribute values. The join is on
/// like-named attributes of `joiner` and `joinee` with the matching tuples placed in a
/// relation-valued attribute. Result relation has a heading consisting
/// of the attributes of the `joiner` relation value plus the `join_attr`
/// attribute. The heading of the `join_attr` relation is the same as
/// the `joinee` relation minus the like-named attributes on which the
/// join was computed.
pub fn groupedJoin(
    joiner: Self,
    joinee: anytype,
    comptime join_attr: [:0]const u8,
) RelValue(GroupedJoinHeading(TupleBase, @TypeOf(joinee).TupleBase, join_attr)) {
    const Joinee = @TypeOf(joinee);
    const GroupedJoined =
        RelValue(GroupedJoinHeading(TupleBase, Joinee.TupleBase, join_attr));

    // Following a strategy similar to group, build a hash map of the tuples
    // of joiner to record which tuples of "joinee" match the source tuples.
    const TupleSet = std.DynamicBitSet;
    const JoinMap = HashMapUnmanaged(
        Tuple,
        TupleSet,
        TupleHashContext(Tuple, enums.values(AttributeId)),
        default_load_factor,

```

```

);

var join_map: JoinMap = .{};

defer join_map.deinit(relvalue_allocator);
join_map.ensureTotalCapacity(
    relvalue_allocator,
    @intCast(joiner.cardinality()),
) catch |err|
    panic("failed to allocate join map memory, '{s}'\n", .{@errorName(err)});

// Create a mapping from joiner tuples to joinee tuples. When the
// both tuples match on the common attributes, add the joinee tuple
// index to the bit set that is the value of the hash map.
for (joiner.body()) |joiner_tuple| {
    var result = join_map.getOrPutAssumeCapacity(joiner_tuple);
    assert(!result.found_existing);
    result.value_ptr.* = TupleSet.initEmpty(
        relvalue_allocator,
        joinee.cardinality(),
    ) catch |err|
        panic("failed to allocate join map memory, '{s}'\n", .{@errorName(err)});
    for (joinee.body(), 0..) |joinee_tuple, joinee_index| {
        if (joinable(joiner_tuple, joinee_tuple)) {
            result.value_ptr.set(joinee_index);
        }
    }
}

// Iterate across the join map and build up the result. The join map
// has the value of the joiner tuples and a bit map of matching joinee
// tuple indices. We build a relation valued attribute for the joinee tuples
// and add it as an attribute to the joiner tuple. If the joiner
// bit set is empty, then the relation valued attribute is also empty.
var joined = GroupedJoined.createCapacity(joiner.cardinality());

const GroupedJoinedBase = GroupedJoined.TupleBase;
const GroupedRelType = @FieldType(GroupedJoinedBase, join_attr);
const GroupedTupleType = RelTuple(GroupedRelType.TupleBase);

var join_map_iter = join_map.iterator();
while (join_map_iter.next()) |map_entry| {
    // Copy over the attributes of the joiner value from the map entry.
    var join_tuple: GroupedJoinedBase = undefined;
    const joiner_attr_ids = comptime.enums.values(AttributeId);
    inline for (joiner_attr_ids) |joiner_attr_id| {
        const field_name = @tagName(joiner_attr_id);
        @field(join_tuple, field_name) =
            @field(map_entry.key_ptr.reltuple, field_name);
    }

    // Using the bit set from the map, generate the relation valued attribute
    // containing the joined tuples.
}

```

```

const joined_set = map_entry.value_ptr;
var joined_attr = GroupedRelType.createCapacity(joined_set.count());

// Tuple at a time, extract the joined tuples from the map entries.
var join_attr_value: GroupedRelType.TupleBase = undefined;
const join_attr_names = comptime meta.fieldNames(GroupedRelType.TupleBase);
var joined_set_iter = joined_set.iterator({});

while (joined_set_iter.next()) |joinee_tuple_index| {
    const tuple_to_group = joinee.storage[joinee_tuple_index];
    inline for (join_attr_names) |join_attr_name| {
        @field(join_attr_value, join_attr_name) =
            @field(tuple_to_group.reltuple, join_attr_name);
    }
    const inserted =
        joined_attr.insertTuple(GroupedTupleType.create(join_attr_value));
    assert(inserted);
}

joined_set.deinit();
@field(join_tuple, join_attr) = joined_attr;

const inserted =
    joined.insertTuple(RelTuple(GroupedJoined.TupleBase).create(join_tuple));
assert(inserted);
}

return joined;
}

```

Tests

```

<<relvalue module tests>>=
test "grouped join relation value" {
    relExprBegin();
    defer relExprEnd();

    // The data for this test was taken from Wikipedia
    // (https://en.wikipedia.org/wiki/Relational\_algebra#Outer\_joins).
    // The data seems strange so we'll try to make up a reasonable story.

    // We start with some subset of employees, knowing their
    // name, ID, and department in which they work.
    const Employee = RelValue(struct { name: []const u8, id: u16, dept: []const u8 });
    const employee = Employee.create(&{
        .{ .name = "Harry", .id = 27, .dept = "Finance" },
        .{ .name = "Sally", .id = 42, .dept = "Sales" },
        .{ .name = "George", .id = 175, .dept = "Finance" },
        .{ .name = "Harriet", .id = 200, .dept = "Sales" },
        .{ .name = "Tim", .id = 83, .dept = "Executive" },
    });
}

```

```

// We also have a collection of departments and who manages them.
// We presume this is a small selection of the departments in the company.
const Department = RelValue<struct { dept: []const u8, manager: []const u8 }>;
const dept = Department.create(&.<
    .{ .dept = "Sales", .manager = "Harriet" },
    .{ .dept = "Production", .manager = "Charles" },
);

// The grouped join answers the question, "Which employees are managed
// by one of the managers in our small collection of departments?"
// If we simply join the two relation values, Harry, George, and Tim
// do not appear in the join because they are in departments that
// do not appear in our department relation value.
const managed = employee.join(dept);
try testing.expectEqual(@as(u8, 2), managed.cardinality());
try testing.expectEqual(@as(u8, 0), managed
    .restrictEqual(.name, "Harry")
    .cardinality());

// The grouped join retains all the employees and indicates missing
// managers by creating a relation-valued attribute,
// managed_by, in this case. For Harry, that relation valued attribute
// will have a cardinality of 0. This gives the same information
// as a left outer join without the use of a NULL value.
const managers = employee.groupedJoin(dept, "managed_by");
//testing.print("managers\n{any}\n", .{managers});
try testing.expectEqual(@as(u8, 5), managers.cardinality());

const harry = managers.restrictEqual(.name, "Harry");
try testing.expectEqual(@as(u8, 0), (try harry.extract(.managed_by)).cardinality());

// Ungrouping the group joined relation value gives the same result as
// the natural join.
const ungrouped = managers.ungroup(.managed_by);
try testing.expect(ungrouped.eql(managed));

// By reversing the order of the two relations, we can achieve a result
// similar to the right outer join, again without any NULL values.
const employs = dept.groupedJoin(employee, "employs");
//testing.print("employs\n{any}\n", .{employs});
try testing.expectEqual(@as(u8, 2), employs.cardinality());

// In this case the query shows that Sales department employs two people
// from our set of employees and that the Production department employs
// none.
try testing.expectEqual(@as(u8, 2), (try employs.restrictEqual(.dept, "Sales")
    .extract(.employs))
    .cardinality());
try testing.expectEqual(@as(u8, 0), (try employs.restrictEqual(.dept, "Production")
    .extract(.employs))
    .cardinality());
}

```

Extension

As with Relation Tuples, a Relation Value may be extended by additional attributes. This operation gives the ability to perform calculations on a tuple-by-tuple basis. For relation values, specialized variations on extension are provided as direct operators in the algebra.

Extend

```
<<relvalue module declarations>>=
const ExtendHeadingMulti = reltuple.ExtendHeadingMulti;
```

```
<<relation public functions>>=
/// The `extend` function returns a new relation value where each tuple
/// has zero or more additional attributes.
/// All the attributes and values of the original tuple are preserved.
/// The `extendAttrs` argument is taken as a list of triples. Each triple contains:
///
/// 1. attribute name
/// 2. attribute type
/// 3. optional attribute default value
///
/// The `values` argument is taken as a list of pairs. Each pair contains:
///
/// 1. the attribute id of an extended attribute
/// 2. the value to which the extended attribute is to be set
///
/// All extended attributes must either have a value in the `values` argument
/// or have a default value given in the `extendAttr` argument.
/// These rules ensure that the extended attribute always have defined values.
/// If further flexibility is needed, an optional `optExtender` argument
/// can be provided. The `optExtender` is an optional function that accepts a
/// tuple as input and returns a tuple as output. The extender function is
/// invoked after the tuple values has been set using either an explicitly
/// given value or a default value and the tuple inserted into the relation
/// value is that returned from the extender function.
pub fn extend(
    self: *const Self,
    comptime extendAttrs: anytype,
    values: anytype,
    optExtender: ?*const fn (RelTuple(ExtendHeadingMulti(TupleBase, extendAttrs)))
        RelTuple(ExtendHeadingMulti(TupleBase, extendAttrs)),
) RelValue(ExtendHeadingMulti(TupleBase, extendAttrs)) {
    const Extended = RelValue(ExtendHeadingMulti(TupleBase, extendAttrs));

    var extended = Extended.createCapacity(self.cardinality());
    for (self.body()) |self_tuple| {
        var extended_tuple = self_tuple.extendMany(extendAttrs, values);
        if (optExtender) |extender| {
            extended_tuple = extender(extended_tuple);
```

```

    }
    const inserted = extended.insertTuple(extended_tuple);
    assert(inserted);
}

return extended;
}

```

Tests

```

<<relvalue module tests>>=
test "extend relation value" {
    relExprBegin(); defer relExprEnd();

    const ExtBase = struct {
        name: []const u8,
        len: usize,
        weight_lb: f32,
    };
    const ExtRelation = RelValue(ExtBase);
    const test_1 = ExtRelation.create(&.{.
        .{ .name = "small", .len = 37, .weight_lb = 5.0 },
        .{ .name = "medium", .len = 52, .weight_lb = 9.0 },
        .{ .name = "large", .len = 100, .weight_lb = 15.0 },
    });

    // Extend by two attribute, neither of which has a default value.
    const ext_attrs = .{ .{
        "weight_kg",
        f32,
        null,
    }, .{
        "weight_oz",
        f32,
        null,
    } };

    const ExtendedBaseTuple = RelTuple(ExtendHeadingMulti(ExtBase, ext_attrs));
    const extend_weight = struct {
        pub fn extend_weight(
            tup: ExtendedBaseTuple,
        ) ExtendedBaseTuple {
            return tup
                .update(.weight_kg, tup.extract(.weight_lb) * 2.2)
                .update(.weight_oz, tup.extract(.weight_lb) * 16.0);
        }
    }.extend_weight;

    const extended = test_1.extend(
        ext_attrs,

```

```

        .{ .{ .weight_kg, 0.0 }, .{ .weight_oz, 0.0 } },
        extend_weight,
    );

const small_props = extended.restrictEqual(.name, "small");

try testing.expectEqual(@as(usize, 1), small_props.cardinality());
try testing.expect((5.0 * 2.2) == try small_props.extract(.weight_kg));
try testing.expect((5.0 * 16.0) == try small_props.extract(.weight_oz));

const ext_2 = test_1.extend(
    ext_attrs,
    .{ .{ .weight_kg, 0.0 }, .{ .weight_oz, 0.0 } },
    null,
);

const small_0_props = ext_2.restrictEqual(.name, "small");

const small_0_kg = try small_0_props.extract(.weight_kg);
try testing.expectEqual(@as(f32, 0.0), small_0_kg);
const small_0_oz = try small_0_props.extract(.weight_oz);
try testing.expectEqual(@as(f32, 0.0), small_0_oz);
}

```

Tag

One useful extension operation of relation values is to “tag” the tuples. To tag the tuples, they are sorted into either ascending or descending order of one or more attributes and an additional `RelationCapacity` typed attribute is added. The added attribute contains the tuple’s offset in the sort order. The effect is to add an attribute which can be used to identify the tuples. In this case, we are only extending by a single attribute and use the `heading` function from the `reltuple.zig` module.

```

<<relvalue module declarations>>=
const ExtendHeadingSingle = reltuple.ExtendHeadingSingle;

```

Tagging can be used with projection to ensure that all the tuples are represented in the projected result. Some relation value projections are for attributes that do not necessarily have unique values. By first tagging the relation value and then projecting both the tag and the other desired attributes, you can be assured that the cardinality of the result is the same as the cardinality of the original relation value because the tag number serves to make each projected tuple unique.

```

<<relation public functions>>=
/// The `tag` function extends the given tuple by a single
/// attribute named, `tag_attr`. The type of `tag_attr` is
/// `RelationCapacity`. Tagging assigns a number starting at zero to
/// each tuple in the sorting order given by, `order`, and sorted over
/// the attributes given by `sortAttrs`.
pub fn tag(
    self: *const Self,

```

```

    comptime tag_attr: [:0]const u8,
    comptime order: IterationOrder,
    comptime sortAttrs: []const AttributeId,
) RelValue(ExtendHeadingSingle(TupleBase, tag_attr, RelationCapacity, null)) {
    const Tagged = RelValue(ExtendHeadingSingle(
        TupleBase,
        tag_attr,
        RelationCapacity,
        null,
));
}

var tagged = Tagged.createCapacity(self.cardinality());
const selfAttrs = comptime attributes();
var tagValue: RelationCapacity = 0;
var selfIter = self.iterator(order, sortAttrs);
while (selfIter.next()) |selfTuple| : (tagValue += 1) {
    var taggedValue: Tagged.TupleBase = undefined;
    inline for (selfAttrs) |selfAttr| {
        @field(taggedValue, selfAttr) = @field(selfTuple.reltuple, selfAttr);
    }
    @field(taggedValue, tagAttr) = tagValue;

    const taggedTuple = RelTuple(Tagged.TupleBase).create(taggedValue);
    const inserted = tagged.insertTuple(taggedTuple);
    assert(inserted);
}

return tagged;
}

```

Tests

```

<<relvalue module tests>>=
test "tag relation value" {
    relExprBegin(); defer relExprEnd();

    const ExtBase = struct {
        name: []const u8,
        len: usize,
        weight_lb: f32,
    };
    const ExtRelation = RelValue(ExtBase);
    const test_1 = ExtRelation.create(&{
        .{ .name = "small", .len = 37, .weight_lb = 5.0 },
        .{ .name = "medium", .len = 52, .weight_lb = 9.0 },
        .{ .name = "large", .len = 100, .weight_lb = 15.0 },
    });

    const taggedAsc = test_1.tag("len_order", .asc, &{.len});
    const smallest = taggedAsc.restrictComparison(.len_order, .lt, 1);
    try testing.expectEqual(@as(usize, 37), try smallest.extract(.len));
}

```

```

    const tagged_desc = test_1.tag("len_order", .desc, &{.len});
    const largest = tagged_desc.restrictComparison(.len_order, .lt, 1);
    try testing.expectEqual(@as(usize, 100), try largest.extract(.len));
}

```

Rank

The rank function is another extension function for relation values. It extends a relation value by one attribute. That attribute contains the rank of the tuple with respect to the other tuples in the relation variable based on the value of some attribute. In these types of queries, you find the top subset or bottom subset of the tuple values according to attribute values against which the ranking is computed. The subsets can be found by ranking the relation value and then restricting by the rank number.

```

<<relation public functions>>=
/// The `rank` function extends the given relation value with a new attribute,
/// `rank_attr`, whose value is the number of tuples in the given relation value
/// that are less than or equal to `rank_attr` if the requested order is `.desc`
/// or greater than or equal to `rank_attr` if the requested order is `.asc`.
/// For `.desc` order ranking, the higher ranking (and therefore lower ranking number)
/// is given to the larger values. Conversely, for the `.asc` order ranking,
/// the higher ranking is given to the smaller values. The computed rank number is
/// always greater than or equal to 1. It is possible for the
/// ranking of more than one tuple to be the same. It is usual to restrict a
/// ranked relation value by a rank number value. Note that because ranking
/// values may be equal, the cardinality of such a restriction is not necessarily
/// the same as the rank number used in the restriction.

pub fn rank(
    self: *const Self,
    comptime rank_attr_id: AttributeId,
    comptime order: IterationOrder,
    comptime rank_attr: [:0]const u8,
) RelValue(ExtendHeadingSingle(TupleBase, rank_attr, RelationCapacity, null)) {
    const Ranked = RelValue(ExtendHeadingSingle(TupleBase, rank_attr, RelationCapacity, null));

    var ranked = Ranked.createCapacity(self.cardinality());
    for (self.body()) |self_tuple| {
        var ranked_tuple = self_tuple.extend(rank_attr, RelationCapacity, 0, null);

        const ranked_value = self_tuple.extract(rank_attr_id);
        var rank_count: RelationCapacity = 0;
        for (self.body()) |compare_tuple| {
            const compare_value = compare_tuple.extract(rank_attr_id);
            const comparison = if (order == .asc)
                attributeLessThan(ranked_value, compare_value)
            else
                attributeLessThan(compare_value, ranked_value);
            rank_count += @intFromBool(comparison);
        }
    }
}

```

```

    @field(ranked_tuple.reltuple, rank_attr) = rank_count + 1; ①

    const inserted = ranked.insertTuple(ranked_tuple);
    assert(inserted);
}

return ranked;
}

```

- ① Only a *less than* comparison is computed. One is added because a tuple is always equal to itself. This has the effect of assigning the same rank number to those values that are equal. This indicates a “tie” in the ranking. The ranking number will be assigned the higher ranking (lower numerical value) and the lower rankings resume with the number after all the ties. For example, if there are two values that are highest ranked, both are assigned rank number 1. The next value after the two highest is assigned a rank number of 3. After the ranking, if the relation value is restricted by *rank* ≤ 1 , then the result has cardinality two, indicating the tie. If the ranked relation value is restricted by *rank* ≤ 2 , then both highest ranked tuples are returned as expected. If the ranking was strictly, *less than or equal to*, the highest ranked tuples would both have a ranking number of 2 and a restriction by, *rank* ≤ 1 , would unexpectedly be empty. Ranking code must be prepared to handle ties, which is indicated by a result with a cardinality different from the rank number restriction.

Tests

```

<<relvalue module tests>>=
test "rank relation value" {
    relExprBegin(); defer relExprEnd();

    const RankRelation = RelValue(struct {
        name: []const u8,
        len: usize,
        weight: f32,
    });
    const test_1 = RankRelation.create(&.{.
        .{ .name = "a1", .len = 37, .weight = 5.0 },
        .{ .name = "a2", .len = 38, .weight = 5.1 },
        .{ .name = "a3", .len = 36, .weight = 5.2 },
        .{ .name = "b1", .len = 52, .weight = 9.0 },
        .{ .name = "b2", .len = 53, .weight = 9.1 },
        .{ .name = "b3", .len = 53, .weight = 9.0 },
        .{ .name = "c1", .len = 100, .weight = 15.0 },
        .{ .name = "c2", .len = 115, .weight = 16.0 },
        .{ .name = "c3", .len = 95, .weight = 16.0 },
    });

    const rank_1 = test_1.rank(.len, .asc, "asc_len_rank");
    const longest = rank_1.restrictComparison(.asc_len_rank, .lte, 1);
    try testing.expectEqual(@as(usize, 115), try longest.extract(.len));
    //testing.print("\n{any}\n", .{rank_1});

    const rank_2 = test_1.rank(.len, .desc, "desc_len_rank");
    const shortest = rank_2.restrictComparison(.desc_len_rank, .lte, 1);
    try testing.expectEqual(@as(usize, 36), try shortest.extract(.len));
}

```

```

//testing.print("\n{any}\n", .{rank_2});

const rank_3 = test_1.rank(.name, .asc, "asc_name_rank");
const alpha_last = rank_3.restrictComparison(.asc_name_rank, .lte, 1);
try testing.expectEqualSlices(u8, "c3", try alpha_last.extract(.name));
//testing.print("\n{any}\n", .{rank_3});

const rank_4 = test_1.rank(.name, .desc, "desc_name_rank");
const alpha_first = rank_4.restrictComparison(.desc_name_rank, .lte, 1);
try testing.expectEqualSlices(u8, "a1", try alpha_first.extract(.name));
//testing.print("\n{any}\n", .{rank_4});
}

```

Modifying Relation Values

Relation values cannot be modified in place. The following functions return a new relation value that has updated attributes. There is a group of these functions that perform the updates depending upon the logical predicates given in the arguments.

Update

The `update` function uses two additional functions to operate. A `selector` function chooses which tuples to update. This is the same selection concept used in the `restrict` operation. A `updater` function returns the modified tuple. This is similar to the `extender` function used by the `extend` operation. Typically, an `updater` performs a tuple update operation on the input tuple and returns a new tuple with desired updates.

```

<<relation public functions>>=
/// The `update` function creates a new relation value that contains
/// tuples that may have been modified. The return result has the same
/// cardinality as the given relation value. The `selector` function is
/// invoked for each tuple and if it returns `true`, then the `updater`
/// function is invoked with the same tuple. The tuple value returned
/// from the `updater` function replaces the original tuple in the
/// resulting relation value. The `selector` and `updater` function are
/// passed the `selection_args` and `update_args` arguments, respectively.
pub fn update(
    self: *const Self,
    selector: *const fn(tuple: Tuple, args: anytype) bool,
    selection_args: anytype,
    updater: *const fn(tuple: Tuple, args: anytype) Tuple,
    update_args: anytype,
) RelValueError!Self {
    // Updating does not change the number of tuples in the relation value.
    var updated = createCapacity(self.cardinality());
    errdefer updated.destroy();

    for (self.body()) |self_tuple| {
        const updated_tuple = if (selector(self_tuple, selection_args))
            updater(self_tuple, update_args)

```

```

    else
        self_tuple;

    const inserted = updated.insertTuple(updated_tuple);
    if (!inserted) {
        return RelValueError.DuplicateTuple;
    }
}

return updated;
}

```

As with restriction, common case updating functions are provided.

```

<<relvalue module public functions>>=
/// The `attributeUpdater` function is used with the `update` operation
/// to modify the value of a single attribute. The `update_args`
/// argument is taken as a pair consisting of the attribute name /
/// attribute value to be updated.
pub fn attributeUpdater(
    tuple: anytype,
    update_args: anytype,
) @TypeOf(tuple) {
    assert(update_args.len == 2);
    return tuple.update(update_args.@"0", update_args.@"1");
}

```

```

<<relvalue module public functions>>=
/// The `manyAttributeUpdater` function is used with the `update`
/// operation to update the values of zero or more attributes.
/// The `update_args` argument is taken to be a list of attribute name/
/// attribute value pairs.
pub fn manyAttributeUpdater(
    tuple: anytype,
    update_args: anytype,
) @TypeOf(tuple) {
    return tuple.updateMany(update_args);
}

```

```

<<relvalue module public functions>>=
/// The `incrAttributeUpdater` is used with the `update` operation
/// to add a value to an attribute. It performs the necessary read / modify /
/// update sequence to change the attribute value. The `update_args`
/// argument is taken as a pair of attribute name and increment value.

```

```

/// The named attribute is updated by adding the increment value.
pub fn incrAttributeUpdater(
    tuple: anytype,
    update_args: anytype,
) @TypeOf(tuple) {
    assert(update_args.len == 2);
    return tuple.update(
        update_args=@"0",
        tuple.extract(update_args[@"0"]) + update_args[@"1"],
    );
}

```

Update Attribute Value For All Tuples

As with the `restrict` function, a set of commonly useful functions are provided. These function use the previous updater functions to provide a more convenient calling interface.

```

<<relation public functions>>=
/// The `updateAllAttributes` function returns a new relation value where all
/// of the attributes given by, `attr_id` are set to the value given by, `attr_value`.
pub fn updateAllAttributes(
    self: *const Self,
    comptime attr_id: AttributeId,
    attr_value: anytype,
) RelValueError!Self {
    return self.update(identitySelector, {}, attributeUpdater, {attr_id, attr_value});
}

```

Update Attribute Value Matching Specified Attribute

```

<<relation public functions>>=
/// The `updateWhereEqual` function returns a new relation value where
/// all of the attributes where the value of `select_attr_id` equals
/// `select_attr_value`. For the matching attributes, the value of the
/// `update_attr_id` is set to `update_attr_value`.
pub fn updateWhereEqual(
    self: *const Self,
    comptime select_attr_id: AttributeId,
    select_attr_value: AttributeType(select_attr_id),
    comptime update_attr_id: AttributeId,
    update_attr_value: AttributeType(update_attr_id),
) RelValueError!Self {
    return self.update(
        equalitySelector,
        {select_attr_id, select_attr_value},
        attributeUpdater,
        {update_attr_id, update_attr_value},
    );
}

```

```
);  
}
```

Update Many Attribute Values Matching Specified Attribute

```
<<relation public functions>>=  
/// The `updateManyWhereEqual` function returns a new relation value  
/// where all of the attributes where the value of `select_attr_id` equals  
/// `select_attr_value` have been updated according to the attributes and  
/// values in the `update_args` argument. The `update_args` argument is  
/// taken to be a list of attribute name/ attribute value pairs. This  
/// function allows updating multiple attributes for each matching tuple.  
pub fn updateManyWhereEqual(  
    self: *const Self,  
    comptime select_attr_id: AttributeId,  
    select_attr_value: AttributeType(select_attr_id),  
    update_args: anytype,  
) RelValueError!Self {  
    return self.update(  
        equalitySelector,  
        .{select_attr_id, select_attr_value},  
        manyAttributeUpdater,  
        update_args,  
    );  
}
```

Update Attribute Value Matching All Specified Attributes

```
<<relation public functions>>=  
/// The `updateWhereAllEqual` function returns a new relation value  
/// where all of the attributes which match the logical *AND* of the  
/// equality of the attribute id / attribute value pairs given by  
/// `select_av_pairs` is true. For the matching tuples, the value  
/// of the `update_attr_id` is set to `update_attr_value`.  
pub fn updateWhereAllEqual(  
    self: *const Self,  
    select_av_pairs: anytype,  
    comptime update_attr_id: AttributeId,  
    update_attr_value: AttributeType(update_attr_id),  
) RelValueError!Self {  
    return self.update(  
        conjunctiveEqualitySelector,  
        select_av_pairs,  
        attributeUpdater,  
        .{update_attr_id, update_attr_value},  
    );
```

```
}
```

Update Attribute Many Values Matching All Specified Attributes

```
<<relation public functions>>=
/// The `updateManyWhereAllEqual` function returns a new relation
/// value where all of the attributes which match the logical *AND* of
/// the equality of the attribute id / attribute value pairs given by
/// `select_av_pairs` is true. For the matching tuples, the values
/// of all the attribute id / attribute value pairs given by by the
/// `update_av_pairs` are updated into the tuple.
pub fn updateManyWhereAllEqual(
    self: *const Self,
    select_av_pairs: anytype,
    update_av_pairs: anytype,
) RelValueError!Self {
    return self.update(
        conjunctiveEqualitySelector,
        select_av_pairs,
        manyAttributeUpdater,
        update_av_pairs,
    );
}
```

Update Attribute Value Matching Some Specified Attributes

```
<<relation public functions>>=
/// The `updateWhereAtLeastOneEqual` function returns a new relation value
/// where all of the attributes which match the logical *OR* of the
/// equality of the attribute id / attribute value pairs given by
/// `select_av_pairs` is true. For the matching tuples, the value
/// of the `update_attr_id` is set to `update_attr_value`.
pub fn updateWhereAtLeastOneEqual(
    self: *const Self,
    select_av_pairs: anytype,
    comptime update_attr_id: AttributeId,
    update_attr_value: AttributeType(update_attr_id),
) RelValueError!Self {
    return self.update(
        disjunctiveEqualitySelector,
        select_av_pairs,
        attributeUpdater,
        .{update_attr_id, update_attr_value},
    );
}
```

Update Many Attribute Values Matching Some Specified Attributes

```
<<relation public functions>>=
/// The `updateManyWhereAtLeastOneEqual` function returns a new relation
/// value where all of the attributes which match the logical *OR* of
/// the equality of the attribute id / attribute value pairs given by
/// `select_av_pairs` is true. For the matching tuples, the values
/// of all the attribute id / attribute value pairs given by by the
/// `update_av_pairs` are updated into the tuple.
pub fn updateManyWhereAtLeastOneEqual(
    self: *const Self,
    select_av_pairs: anytype,
    update_av_pairs: anytype,
) RelValueError!Self {
    return self.update(
        disjunctiveEqualitySelector,
        select_av_pairs,
        manyAttributeUpdater,
        update_av_pairs,
    );
}
```

```
<<relvalue module tests>>=
test "update relation value" {
    relExprBegin(); defer relExprEnd();

    const test_relation = create_test_relation();

    // Update the tuple with .name = "foo" to .name = "baz".
    const updated_1 = try test_relation.updateWhereEqual(
        .name,
        "foo",
        .name,
        "baz",
    );

    const baz_named = updated_1.restrictEqual(.name, "baz");
    try testing.expectEqual(@as(usize, 3), baz_named.cardinality());

    // For foo's, values of 0, 1, 2 now become 1, 2, 3.
    const updated_2 = try test_relation.update(
        equalitySelector,
        .{ .name, "foo" },
        incrAttributeUpdater,
        .{ .value, 1 },
    );
    const foo_updated_2 = updated_2.restrictEqual(.name, "foo");
    try testing.expectEqual(@as(usize, 6), foo_updated_2.sum(.value));
}
```

```

// Update any .value that is 3 to be 4.
const updated_3 = try updated_2.updateWhereEqual(.value, 3, .value, 4);
const foo_updated_3 = updated_3.restrictEqual(.name, "foo");
try testing.expectEqual(@as(usize, 7), foo_updated_3.sum(.value));

// Force a duplicate tuple by forcing .value from 0 to 1.
const set_0_1 = test_relation.updateWhereAllEqual(
    .{
        .{ .name, "foo" },
        .{ .value, 0 },
    },
    .value,
    1,
);
try testing.expectError(RelValueError.DuplicateTuple, set_0_1);

// Update all the .value attributes to fixed number.
// This causes duplicate tuples.
const value_4 = test_relation.updateAllAttributes(.value, 10);
try testing.expectError(RelValueError.DuplicateTuple, value_4);
}

```

Relation Heading Operations

The operations in this section construct relation values that have modifications to the heading of the input operand. Borrowing from Relation Tuples, the `rename`, `wrap`, and `unwrap` operations perform their operation to all the tuples in the relation value. By analogy to `wrap` and `unwrap`, the `group` and `ungroup` operations create relation-valued attributes from a subset of the relation heading.

Rename

Renaming is most frequently used to obtain like-named attributes for use in a `join` operation. We use the `header type` function for renaming from the `reltuple` module.

```

<<relvalue module declarations>>=
const RenameHeading = reltuple.RenameHeading;

```

```

<<relation public functions>>=
/// The `rename` function returns a new relation value which is the same
/// as the given value except that the names of zero or more attributes
/// have been modified. Renaming attributes is not allowed to create
/// duplicate attribute names.
pub fn rename(
    self: *const Self,
    comptime renamings: anytype,
) RelValue(RenameHeading(TupleBase, renamings)) {
    const Renamed = RelValue(RenameHeading(TupleBase, renamings));
}

```

```

var renamed = Renamed.createCapacity(self.cardinality());
for (self.body() |self_tuple| {
    const renamed_tuple = self_tuple.rename(renamings);
    const inserted = renamed.insertTuple(renamed_tuple);
    assert(inserted);
}

return renamed;
}

```

Tests

```

<<relvalue module tests>>=
test "rename relation value" {
    relExprBegin(); defer relExprEnd();

    const test_relation = create_test_relation();
    const renamed = test_relation.rename(.{
        .{ .name, "Id" },
        .{ .value, "size" },
    });
    const sizes = renamed.project(&{.size});
    try testing.expectEqual(@as(u8, 3), sizes.cardinality());
}

```

Wrap

As with Relation Tuples, you can *wrap* attributes from a Relation Value into a tuple-valued attribute for all the tuples in the relation value.

The following shows a tabular representation of a relation value where several attributes have been wrapped into a tuple-valued attribute.

+-----+ RelValue +-----+ name address				
+-----+ []const u8 RelTuple				
+---+ +---+ +---+ +---+ number street city state +---+ +---+ +---+ +---+ usize []const u8 []const u8 []const u8				
+-----+ Rhonda 1050 Elm Street San Francisco CA				
+-----+ John 1127 Embarcadero San Francisco CA				

```

|Sue      |100    |Pine Street|San Francisco|CA      |
+-----+-----+-----+-----+
|George   |100    |Pine Street|San Francisco|CA      |
+-----+-----+-----+-----+

```

```

<<relvalue module declarations>>=
const WrapHeading = reltuple.WrapHeading;

```

```

<<relation public functions>>=
/// The `wrap` function returns a new relation value where attributes
/// of the given relation value are combined into a tuple-valued
/// attribute. The name of the tuple-valued attribute is given by the
/// `new_attr` argument. The attributes given by the `wrapAttrs` are
/// the heading for `new_attr`. The heading of the returned relation
/// value is the same as the given relation value minus the `wrapAttrs`
/// plus the `new_attr`. The cardinality of the returned relation value
/// is the same as the input operand.
pub fn wrap(
    self: *const Self,
    comptime new_attr: [:0]const u8,
    comptime wrapAttrs: []const AttributeId,
) RelValue(WrapHeading(TupleBase, new_attr, wrapAttrs)) {
    const Wrapped = RelValue(WrapHeading(TupleBase, new_attr, wrapAttrs));

    var wrapped = Wrapped.createCapacity(self.cardinality());
    for (self.body()) |self_tuple| {
        const wrap_tuple = self_tuple.wrap(new_attr, wrapAttrs);
        const inserted = wrapped.insertTuple(wrap_tuple);
        assert(inserted);
    }

    return wrapped;
}

```

Unwrap

Unwrapping a relation value simply undoes what the wrap function does.

```

<<relvalue module declarations>>=
const UnwrapHeading = reltuple.UnwrapHeading;

```

```

<<relation public functions>>=
/// The `unwrap` function returns a new relation value where the attributes
/// of the tuple-valued attribute given by, `attr_id`, are included in the
/// heading as ordinary attributes. The `attr_id` attribute must be tuple valued,

```

```

/// and the heading of `attr_id` may not contain like-named attributes
/// of the original heading. The return result has the cardinality of the
/// input operand and a heading consisting of all the input operand attributes,
/// minus `attr_id` plus all the attributes of `attr_id`.
pub fn unwrap(
    self: *const Self,
    comptime attr_id: AttributeId,
) RelValue(UnwrapHeading(TupleBase, attr_id)) {
    const Unwrapped = RelValue(UnwrapHeading(TupleBase, attr_id));

    var unwrapped = Unwrapped.createCapacity(self.cardinality());
    for (self.body() |self_tuple| {
        const unwrap_tuple = self_tuple.unwrap(attr_id);
        const inserted = unwrapped.insertTuple(unwrap_tuple);
        assert(inserted);
    }

    return unwrapped;
}

```

Tests

```

<<relvalue module tests>>=
test "wrap/unwrap relation value" {
    relExprBegin(); defer relExprEnd();

    const Person = RelValue(struct {
        name: []const u8,
        number: usize,
        street: []const u8,
        city: []const u8,
        state: []const u8,
    });
    const people = Person.create(&.<
        .{
            .name = "Rhonda",
            .number = 1050,
            .street = "Elm Street",
            .city = "San Francisco",
            .state = "CA",
        },
        .{
            .name = "John",
            .number = 1127,
            .street = "Embarcadero",
            .city = "San Francisco",
            .state = "CA",
        },
        .{
    
```

```

        .name = "Sue",
        .number = 100,
        .street = "Pine Street",
        .city = "San Francisco",
        .state = "CA",
    },
    .{
        .name = "George",
        .number = 100,
        .street = "Pine Street",
        .city = "San Francisco",
        .state = "CA",
    },
});

const wrap_people = people.wrap("address", &{ .number, .street, .city, .state });
try testing.expectEqual(@as(usize, 2), wrap_people.degree());
try testing.expectEqual(@as(usize, 4), wrap_people.cardinality());

const john_people = wrap_people.restrictEqual(.name, "John");
try testing.expectEqual(@as(usize, 1), john_people.cardinality());
try testing.expectEqualSlices(u8, "CA",
    (try john_people.extract(.address)).extract(.state));

const unwrap_people = wrap_people.unwrap(.address);
try testing.expectEqual(@as(usize, 5), unwrap_people.degree());
try testing.expectEqual(@as(usize, 4), unwrap_people.cardinality());
try testing.expect(unwrap_people.eql(people));
}

```

Group

The group operation is analogous to wrap. Where wrap created a tuple-valued attribute, group creates a relation-valued attribute. The heading of the resulting relation value contains all the attributes that were **not** grouped together plus a new attribute that is relation-valued. The relation-valued attribute holds the set of tuples where the ungrouped attribute(s) match. This implies that the cardinality of the grouped relation may be less than that of the input operand.

A tabular representation is helpful. Consider the following relation value:

+-----+		
RelValue		
+-----+-----+-----+		
owner_name dog_name	acquired	
+-----+-----+-----+		
[]const u8 []const u8 u16		
+-----+-----+-----+		
Sue	Fido	2001
+-----+-----+-----+		
Sue	Sam	2000
+-----+-----+-----+		
George	Fido	2001

George	Sam	2000	
Alice	Spot	2001	
Mike	Rover	2002	
Jim	Fred	2003	

In this data set, Sue and George are co-owners of two dogs.

If this relation value is grouped by `dog_name` and acquired into a new attribute called, `acquisition`, a tabular representation is:

RelValue			
owner_name	acquisition		
[]const u8	RelValue		
	dog_name	acquired	
	[]const u8	u16	
Sue	Fido	2001	
	Sam	2000	
Alice	Spot	2001	
Mike	Rover	2002	
Jim	Fred	2003	
George	Fido	2001	
	Sam	2000	

Notice the reduction in cardinality of the grouped result because Sue and George have both acquired two dogs that are the same.

Strategy of the implementation is to build a map keyed by the ungrouped attributes whose value is a bit-set indicating which tuples in the relation match the ungrouped attributes. Then, the map is traversed building the new relation from the remaining attribute values and a relation-valued attribute containing the grouped attributes as stored in the bit-set value from the map.

<<relation public functions>>=

```

/// The `group` function creates a new relation value where a subset
/// of attributes have been gathered together in a relation-valued
/// attribute. The heading of the result is the heading of the input
/// operand, minus the `group_attrs` attributes plus a relation-valued
/// attribute named, `new_attr`. The body of the relation value consists
/// of those tuples where the ungrouped attributes match along with a
/// relation-valued attribute whose heading is the `grouped_attrs`. The
/// body of the relation-valued attribute are those tuples composed from
/// the grouped attributes of the matching ungrouped attributes.

pub fn group(
    self: *const Self,
    comptime new_attr: [:0]const u8,
    comptime groupAttrs: []const AttributeId,
) RelValue(GroupHeading(TupleBase, new_attr, groupAttrs)) {
    // The attributes ids that remain from the heading when the
    // grouped attributes are removed.
    comptime var remainAttrs: [Tuple.tuple_degree - groupAttrs.len]AttributeId =
        undefined;
    inline for (0..Tuple.tuple_degree) |field_index| {
        comptime var match = false;
        inline for (groupAttrs) |groupAttrId| {
            match = match or (field_index == @intFromEnum(groupAttrId));
        }
        if (!match) {
            remainAttrs[field_index] = @enumFromInt(field_index);
        }
    }

    // Use a dynamic bit set since we only know the cardinality of `self` at
    // run time.
    const TupleSet = std.DynamicBitSet;
    const hashAttrs = remainAttrs; // prevent comptime var reference escape
    const GroupMap = HashMapUnmanaged(
        Tuple,
        TupleSet,
        TupleHashContext(Tuple, &hashAttrs), ①
        default_load_factor,
    );

    var groupMap: GroupMap = {};
    defer groupMap.deinit(relvalue_allocator);
    groupMap.ensureTotalCapacity(
        relvalue_allocator,
        @intCast(self.cardinality()),
    ) catch |err|
        panic("failed to allocate group map: '{s}'\n", {@errorName(err)});

    for (self.body(), 0..) |self_tuple, tuple_index| {
        var result = groupMap.getOrPutAssumeCapacity(self_tuple);
        // First time inserting a tuple, initialize the bit set.
        if (!result.found_existing) {
            result.value_ptr.* =

```

```

        TupleSet.initEmpty(relvalue_allocator, self.cardinality()) catch |err|
            panic("failed to allocate tuple bit set: '{s}'\n", .{@errorName(err)});
    }
    result.value_ptr.set(tuple_index);
}

// The relation-valued attribute is a projection of the groupAttrs over
// the TupleBase of the input operand.
const GroupedAttrHeading = ProjectHeading(TupleBase, groupAttrs);
const GroupedAttrTuple = RelTuple(GroupedAttrHeading);
const GroupedAttribute = RelValue(GroupedAttrHeading);
// The GroupHeading type function constructs the TupleBase for the result.
const GroupedBase = GroupHeading(TupleBase, new_attr, groupAttrs);
const GroupedTuple = RelTuple(GroupedBase);
const Grouped = RelValue(GroupedBase);
var grouped = Grouped.createCapacity(group_map.count());

var group_map_iter = group_map.iterator();
while (group_map_iter.next() |map_entry| {
    // Copy over the remaining attributes from the map entry.
    var grouped_tuple: GroupedTuple = undefined;
    inline for (remainAttrs) |remain_attr_id| {
        @field(grouped_tuple.reltuple, @tagName(remain_attr_id)) =
            @field(map_entry.key_ptr.reltuple, @tagName(remain_attr_id));
    }
}

// Using the bit set from the hash map, generate the relation-valued attribute.
const group_set = map_entry.value_ptr;
var grouped_attr = GroupedAttribute.createCapacity(group_set.count());

// Tuple at a time, extract the grouped attributes from the map entries
var group_set_iter = group_set.iterator(.{});
while (group_set_iter.next() |tuple_index| {
    var group_attr_tuple: GroupedAttrTuple = undefined;
    const tuple_to_wrap = self.storage[tuple_index];
    inline for (groupAttrs) |group_attr| {
        @field(group_attr_tuple.reltuple, @tagName(group_attr)) =
            @field(tuple_to_wrap.reltuple, @tagName(group_attr));
    }
    const inserted = grouped_attr.insertTuple(group_attr_tuple);
    assert(inserted);
}
group_set.deinit();
@field(grouped_tuple.reltuple, new_attr) = grouped_attr;

const inserted = grouped.insertTuple(grouped_tuple);
assert(inserted);
}

return grouped;
}

```

① The hash is computed over the remaining attributes only. This is what gives us the *grouping*.

Since Relation Tuples do not support a group operation, we must supply the heading calculation here.

```
<<relvalue module private functions>>=
fn GroupHeading(
    comptime Heading: type,
    comptime new_attr: [:0]const u8,
    comptime groupAttrs: []const meta.FieldEnum(Heading),
) type {
    const GroupedAttribute = RelValue(ProjectHeading(Heading, groupAttrs));

    const HeadingAttributeId = meta.FieldEnum(Heading);
    const groupAttrSet = EnumSet(HeadingAttributeId).initMany(groupAttrs);

    const headingFields = meta.fields(Heading);
    const groupedFieldCount = headingFields.len - groupAttrs.len + 1; // new attribute
    var groupedHeadingFields: [groupedFieldCount]StructField = undefined;
    var groupedFieldCounter: usize = 0;
    inline for (headingFields, 0..) |headingField, fieldIndex| {
        if (comptime !groupAttrSet.contains(@enumFromInt(fieldIndex))) {
            groupedHeadingFields[groupedFieldCounter] = headingField;
            groupedFieldCounter += 1;
        }
    }

    groupedHeadingFields[groupedFieldCounter] = .{
        .name = new_attr,
        .type = GroupedAttribute,
        .defaultValue_ptr = null,
        .is_comptime = false,
        .alignment = @alignOf(GroupedAttribute),
    };

    return @Type(.{
        .@"struct" = .{
            .layout = .auto,
            .fields = &groupedHeadingFields,
            .decls = &.{},
            .is_tuple = false,
        },
    });
}
```

Tests

```
<<relvalue module tests>>=
```

```

test "group relation value" {
    relExprBegin(); defer relExprEnd();

    const Ownership = RelValue(struct {
        owner_name: []const u8,
        dog_name: []const u8,
        acquired: u16,
    });

    const ownership = Ownership.create(&{.
        .{ .owner_name = "Sue", .dog_name = "Fido", .acquired = 2001 },
        .{ .owner_name = "Sue", .dog_name = "Sam", .acquired = 2000 },
        .{ .owner_name = "George", .dog_name = "Fido", .acquired = 2001 },
        .{ .owner_name = "George", .dog_name = "Sam", .acquired = 2000 },
        .{ .owner_name = "Alice", .dog_name = "Spot", .acquired = 2001 },
        .{ .owner_name = "Mike", .dog_name = "Rover", .acquired = 2002 },
        .{ .owner_name = "Jim", .dog_name = "Fred", .acquired = 2003 },
    });

    const acquisition = ownership.group("acquisition", &{ .dog_name, .acquired });
    //testing.print("\n{any}\n", .{ownership});
    //testing.print("\n{any}\n", .{acquisition});

    try testing.expectEqual(@as(usize, 2), acquisition.degree());
    try testing.expectEqual(@as(usize, 5), acquisition.cardinality());

    const sue_rec = acquisition.restrictEqual(.owner_name, "Sue");
    const sue_acquired = try sue_rec.extract(.acquisition);
    try testing.expectEqual(@as(usize, 2), sue_acquired.cardinality());

    const sue_fido = sue_acquired.restrictEqual(.dog_name, "Fido");
    try testing.expectEqual(@as(usize, 1), sue_fido.cardinality());
    try testing.expectEqualSlices(u8, "Fido", try sue_fido.extract(.dog_name));
    try testing.expectEqual(@as(u16, 2001), try sue_fido.extract(.acquired));

    const sue_sam = sue_acquired.restrictEqual(.dog_name, "Sam");
    try testing.expectEqual(@as(usize, 1), sue_sam.cardinality());
    try testing.expectEqualSlices(u8, "Sam", try sue_sam.extract(.dog_name));
    try testing.expectEqual(@as(u16, 2000), try sue_sam.extract(.acquired));
}

```

Ungroup

A group operation can be inverted by using ungroup.

```

<<relation public functions>>=
/// The `ungroup` function returns a new relation value where a
/// relation-valued attribute named, `attr`, is placed back into
/// the result. The heading of the returned result consists of the
/// heading of the given relation value, minus `attr`, and plus

```

```

/// the attributes from the heading of `attr`. The body of the
/// return result is the extended Cartesian product of the
/// tuples in input operand and the tuples contained in `attr`.
pub fn ungroup(
    self: *const Self,
    comptime attr: Tuple.AttributeId,
) RelValue(UngroupHeading(TupleBase, attr)) {
    const UngroupedBase = UngroupHeading(TupleBase, attr);
    const UngroupedTuple = RelTuple(UngroupedBase);
    const Ungrouped = RelValue(UngroupedBase);

    // Pass over the relation value to sum the cardinalities of the
    // relation valued attributes.
    var result_cardinality: usize = 0;
    for (self.body()) |self_tuple| {
        const grouped_attr_value = self_tuple.extract(attr);
        result_cardinality += grouped_attr_value.cardinality();
    }

    const attr_names = comptime meta.fieldNames(TupleBase);
    const GroupedAttrType = @FieldType(TupleBase, @tagName(attr));
    const GroupedTuple = GroupedAttrType.Tuple;
    const grouped_attr_names = comptime meta.fieldNames(GroupedTuple.TupleBase);

    var ungrouped = Ungrouped.createCapacity(result_cardinality);

    for (self.body()) |self_tuple| {
        // copy over the remaining attributes from the map entry.
        var ungrouped_tuple: UngroupedTuple = undefined;
        inline for (attr_names, 0..) |attr_name, attr_index| {
            const attr_id: Tuple.AttributeId = @enumFromInt(attr_index);
            if (attr_id != attr) {
                @field(ungrouped_tuple.reltuple, attr_name) =
                    @field(self_tuple.reltuple, attr_name);
            }
        }
        inline for (0..attr_names.len) |attr_index| {
            const attr_id: Tuple.AttributeId = @enumFromInt(attr_index);
            if (attr_id == attr) {
                const grouped_attr_value = self_tuple.extract(attr);
                for (grouped_attr_value.body()) |grouped_tuple| {
                    inline for (grouped_attr_names) |grouped_attr_name| {
                        @field(ungrouped_tuple.reltuple, grouped_attr_name) =
                            @field(grouped_tuple.reltuple, grouped_attr_name);
                    }
                    const inserted = ungrouped.insertTuple(ungrouped_tuple);
                    assert(inserted);
                }
            }
        }
    }
}

```

```

    return ungrouped;
}

```

Like grouping, we must supply a type function to compute the heading of an ungroup operation.

```

<<relvalue module private functions>>=
fn UngroupHeading(
    comptime Heading: type,
    comptime attr: meta.FieldEnum(Heading),
) type {
    const GroupedAttrType = @FieldType(Heading, @tagName(attr));
    if (!isRelValueType(GroupedAttrType)) {
        @compileError("attribute '" ++ @typeName(GroupedAttrType) ++
            "' is not a relation value");
    }

    const HeadingAttributeId = meta.FieldEnum(Heading);
    const heading_fields = meta.fields(Heading);
    const ungrouped_field_count: usize = heading_fields.len +
        GroupedAttrType.Tuple.tuple_degree - 1; // -1 the ungroup attribute is removed
    var ungrouped_heading_fields: [ungrouped_field_count]StructField = undefined;
    var ungrouped_index: usize = 0;
    inline for (heading_fields, 0..) |heading_field, field_index| {
        const attr_id: HeadingAttributeId = @enumFromInt(field_index);
        if (attr_id == attr) {
            const grouped_info = meta.fields(GroupedAttrType.Tuple.TupleBase);
            inline for (grouped_info) |grouped_field| {
                ungrouped_heading_fields[ungrouped_index] = grouped_field;
                ungrouped_index += 1;
            }
        } else {
            ungrouped_heading_fields[ungrouped_index] = heading_field;
            ungrouped_index += 1;
        }
    }

    return @Type(.{
        .@"struct" = .{
            .layout = .auto,
            .fields = &ungrouped_heading_fields,
            .decls = &.{},
            .is_tuple = false,
        },
    });
}

```

Tests

```

<<relvalue module tests>>=

```

```

test "ungroup relation value" {
    relExprBegin(); defer relExprEnd();

    const Ownership = RelValue(struct {
        owner_name: []const u8,
        dog_name: []const u8,
        acquired: u16,
    });

    const ownership = Ownership.create(&.{.
        .{ .owner_name = "Sue", .dog_name = "Fido", .acquired = 2001 },
        .{ .owner_name = "Sue", .dog_name = "Sam", .acquired = 2000 },
        .{ .owner_name = "George", .dog_name = "Fido", .acquired = 2001 },
        .{ .owner_name = "George", .dog_name = "Sam", .acquired = 2000 },
        .{ .owner_name = "Alice", .dog_name = "Spot", .acquired = 2001 },
        .{ .owner_name = "Mike", .dog_name = "Rover", .acquired = 2002 },
        .{ .owner_name = "Jim", .dog_name = "Fred", .acquired = 2003 },
    });

    const acquisition = ownership.group("acquisition", &{ .dog_name, .acquired });
    try testing.expectEqual(@as(usize, 2), acquisition.degree());
    try testing.expectEqual(@as(usize, 5), acquisition.cardinality());

    const orig_rel = acquisition.ungroup(.acquisition);
    try testing.expectEqual(@as(usize, 7), orig_rel.cardinality());
    const sue_record = orig_rel.restrictEqual(.owner_name, "Sue");
    try testing.expectEqual(@as(usize, 2), sue_record.cardinality());

    const george_record = orig_rel.restrictEqual(.owner_name, "George");
    try testing.expectEqual(@as(usize, 2), george_record.cardinality());

    const jim_record = orig_rel.restrictEqual(.owner_name, "Jim");
    try testing.expectEqual(@as(usize, 1), jim_record.cardinality());

    const mike_record = orig_rel.restrictEqual(.owner_name, "Mike");
    try testing.expectEqual(@as(usize, 1), mike_record.cardinality());

    const alice_record = orig_rel.restrictEqual(.owner_name, "Alice");
    try testing.expectEqual(@as(usize, 1), alice_record.cardinality());
}

```

Transitive Closure

A relational formulation of hierarchical data is verbose. This can be surprising to newcomers to relation data theory. Trees are so common, regularly taught, and have such elegant recursiveness that it is easy to forget that hierarchical data structures have many *rules* encoded into the data structure itself. Rules about root nodes, intermediate nodes, and leaf nodes are part of the constraints. All nodes, except for leaf nodes, have child nodes. All nodes, except for the root node, have a parent. In relational terms, these rules must be expressed explicitly.

Graph oriented data, of which trees are an example, is difficult to handle with conventional relational algebra. Most algebras add a *transitive closure* operation. This allows answering graph reachability questions.

The algorithm used here is taken from [Aho, Hopcroft, and Ullman](#). p. 213. Be aware that transitive closure is more computationally expensive than other provided operations. It is $O(n^3)$ in operation and $O(n^2)$ in memory. This implementation also has some overhead to scan the relation value to set up the main part of the algorithm and finally to produce the resulting relation. That is **not** to say that transitive closure should be avoided. There are no alternatives in relational algebra to computing this type of information. The transitive closure of a binary relation cannot, in general, be expressed in first-order logic^[7].

```

<<relation public functions>>=
/// The `tclose` function computes the transitive closure
/// for the given relation value. The given relation must be of degree 2.
/// The returned result has the same heading as the given relation value
/// and the body contains the transitive closure of the relation value
/// over the attributes. Note that the given relation value
/// always appears in the result.
pub fn tclose(
    self: *const Self,
) Self {
    if (Tuple.tuple_degree != 2) {
        @compileError("transitive closure requires a binary relation value");
    }

    const attrs = comptime enums.values(AttributeId);
    const HeadType = comptime AttributeType(attrs[0]);
    const TailType = comptime AttributeType(attrs[1]);
    if (HeadType != TailType)
        @compileError("transitive closure attributes must be the same type");

    // Build a two mappings between closure attributes and a set of
    // consecutive array indices. One mapping is for the attribute
    // values (so as to eliminate duplicates) and one for the indices
    // (to facilitate value lookup when constructing the result).
    // Make a single pass over the relation value to construct both the
    // mappings and to establish the initial set of edges (for those vertices
    // where there is an edge connection in the relation value itself).

    const max_vertices = self.cardinality() * 2; // worst case

    const AttrMap = HashMapUnmanaged(
        HeadType,
        usize, // used for the initial population of the adjacency matrix
        TcloseHashContext(HeadType),
        default_load_factor,
    );
    var attr_map: AttrMap = .{};

    attr_map.ensureTotalCapacity(
        relvalue_allocator,
        @intCast(max_vertices),
    ) catch |err|
        panic("failed to allocate closure attribute map: '{s}'\n", .{@errorName(err)});
    defer attr_map.deinit(relvalue_allocator);
}

```

```

var index_map = relExprAlloc(HeadType, max_vertices);
defer relvalue_allocator.free(index_map);

// Dynamically allocate the worst case size adjacency matrix. This is
// raw memory for a 2-D matrix. To be more parsimonious about memory
// usage, would require another pass across the relation value. The
// pass we make below both builds the attribute and index mappings
// and sets the initial values in the adjacency matrix. If two passes
// were made, then the first pass would only index the attributes.
// The size of the attribute map would then allow allocating the
// required memory for the adjacency matrix precisely, but a second
// pass would be required to establish the initial set of edges. We
// are trading off memory usage to avoid another pass over the tuples
// in the relation value. It is difficult to determine if this is
// a good trade off without some additional characterization. We
// are less concerned about wasting this memory here since it is
// recovered at the end of the function. We count the number of
// tuples in the returned result exactly since that memory remains
// allocated during the entire relational expression.
const mat = relExprAlloc(bool, max_vertices * max_vertices);
defer relvalue_allocator.free(mat);
@memset(mat, false);

// Slice the 2-D matrix memory to make the indexing operations easier.
var adj_matrix = relExprAlloc([]bool, max_vertices);
defer relvalue_allocator.free(adj_matrix);

var row_start: usize = 0;
for (adj_matrix) |*matrix_row| {
    const row_end = row_start + max_vertices;
    matrix_row.* = mat[row_start..row_end];
    row_start = row_end;
}

// Since we allocated the adjacency matrix for worst case usage,
// it is necessary to tally the actual usage.
var max_adj_matrix_index: usize = 0;
// Tally the cardinality of the resulting closure.
// Each time we set an adjacency matrix cell to 1, `closure_cardinality`
// is incremented.
var closure_cardinality: usize = 0;
for (self.body()) |self_tuple| {
    const value_1 = self_tuple.extract(attrs[0]);
    const result_1 = attr_map.getOrPutAssumeCapacity(value_1);
    const i: usize = if (result_1.found_existing)
        result_1.value_ptr.*;
    else nblk: {
        const vertex_index = max_adj_matrix_index;
        max_adj_matrix_index += 1;

        result_1.value_ptr.* = vertex_index;
}

```

```

        index_map[vertex_index] = value_1;

        break :nblk vertex_index;
    };

    const value_2 = self_tuple.extract(attrs[1]);
    const result_2 = attr_map.getOrPutAssumeCapacity(value_2);
    const j: usize = if (result_2.found_existing)
        result_2.value_ptr.*;
    else nblk: {
        const vertex_index = max_adj_matrix_index;
        max_adj_matrix_index += 1;

        result_2.value_ptr.* = vertex_index;
        index_map[vertex_index] = value_2;

        break :nblk vertex_index;
    };

    // Set the [i, j] cell of the adjacency matrix as an initial edge.
    adj_matrix[i][j] = true;
    closure_cardinality += 1;
}

// Compute the transitive edges of the adjacency matrix.
// Here's where the magic happens.
var k: usize = 0;
while (k < max_adj_matrix_index) : (k += 1) {
    var i: usize = 0;
    while (i < max_adj_matrix_index) : (i += 1) {
        var j: usize = 0;
        while (j < max_adj_matrix_index) : (j += 1) {
            if (!adj_matrix[i][j]) {
                adj_matrix[i][j] = adj_matrix[i][k] and adj_matrix[k][j];
                if (adj_matrix[i][j]) {
                    closure_cardinality += 1;
                }
            }
        }
    }
}

// Now build the resulting relation.
var closure = createCapacity(closure_cardinality);

var i: usize = 0;
while (i < max_adj_matrix_index) : (i += 1) {
    var j: usize = 0;
    while (j < max_adj_matrix_index) : (j += 1) {
        if (adj_matrix[i][j]) {
            const value_1 = index_map[i];
            const value_2 = index_map[j];
            var tuple: TupleBase = undefined;

```

```

        @field(tuple, @tagName(attrs[0])) = value_1;
        @field(tuple, @tagName(attrs[1])) = value_2;
        const inserted = closure.insertTuple(RelTuple(TupleBase).create(tuple));
        assert(inserted);
    }
}

return closure;
}

```

The following hash context is constructed specifically for transitive closure.

```

<<relation private functions>>=
// The `TcloseHashContext` type function computes a hash map context used
// for accumulating closure attribute values for the `tclose` function.
fn TcloseHashContext(
    comptime K: type,
) type {
    return struct {
        const CtxSelf = @This();

        pub fn hash(
            self: CtxSelf,
            value: K,
        ) u64 {
            _ = self;
            var hasher = std.hash.Fnv1a_64.init();
            hasher.update(mem.asBytes(&value));
            return hasher.final();
        }

        pub fn eql(
            self: CtxSelf,
            a: K,
            b: K,
        ) bool {
            _ = self;
            return meta.eql(a, b);
        }
    };
}

```

Tests

```

<<relvalue module tests>>=
test "tclose relation value" {

```

```

relExprBegin(); defer relExprEnd();

const BinRel = RelValue(struct {a: []const u8, b: []const u8});

// single path
//      +---+
//      | A |
//      +---+
//      |
//      v
//      +---+
//      | B |
//      +---+
//      |
//      v
//      +---+
//      | C |
//      +---+
const test_1 = BinRel.create(&.{.
    .{ .a = "A", .b = "B" },
    .{ .a = "B", .b = "C" },
});
const close_1 = test_1.tclose();
const expect_1 = BinRel.create(&.{.
    .{ .a = "A", .b = "B" },
    .{ .a = "B", .b = "C" },
    .{ .a = "A", .b = "C" }, // C is reachable from A
});
try testing.expect(close_1.eql(expect_1));

const test_2 = BinRel.create(&.{ // reverse path
    .{ .a = "B", .b = "A" },
    .{ .a = "C", .b = "B" },
});
const close_2 = test_2.tclose();
const expect_2 = BinRel.create(&.{.
    .{ .a = "B", .b = "A" },
    .{ .a = "C", .b = "B" },
    .{ .a = "C", .b = "A" }, // A is reachable from C
});
try testing.expect(close_2.eql(expect_2));

// joining path
//      +---+
//      | A |
//      +---+
//      |
//      v   v
//      +---+ +---+
//      | B | | C |
//      +---+ +---+
//      |

```

```

//      v   v
//      +---+
//      | D |
//      +---+
const test_3 = BinRel.create(&.<
    .{ .a = "A", .b = "B" },
    .{ .a = "A", .b = "C" },
    .{ .a = "B", .b = "D" },
    .{ .a = "C", .b = "D" },
});
const close_3 = test_3.tclose();
const expect_3 = BinRel.create(&.<
    .{ .a = "A", .b = "B" },
    .{ .a = "A", .b = "C" },
    .{ .a = "B", .b = "D" },
    .{ .a = "C", .b = "D" },
    .{ .a = "A", .b = "D" }, // There is a path from A to D.
>);
try testing.expect(close_3.eql(expect_3));

// diverging path
//      +---+
//      | A |
//      +---+
//      |   |
//      v   v
//      +---+ +---+     +---+
//      | B | | C | ----->| E |
//      +---+ +---+     +---+
//      |   |           |
//      v   v           v
//      +---+           +---+
//      | D |           | F |
//      +---+           +---+
const test_4 = BinRel.create(&.<
    .{ .a = "A", .b = "B" },
    .{ .a = "A", .b = "C" },
    .{ .a = "B", .b = "D" },
    .{ .a = "C", .b = "D" },
    .{ .a = "C", .b = "E" },
    .{ .a = "E", .b = "F" },
});
const close_4 = test_4.tclose();
const expect_4 = BinRel.create(&.<
    .{ .a = "A", .b = "B" },
    .{ .a = "A", .b = "C" },
    .{ .a = "A", .b = "D" },
    .{ .a = "A", .b = "E" },
    .{ .a = "A", .b = "F" },
    .{ .a = "B", .b = "D" },
    .{ .a = "C", .b = "D" },
    .{ .a = "C", .b = "E" },

```

```

    .{ .a = "C", .b = "F" },
    .{ .a = "E", .b = "F" }, // No paths from D or F to anywhere
});
try testing.expect(close_4.eql(expect_4));

const test_5 = BinRel.create(&.{

    .{ .a = "v1", .b = "v1" },
    .{ .a = "v1", .b = "v2" },
    .{ .a = "v1", .b = "v3" },
    .{ .a = "v2", .b = "v1" },
    .{ .a = "v3", .b = "v2" },
});

const close_5 = test_5.tclose(); // AHU example
const expect_5 = BinRel.create(&.{

    .{ .a = "v1", .b = "v1" },
    .{ .a = "v1", .b = "v2" },
    .{ .a = "v1", .b = "v3" },
    .{ .a = "v2", .b = "v1" },
    .{ .a = "v2", .b = "v2" },
    .{ .a = "v2", .b = "v3" },
    .{ .a = "v3", .b = "v1" },
    .{ .a = "v3", .b = "v2" },
    .{ .a = "v3", .b = "v3" },
});

try testing.expect(close_5.eql(expect_5));

const test_6 = BinRel.create(&.{

    .{ .a = "root", .b = "n1" },
    .{ .a = "root", .b = "n2" },
    .{ .a = "n1", .b = "n3" },
    .{ .a = "n1", .b = "n4" },
    .{ .a = "n2", .b = "n5" },
    .{ .a = "n2", .b = "n6" },
});

const close_6 = test_6.tclose(); // tree
const expect_6 = BinRel.create(&.{

    .{ .a = "root", .b = "n1" },
    .{ .a = "root", .b = "n2" },
    .{ .a = "root", .b = "n3" },
    .{ .a = "root", .b = "n4" },
    .{ .a = "root", .b = "n5" },
    .{ .a = "root", .b = "n6" },
    .{ .a = "n1", .b = "n3" },
    .{ .a = "n1", .b = "n4" },
    .{ .a = "n2", .b = "n5" },
    .{ .a = "n2", .b = "n6" },
});

try testing.expect(close_6.eql(expect_6));
}

```

Ordering Operations

As we have stated previously, there is no inherent order to the tuples in a relation value. This is not to say that order is unnecessary or to be avoided. The design decision is when to introduce ordering operations.

For this design, order is introduced during iteration over the relation value. In the usual relational expression, there is no explicit iteration: that's what the relational algebra is all about: operations are done set-at-a-time. However, when relationally organized data crosses the boundaries to another data organization, some iteration is usually required. For example, serializing data for communications is an archetypical example of having to devise a scheme to treat data values as sequences of bytes.

Iteration

In this section, an iterator is shown that visits each tuple in either ascending order or descending order of some subset of attributes. Not every attribute of a tuple may have a total order. For example, we do not consider tuple values as supporting any sense of order. Relation values are ordered only in the sense that subsets provide an analogy to relational operations such as less than. We must also be careful of imposing an order on string data. This design uses the simplistic notion of string order being the same as lexicographical ordering of the bytes in the string. In general, string ordering is much more complicated.

```
<<relvalue module declarations>>=
/// The `IterationOrder` enumeration encodes the choice of ordering
/// for an iterator.
pub const IterationOrder = enum {
    asc,
    desc,
};
```

To visit the tuples of a relation value in some order implies that there is a sorting process going on. We intend to use the heap sort function of the standard library. The sort algorithm performs the sort *in situ*, *i.e.* the elements in the container are swapped as necessary to end up with the requested order. In this case, it is not desirable to swap the tuples of a relation value. Rather, we build a sorted index into the tuples. As the sort proceeds, we arrange to compare tuple attributes, but should swapping be necessary, only the indices of the tuples and not the tuple values themselves are moved. At the end of the sorting process, we have generated a mapping. As you iterate, the iteration is actually across the mapping where each value in the mapping is the index of the tuple that is next in sorted order. The mapping creates a level of indirection so that the tuple values do not have to be moved.

The interface to the heap sort function in the standard library gives two ways to attach the sort to the application. First, a sorting context is provided. This context is passed to a `lessThanFn` which determines the relationship between two tuples. Since we are sorting tuple indices, the comparison function needs access to the slice of tuples to compare appropriate attribute values.

```
<<relation declarations>>=
/// The context used when sorting tuples in a relation value
/// for use with an iterator.
const RelSortContext = struct {
    tuples: []const Tuple,
```

```
};
```

What is presented to the `lessThanFn` is a pair of tuple indices. What the function calculates is the relative order of the tuples based on both sorting order, ascending or descending, and a subset of tuple attributes. First, we need a `comptime` function to create the comparison function, given the ordering of the sort and attributes that are subject to the sort.

```
<<relation private functions>>=
/// The `SortLessThanFn` function creates a function at `comptime`
/// suitable for use in sorting the tuples in a relation value.
fn SortLessThanFn(
    comptime order: IterationOrder,
    comptime sortAttrs: []const AttributeId,
) fn(RelSortContext, RelationCapacity, RelationCapacity) bool {
    return struct {
        fn compare(
            cntx: RelSortContext,
            a: RelationCapacity,
            b: RelationCapacity,
        ) bool {
            const a_tuple = cntx.tuples[a];
            const b_tuple = cntx.tuples[b];
            var less_than = true;
            inline for (sortAttrs) |sortAttr| {
                const attr_name = @tagName(sortAttr);
                const a_attr_value = @field(a_tuple.reltuple, attr_name);
                const b_attr_value = @field(b_tuple.reltuple, attr_name);
                const comparison = if (order == .asc)
                    attributeLessThan(a_attr_value, b_attr_value)
                else
                    attributeLessThan(b_attr_value, a_attr_value);
                less_than = less_than and comparison;
            }
            return less_than;
        }
    }.compare;
}
```

In addition to attribute equality, we need to compute the order of attributes for those attribute types which are comparable. As of the the 0.12.X development of the Zig compiler, the `meta.trait` module was removed. The function, `isZigString`, function of the removed module has been placed in its own file. This function determines if a given data type can be coerced to a canonical string type. It is used in some in computing the *less than* relation during sorting.

```
<<relvalue module declarations>>=
const trait = @import("zigtrait.zig");
```

```

<<relvalue module public functions>>=
/// The `attributeLessThan` function compares the `one_value` and
/// `other_value` attribute values and returns `true` if
/// `one_value` is less than `other_value`.
/// User defined types that are structures may define a `lessThan` function
/// and that function is used in the comparison. If the values are
/// relation values, then `less than` is interpreted as `properSubSetOf` .
/// Relation tuple values cannot be compared.
pub fn attributeLessThan(
    one_value: anytype,
    other_value: anytype,
) bool {
    const OneType = @TypeOf(one_value);

    if (comptime trait.isZigString(OneType)) {
        const one_string: []const u8 = one_value;
        const other_string: []const u8 = other_value;
        return mem.lessThan(u8, one_string, other_string); ①
    } else {
        return if (comptime isRelValueType(OneType))
            one_value.properSubSetOf(other_value)
        else if (@typeInfo(OneType) == .@("struct" and meta.hasFn(OneType, "lessThan")))
            one_value.lessThan(other_value) ②
        else
            math.compare(one_value, .lt, other_value); ③
    }
}

```

- ① Simplistic string comparison based on lexicographic order.
- ② If there is a `lessThan` function defined on a structure, then use that.
- ③ For everything else, `math.compare` determines attribute comparisons.

We now have the pieces to put together an iterator function. This function follows the iterator pattern used in the standard library.

```

<<relation public functions>>=
/// The `iterator` function generates an iterator for a `RelValue` that visits
/// the tuples in the order given by the `order` argument based on the
/// sorting the tuples by the attribute values given by the attribute ids in
/// the `sortAttrs` argument.
/// The tuples are obtained in the sorted order by invoking the `next` function
/// on the returned type.
/// If the `sortAttrs` are empty, i.e. `sortAttrs.len` is 0, then the
/// order of the returned tuples is indeterminate.
pub fn iterator(
    self: *const Self,
    comptime order: IterationOrder,
    comptime sortAttrs: []const AttributeId,
) Iterator() {

```

```

// Initialize a sorting map to the identity mapping.
var sort_map = relExprAlloc(RelationCapacity, self.cardinality());
var index: RelationCapacity = 0;
while (index < sort_map.len) : (index += 1) {
    sort_map[index] = index;
}

if (sort_attrs.len != 0) {
    sort.heap(
        RelationCapacity,
        sort_map,
        RelSortContext{ .tuples = self.body() },
        SortLessThanFn(order, sort_attrs),
    );
}

return .{
    .sort_map = sort_map,
    .tuples = self.body(),
};
}

```

```

<<relation private functions>>=
/// The `Iterator` type function returns a `struct` that contains a `next`
/// function that can be used to access tuples in a relation value one at a time.
fn Iterator() type {
    return struct {
        const IterSelf = @This();

        cursor: RelationCapacity = 0,
        sort_map: []RelationCapacity,
        tuples: []const Tuple,

        pub fn next(
            self: *IterSelf,
        ) ?Tuple {
            if (self.cursor >= self.sort_map.len) return null;

            const tup = self.tuples[self.sort_map[self.cursor]]; ①
            self.cursor += 1;
            return tup;
        }
    };
}

```

① Note the indirection through the sort map.

```
<<relvalue module tests>>=
```

```

test "iteration" {
    relExprBegin(); defer relExprEnd();

    const Person = RelValue(struct {
        name: []const u8,
        number: usize,
        street: []const u8,
        city: []const u8,
        state: []const u8,
    });
    const people = Person.create(&.<
        .{
            .name = "Rhonda",
            .number = 1050,
            .street = "Elm Street",
            .city = "San Francisco",
            .state = "CA",
        },
        .{
            .name = "John",
            .number = 1127,
            .street = "Embarcadero",
            .city = "San Francisco",
            .state = "CA",
        },
        .{
            .name = "Sue",
            .number = 100,
            .street = "Pine Street",
            .city = "San Francisco",
            .state = "CA",
        },
        .{
            .name = "George",
            .number = 100,
            .street = "Pine Street",
            .city = "San Francisco",
            .state = "CA",
        },
    >);
}

var people_iter = people.iterator(.asc, &{.number});
var person_number = people_iter.next()?.extract(.number);
try testing.expectEqual(@as(usize, 100), person_number);
person_number = people_iter.next()?.extract(.number);
try testing.expectEqual(@as(usize, 100), person_number);
person_number = people_iter.next()?.extract(.number);
try testing.expectEqual(@as(usize, 1050), person_number);
person_number = people_iter.next()?.extract(.number);
try testing.expectEqual(@as(usize, 1127), person_number);
try testing.expect(people_iter.next() == null);

```

```

people_iter = people.iterator(.desc, &{.number});
person_number = people_iter.next()?.extract(.number);
try testing.expectEqual(@as(usize, 1127), person_number);
person_number = people_iter.next()?.extract(.number);
try testing.expectEqual(@as(usize, 1050), person_number);
person_number = people_iter.next()?.extract(.number);
try testing.expectEqual(@as(usize, 100), person_number);
person_number = people_iter.next()?.extract(.number);
try testing.expectEqual(@as(usize, 100), person_number);
try testing.expect(people_iter.next() == null);

people_iter = people.iterator(.asc, &{.name});
var person_name = people_iter.next()?.extract(.name);
try testing.expectEqualSlices(u8, "George", person_name);
person_name = people_iter.next()?.extract(.name);
try testing.expectEqualSlices(u8, "John", person_name);
person_name = people_iter.next()?.extract(.name);
try testing.expectEqualSlices(u8, "Rhonda", person_name);
person_name = people_iter.next()?.extract(.name);
try testing.expectEqualSlices(u8, "Sue", person_name);
try testing.expect(people_iter.next() == null);

people_iter = people.iterator(.desc, &{.name});
person_name = people_iter.next()?.extract(.name);
try testing.expectEqualSlices(u8, "Sue", person_name);
person_name = people_iter.next()?.extract(.name);
try testing.expectEqualSlices(u8, "Rhonda", person_name);
person_name = people_iter.next()?.extract(.name);
try testing.expectEqualSlices(u8, "John", person_name);
person_name = people_iter.next()?.extract(.name);
try testing.expectEqualSlices(u8, "George", person_name);
try testing.expect(people_iter.next() == null);

// Sorting on multiple attributes can yield unanticipated results.
// In this case we sort, ascending, on both name and number.
// But note that since Sue and George both have 100 as their
// number attribute, the two attributes do not form a total
// order. You cannot say that Sue is "less than" George or
// that George is "less than" Sue because one of the sort
// attributes has the same value. So the order between these
// two tuples is not specified by the sorting attributes.
// However, they are visited before any of the other tuples
// because there is enough difference with the other tuples
// to distinguish a "less than" notion for them.

people_iter = people.iterator(.asc, &{ .name, .number });
// Skip George and Sue, they don't have a total order and
// the sort leaves them in an indeterminate order.
person_name = people_iter.next()?.extract(.name);
person_name = people_iter.next()?.extract(.name);
person_name = people_iter.next()?.extract(.name);
try testing.expectEqualSlices(u8, "Rhonda", person_name);
person_name = people_iter.next()?.extract(.name);

```

```
    try testing.expectEqualSlices(u8, "John", person_name);
}
```

Omitted Operations

As stated previously, relation value algebra is open ended so there is no standard, required set of operations. In this algebra, there are several interesting operations that have been omitted. The reason for the omission is the expectation that they would rarely be used. Since the omitted operations are not primitives, they can be calculated from the other algebraic operations presented here. If further characterization of this module shows the missing operations are more frequently used than assumed here, the operations may be added at a later time. The missing operations that are included in other relation value algebras are:

- Semi-join
- Semi-minus (aka anti-join).
- Compose
- Summarize
- Division
- Theta-join

Formatted Output

Like with tuples, tabular representations of relation values are indispensable when developing relational algebraic expressions.

```
<<relvalue module declarations>>=
const reformat = @import("reformat.zig");
const RelFormatter = reformat.RelFormatter;
```

```
<<relation public functions>>=
/// The `format` function is provided for use by the standard library
/// output formatting specialized for `RelValue` values.
/// Note that no formatting options are used and no format string is
/// accepted.
pub fn format(
    self: Self,
    writer: *std.IO.Writer,
) std.IO.Writer.Error!void {
    var formatter = RelFormatter.init(relvalue_allocator) catch
        return std.IO.Writer.Error.WriteFailed;
    defer formatter.deinit();
    try writer.writeAll(formatter.formatRelValue(self)) catch
        return std.IO.Writer.Error.WriteFailed;
}
```

Code layout

```
<<relvalue.zig>>=
```

```

///! The `relvalue.zig` module implements the concept of a *Relation Value*.
///! A Relation Value is a set of Relation Tuples, all of which
///! have the same heading. The heading is a set of attribute names
///! / attribute data types as for a Relation Tuple. Relation values
///! are immutable in the sense that no functions are provided to modify
///! directly the tuple values of the relation value. The attributes
///! of a relation heading have no inherent _order_. Relational Values
///! are type generators and new headings are created in a large number
///! of circumstances. The notion of heading equality is independent of
///! the order the attributes are actually stored in memory. Note also
///! that there is no inherent order to the tuples in a relation value.
///! The tuples of a relation value form a set, so there are no duplicate
///! tuples.

<<edit warning>>
<<copyright info>>

const std = @import("std");
const builtin = @import("builtin");

const heap = std.heap;
const mem = std.mem;
const math = std.math;
const enums = std.enums;
const EnumSet = enums.EnumSet;
const meta = std.meta;
const sort = std.sort;
const testing = if (builtin.os.tag == .freestanding)
    @import("resee_testing")
else
    std.testing;
const HashMapUnmanaged = std.HashMapUnmanaged;

const assert = std.debug.assert;
const panic = std.debug.panic;

pub const version = "1.0.0-a1";

<<relvalue module declarations>>

<<relation value>>

<<relvalue module public functions>>

<<relvalue module private functions>>

<<relvalue module tests>>

```

Relation Variables

A Relation Variable is, conceptually, a location in memory that holds a Relation Value. Since the headings of all the tuples of a Relation Value are equal, it is convenient to consider a relation variable as having the same heading as the relation value it holds.

In the last section, we saw that Relation Values are ephemeral and only exist during the evaluation of a relational expression. Relation variables persist through the lifetime of the program and hold all the application program data. Typically, the results of relation value expressions are stored in relation variables^[8].

However, relation variables have other properties that are essential to their role in the relational model of data and to their utility in this context.

- Relation variables introduce the notion of a *minimal* identifier^[9]. Recall that the tuples contained in a relation value must be unique. Every attribute of the tuple is considered part of the identification scheme, *i.e.* every attribute is part of the set member identification for the tuple. For relation variables, we allow for uniquely identifying tuples using a subset of the tuple attributes. Further, we allow a relation variable to have multiple identifiers with the rule that the attribute sets of any two identifiers may *not* be subsets of each other^[10].
- Relation variables participate in relationships. A relationship is an implementation of a [mathematical binary relation](#) ^[11]. There are two types of relationships:
 - a. Associations are based on subsets of the extended Cartesian product.
 - b. Generalizations are based on the disjoint union of sets.
- Relation variables may be updated in place. They are, after all, *variables* in the usual computing sense of that term. However, relation variables form a *shadow* variable mechanism to Zig programming language variables. They share some properties with Zig variables because the relation variables clearly must be implemented by some form of Zig variable. This is discussed below.

These additional properties cause us to consider an implementation that is substantially different than simply holding a relation value is some memory location that is outside of a relation expression evaluation.

To reiterate, we are *not* building a RDMS here. For this implementation, the lifetime for relation variables starts when the program starts, ends at program termination, and provides no automatic persistence to external storage. In this sense, relation variables have the usual characteristics of programming language variables. Many concepts from the relational model are used, such as identification constraints and referential consistency, but the many other guarantees of RDMS are not implemented in this execution model. If your application requires such capability, then this execution model is probably not the correct one to use.

Relation Variable Storage

Since a relation variable holds the persistent data state of the program, the tuples in the conceptual relation value held in a relation variable can change during the runtime of the program. Tuples contained in a relation variable may be created, deleted, or updated by program execution. In keeping with our pattern of frugal use of memory and avoidance of heap allocated memory, we place an upper bound on the number of Relation Tuples that can be held in a given Relation Variable. This design decision guides us toward storing the tuples of a relation variable in a array, whose size is fixed at compile time. The array is treated as a memory pool and pool items can be dynamically allocated. We must also allow for tuples to be deleted and the memory of deleted tuples is reused.

Representing a Relation Variable

In previous sections, the storage design for relation tuples and relation values was closely matched to the Zig language construct of a `struct` with fields. A tuple is an augmented `struct` with functions to provide the tuple algebraic operations. Limits were placed on the data types that could be specified for tuple attributes. Thus, the Zig `struct` representing a relational tuple can be

instantiated by supplying values for the attributes and we require that there can be multiple relation tuples of the same type.

Similar reasoning applies to relation values. In that case, mapping a relation value onto a Zig struct having a field that is a slice of tuples matches the uses to which relation values are put. Again, we require that there can be multiple instances of relation values of the same type containing, in general, different relational tuples.

A relation variable, however, has different characteristics. There is no requirement to have multiple relation variables of the same type within any given subject matter domain. In fact, we want to insure that a relation variable is a singleton in whatever context it is defined and used. This design maps a relation variable onto a Zig namespace `struct` which uses a container level variable to store the relation variable tuples. Zig namespaces are not intended to be instantiated, but can contain variables, constants, and functions. For relation variables, the tuples are stored in a container variable of the associated namespace and the container functions provide the necessary algebraic operations.

Since we are intending to store relation tuples and handle relation values, we import the files from the previous two sections.

```
<<relvar module declarations>>=
const reltuple = @import("reltuple.zig");
const RelTuple = reltuple.RelTuple;

const relvalue = @import("relvalue.zig");
const RelValue = relvalue.RelValue;
```

Following our established pattern, a type function is used to declare the namespace type for a Relation Variable.

```
<<relation variable>>=
/// `RelVar` is a type function which generates a namespace type suitable for
/// use as a Relation Variable whose Relation Tuples may be dynamically
/// created, deleted, read, and updated. The heading of the `RelVar`
/// is given by the `Heading` argument. Tuples in the relation variable
/// are identified by a non-empty set of identifiers as given by the
/// `identifiers` argument. The `identifiers` argument is a slice
/// of slices. The outer slice holds each identifier. The inner slice
/// holds the set of attribute ids for the attributes that constitute the
/// identifier. An identifier defines a set of attributes whose values,
/// taken together, uniquely identify a tuple in the relation variable. A
/// `RelVar` must have at least one identifier, each identifier must
/// have at least one attribute, and no two identifiers may have
/// attribute sets that are subsets of each other. The maximum number
/// of tuples that can be held is given by the `capacity` argument.
/// An optional `load_factor` may be given to control sizing of the
/// hash tables used to enforce identity constraints for the tuples contained
/// in the variable. <1> An initial tuple population may be specified by
/// the `initial_population` slice.
```

```

pub fn RelVar(
    comptime Heading: type,
    comptime identifiers: []const []const RelTuple(Heading).AttributeId,
    comptime capacity: RelationCapacity,
    comptime load_factor: ?Loading, ①
    comptime initial_population: []const struct { @Type(.enum_literal), Heading },
) type {
    validateRelVarHeading(Heading);
    validateIdentifiers(Heading, identifiers);

    return struct {
        const Self = @This();

        <<variable variables>>
        <<variable declarations>>
        <<variable public functions>>
        <<variable private functions>>
    };
}

```

① The Loading type is discussed in the next section.

The heading for a RelVar must be a valid heading for a RelValue or RelTuple. However, for relation variables we add one more restriction. Relation variables are not allowed to have relation-valued attributes. There is nothing technically out-of-bounds for a relation variable to have a relation-valued attribute. We have already seen Relation Values that contain relation-valued attributes. But for this usage where we are encoding application logic in data, a relation-valued attribute distorts intent and is indicative of an analysis problem. A tuple-valued attribute is allowed since tuple values are a way to collect several data items together in a form that can be tested for equality and accessed individually. It is not usual to have tuple-valued attributes in the heading of a relation variable.

```

<<relvar module private functions>>=
/// The `validateRelVarHeading` function declares a compiler error if it
/// finds a relation-valued attribute in the `Heading`.
fn validateRelVarHeading(
    comptime Heading: type,
) void {
    inline for (meta.fields(Heading)) |heading_field| {
        if (comptime relvalue.isRelValueType(heading_field.type)) {
            @compileError(
                "relation-valued attributes are not allowed in RelVar headings: '" ++
                heading_field.name ++ "'"
            );
        } else if (comptime reltuple.isRelTupleType(heading_field.type)) { ①
            validateRelVarHeading(heading_field.type);
        }
    }
}

```

- ① Check recursively for any attributes that are relational tuples. A Relation Tuple can hold relation-valued attributes.

The rules for minimal identifiers for a relation variable are enforced at `comptime` to give earlier error detection and better error messages.

```
<<relvar module private functions>>=
/// The `validateIdentifiers` function declares a compiler error if any of
/// the following conditions about the `identifiers` are true:
///
/// 1. The `identifiers` argument has a length of zero.
/// 2. Any identifier that is part of `identifiers` has a length of zero or
///    contains duplicate attribute ids.
/// 3. The attributes of any two identifiers from the `identifiers` slice
///    cannot be a subset of each other.
fn validateIdentifiers(
    comptime Heading: type,
    comptime identifiers: []const []const RelTuple(Heading).AttributeId,
) void {
    // At least one identifier is required.
    if (identifiers.len == 0)
        @compileError("RelVars must specify at least one identifier");

    // Build a set of attribute ids for each identifier. These sets are used to
    // enforce the rules about identifiers.
    comptime var identifiersAttrs: [identifiers.len]
        enums.EnumSet(RelTuple(Heading).AttributeId) = @splat(.initEmpty());

    // An identifier may not be empty.
    inline for (identifiers, 0..) |identifier, identifier_index| {
        if (identifier.len == 0)
            @compileError("RelVar identifiers must have at least one attribute");

        // An identifier may not contain duplicate attributes.
        comptime var attr_set = &identifiersAttrs[identifier_index];
        inline for (identifier) |attr_id| {
            if (attr_set.contains(attr_id)) {
                @compileError(
                    "duplicate attribute in identifier: '" ++
                    @tagName(attr_id) ++
                    "'",
                );
            } else {
                attr_set.insert(attr_id);
            }
        }
    }

    // Identifiers may not be subsets of each other. The design here
    // is to pair-wise compare all the identifiers that were accumulated
    // in the `identifiersAttrs` variable.
    comptime var idThis: usize = 0;
```

```

inline while (id_this < (identifiers.len - 1)) : (id_this += 1) {
    comptime var id_next: usize = id_this + 1;
    inline while (id_next < identifiers.len) : (id_next += 1) {
        if (identifiersAttrs[id_this].subsetOf(identifiersAttrs[id_next])) {
            @compileError("identifiers may not be subsets of each other");
        }
    }
}
}

```

As with Relation Values, it is convenient to make a number of declarations for symbolic shorthand.

```

<<variable declarations>>=
/// The base structure of a tuple for this relation variable.
pub const TupleBase = Heading;
/// The type of a tuple for this relation variable.
pub const Tuple = RelTuple(TupleBase);
/// The enumerated type representing tuple attributes.
pub const AttributeId = Tuple.AttributeId;
/// Function to return the type of an attribute.
pub const AttributeType = Tuple.AttributeType;
/// The specific type of this incarnation of a relation value.
pub const RelValueType = RelValue(TupleBase);
/// The capacity of the relation variable.
pub const relvar_capacity = capacity;
/// The identifiers of the relation variable;
pub const relvar_identifiers = identifiers;

```

Relation Values were limited in their maximum size. The type used to limit the size of Relation Values is also used for Relation Variables.

```

<<relvar module declarations>>=
/// The RelationCapacity type from the relation value module is used
/// to limit the capacity of relation variables.
pub const RelationCapacity = relvalue.RelationCapacity;

```

The RelationCapacity type is used to set a limit on how big the relation variable array can be. We consider 65,536 entries for an array to be sufficient for the use cases envisioned. If that is too few for your application, then this module is probably not the best choice. However, the type is parameterized and can be made larger.

As we did with relation tuples and relation values, we define an `is_relvar` field to make it easy to identify structures as being a RelVar.

```

<<variable declarations>>=
/// A field used to easily determine if a given `struct` is a Relation Variable.
const is_relvar: void = {};

```

A simple function is provided by the module to test types for being a RelVar.

```

<<relvar module public functions>>=
/// The `isRelVarType` function returns `true` if the type given by the
/// `RelVarType` argument can be identified as resulting from an invocation
/// of the `RelVar` type function.
pub fn isRelVarType(
    comptime RelVarType: type,
) bool {
    return @typeInfo(RelVarType) == .@"struct" and @hasDecl(RelVarType, "is_relvar");
}

```

Relation Variable Storage Management

The tuples in a relation variable may be created or deleted during the program run time. Unlike relation values, the relation tuples contained in a relation variable are mutable. Although the maximum number of tuples in a given relation variable are fixed at compile time, empty storage from deleted elements is reused. This requires some additional bookkeeping. First, we track the allocation status of each tuple.

```

<<variable declarations>>=
/// The encoding of the allocation status for Relation Variable storage elements.
const AllocationStatus = enum {
    free,           // The element is not allocated.
    reserved,       // The element is set aside and is not subject to allocation.
    allocated,      // The element is allocated and active.
};

```

Notice that the status encodes a notion of `reserved`. This accommodates operations that need to allocate and store values in a tuple, but are not yet ready to include the tuple into universe of active use. If an element is `reserved`, then some time later it must be *activated* before it can be used. In the intervening time, we are assured that the element is not allocated again.

The following state diagram shows the allowed transitions in the status of an array element.

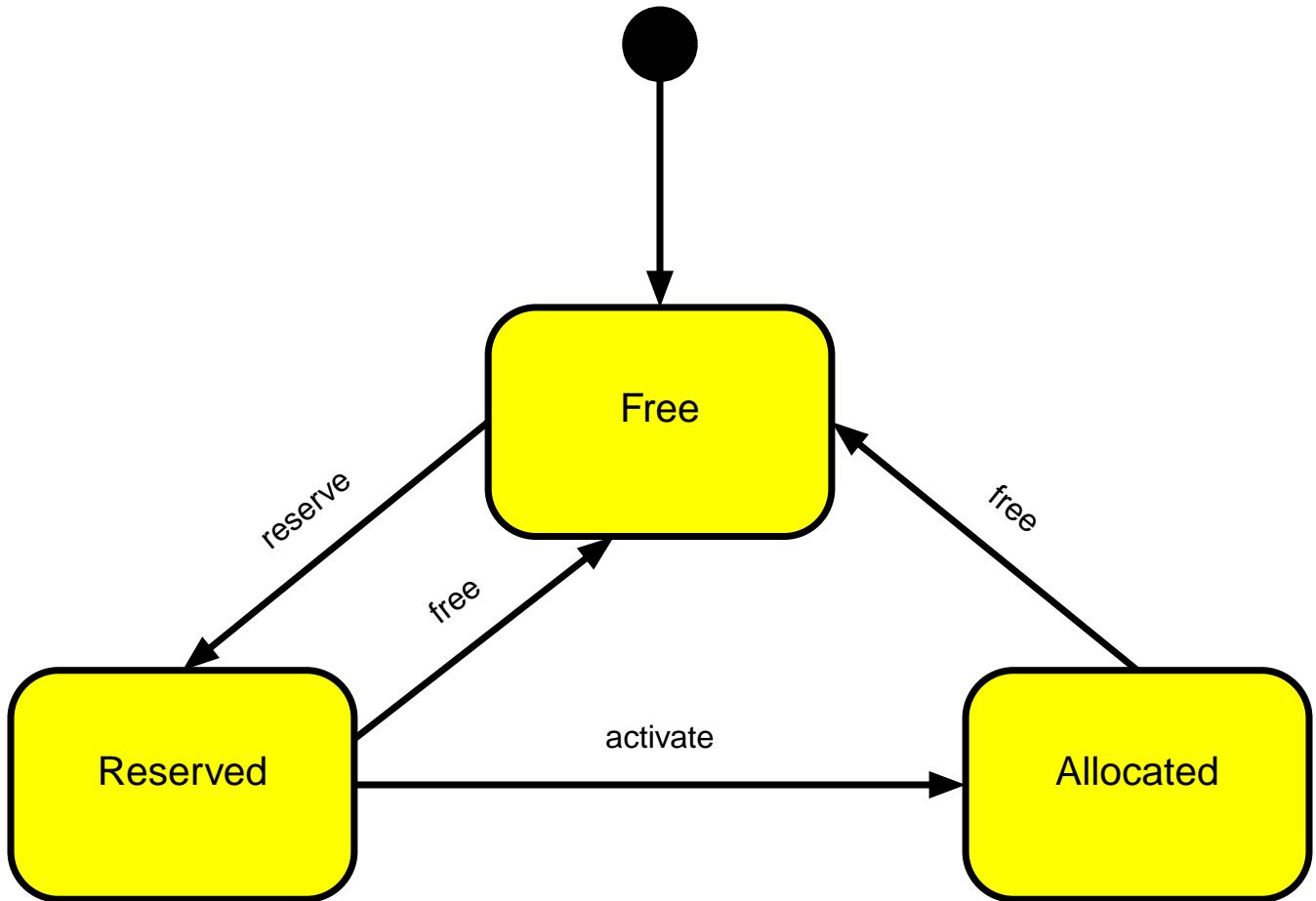


Figure 31. State Diagram for a Relvar Tuple Storage Element

The second consideration for managing relation variable storage arises from two considerations.

1. Storage from deleted tuples must be reused, and
2. It is desirable to have a *reference* to a tuple in a relation variable.

To satisfy these requirements, we store generation information about each tuple slot as a means of tracking the validity of references. References to relation variable tuples are discussed in more detail [below](#).

```

<<relvar module declarations>>=
/// The data type which records the generation usage of a storage slot.
pub const GenerationNumber = u8;
pub const undefined_generation = @as(GenerationNumber, 0);

```

```

<<relvar module declarations>>=
fn nextGeneration(
    gen: *GenerationNumber,
) void {
    gen.* += 1;
    if (gen.* == undefined_generation) gen.* += 1;
}

```

```
}
```

To manage each RelVar storage element, the design records the following information:

- A storage slot for the tuple.
- The allocation status of the tuple storage slot.
- The generation number of the tuple storage slot.

```
<<variable declarations>>=
const RelVarStorageElement = struct {
    tuple: Tuple,
    allocation: AllocationStatus,
    generation: GenerationNumber,
};
```

The memory for the tuple storage is allocated as a container variable of the namespace generated by RelVar.

```
<<variable variables>>=
var storage: [capacity]RelVarStorageElement = initPopulation(initial_population);
```

The final piece required to manage relation variable storage is to keep a notion of a *universe* of the relation variable. For our purposes, the universe is the set of all tuple storage slots that are currently allocated and, consequently, in use. We could compute that set upon demand by scanning the storage array examining the allocation of each slot. Because the universe of allocated tuple storage slots is used in my roles and to save the computation, we store the universe separately being scrupulous to keep it properly updated. We can track the allocation of slots using a bit-set whose size is the same as the capacity of the RelVar. These bit sets are of type, *TupleRefSet*, and this type is explained further [below](#).

```
<<variable variables>>=
/// A bit set that contains all the allocated instance slots in the relation variable.
pub var universe: TupleRefSet = initUniverse();
```

To accommodate an initial tuple population, we initialize the universe knowing the number of tuples in the initial population. Initial tuple populations are discussed in the next section.

```
<<variable private functions>>=
/// The `initUniverse` function return a `TupleRefSet` value that is set according
/// to the number of tuple in the initial instance population.
/// _N.B._ this function has detailed knowledge of the manner in which relation variable
/// storage slots are allocated.
fn initUniverse() TupleRefSet {
    var initial_universe: TupleRefSet = .initWithCapacity(initial_population.len);
    initial_universe.setRangeValue({ .start = 0, .end = initial_population.len }, true);
    return initial_universe;
}
```

Initial Tuple Population

An *initial tuple population* may be specified for a `RelVar`. This allows `comptime` generation of the values for the storage elements of a tuple. The intent is to try to minimize initialization code since for long-running applications initialization code executes only once but still occupies program memory space.

Each initial instances gives both an enumeration literal and a tuple base value. The enumeration literal serves as a convenient moniker to reference the tuple in other contexts. For example, the enumeration literal can be used to `comptime` initialize an association of the instance.

```
<<variable declarations>>=
/// The structure of an initial instance.
pub const InitialInstance = struct {
    @Type(.enum_literal),
    TupleBase,
};
```

```
<<variable private functions>>=
/// The `initPopulation` function sets initial values for the tuples in a
/// `RelVar` based on the slice of initial instances given by, `pop`.
fn initPopulation(
    comptime pop: []const InitialInstance,
) [capacity]RelVarStorageElement {
    var tuple_pop: [capacity]RelVarStorageElement = undefined;

    // N.B. this code simply saves the population data into the correct
    // storage locations. It is required to invoke the `init` function
    // to complete the indexing of the tuples and enforcement of the
    // identity constraints. The current implementation of the
    // tuple identifier indices cannot be executed at compile time.
    inline for (pop, 0..) |inst, inst_index| {
        tuple_pop[inst_index] = .{
            .tuple = Tuple.create(inst@"1"),
            .allocation = .allocated,
            .generation = 1,
        };
    }

    return tuple_pop;
}
```

Since the enumeration literal for an initial instance is only available at compile time, we create an enumeration data type with the same tags so that the instances can be referenced at runtime.

```
<<variable declarations>>=
pub const InitialInstanceId = TagInitialPopulation(initial_population);
```

```
<<variable private functions>>=
fn TagInitialPopulation(
    comptime pop: []const InitialInstance,
```

```

) type {
    var pop_tags: []const EnumField = &[_]EnumField{};

    inline for (pop, 0..) |inst, inst_index| {
        pop_tags = pop_tags ++ &[_]EnumField{{
            .name = @tagName(inst.@"0"),
            .value = inst_index,
        }};
    }

    return @Type(.{
        .@"_enum" = .{
            .tag_type = comptime.math.IntFittingRange(0, pop.len),
            .fields = pop_tags,
            .decls = &.{},
            .is_exhaustive = true,
        },
    });
}

```

At this time, it is not possible to initialize all aspects of a `RelVar` at compile time. Specifically, the computation of indices for the tuple identifiers uses memory pointers that cannot be resolved for run time usage. This is an area for further investigation, but for now it is necessary to invoke the following function immediately after invoking `RelVar` to complete the necessary steps for an initial instance population.

```

<<variable public functions>>=
/// The `init` function completes the necessary initialization of a
/// `RelVar` namespace and must be invoked
/// before any attempt to use the relation variable namespace.
pub fn init() Self {
    // The only task is to create the indices for the identifiers from
    // the initial population.
    var iter = universe.iterator(.{});
    while (iter.next() |tuple_ref| {
        const tuple_index = tuple_ref(tuple_id);
        for (&indices) |*index| {
            const inserted = index.insert(&storage[tuple_index].tuple);
            if (!inserted) {
                panic("failed to insert tuple from initial population:\n{any}", .{
                    storage[tuple_index].tuple
                });
            }
        }
    })
    return .{};
}

```

Relation Variable Errors

There are a number of errors that can arise from the operations. We define an error set for them.

```
/// The list of errors that may be returned by `RelVar` functions.  
<<relvar module declarations>>=  
pub const RelVarError = error{  
    OutOfMemory,           // All array elements are allocated or reserved.  
    AllocationMismatch,   // Unexpected allocation status for operation.  
    DuplicateTuple,       // Attempt to insert duplicate tuple.  
    GenerationMismatch,  // Reference to tuple storage of a different reuse generation.  
    NoSuchTuple,          // Unable to find the requested tuple.  
};
```

Core Storage Element Functions

In this section, the foundational storage functions are described. These functions derive directly from the previous state diagram for Relvar storage element. Note that the interface for these functions is in terms of the index into the relation variable storage array. Those indices are then combined as tuple references later.

Since all allocated elements go through the `.reserved` status, the `reserve` function does the heavy lifting to find an element with a `.free` status. The strategy is to iterate across the universe bit set looking for unset bits.

Reserve

The first step in creating a relation variable tuple is to reserve a slot in the variable storage. This action is done in a multi-step manner to support the capability of creating a unique identifier that can be included in tuple value.

```
<<variable private functions>>=  
/// The `reserve` function searches for an empty tuple element storage  
/// slot in a relation variable and, upon finding one, marks the slot  
/// as .reserved. Reserving a slot gives the slot a new generation  
/// number. Failure to find a .free slot results in an OutOfMemory  
/// error. Upon success, this function returns the element index of  
/// the newly reserved tuple slot.  
fn reserve() RelVarError!RelationCapacity {  
    // Operate directly on the bit set that contains the universe.  
    // This avoids making a TupleRef to reference a .free tuple.  
    // This is the only function which looks for unset bits in the  
    // universe and so the allocation status is indeterminate.  
    var univ_iter = universe.tset.iterator({ .kind = .unset });  
    while (univ_iter.next()) |element_index| {  
        const element = &storage[element_index];  
        if (element.allocation == .free) { ①  
            element.allocation = .reserved;
```

```

        nextGeneration(&element.generation);
        return @intCast(element_index);
    }
}
return RelVarError.OutOfMemory;
}

```

- ① The test for `.free` is necessary since storage slots with a status of `.reserved` are not recorded in the universe bit set.

Update Reserved Element

Once a slot is reserved, it is necessary to store a value in it. Typically, a call to `reserve` is followed immediately by a call to `updateReserved`.

```

<<variable private functions>>=
/// The `updateReserved` function stores `tuple` in a previously reserved storage
/// slot as returned from a successful invocation of `reserve`.
fn updateReserved(
    element_index: RelationCapacity,
    tuple: TupleBase,
) RelVarError!void {
    const element = &storage[element_index];
    if (element.allocation != .reserved) return RelVarError.AllocationMismatch;
    element(tuple = Tuple.create(tuple));
}

```

As a convenience function, for those cases where the tuple value is fully determined at the time the `reserve` function is invoked, a helper function is available.

```

<<variable private functions>>=
/// The `reserveAndUpdate` function is a convenience function to reserve
/// a relation variable tuple storage slot and update the slot with
/// the value of `tuple`. This is a common case that can be used when
/// the value of `tuple` is completely determined at the time of the
/// reservation.
fn reserveAndUpdate(
    tuple: TupleBase,
) RelVarError!RelationCapacity {
    const element_index = try reserve();
    updateReserved(element_index, tuple) catch unreachable; ①
    return element_index;
}

```

- ① Since we have just reserved the storage slot and have had no intervening computation, `updateReserved` will always succeed.

Activate

A tuple element must be activated before it can be used. Activation creates entries in the indices of the relation variable, verifying the uniqueness constraint on tuple values, and inserts the tuple in the universe set so that it will appear when operating the relation variable tuples.

```
<<variable private functions>>=
/// The `activate` function sets the status of the reserved tuple element,
/// `element_index`, to `allocated` and inserts the tuple present in the
/// storage element into the identifier indices of the relation variable.
/// It is an error for `element_index` to refer to a slot that does not
/// have a `reserved` allocation. It is an error if the tuple value
/// currently stored in the relation variable storage slot fails to be
/// inserted. Any index insertion failure results in the storage slot
/// being set to `free`. After successful completion, the storage slot
/// contains a valid tuple that is guaranteed to be unique and appears
/// in the universe of tuples for the relation variable.

fn activate(
    element_index: RelationCapacity,
) RelVarError!void {
    const element = &storage[element_index];

    if (element.allocation != .reserved) return RelVarError.AllocationMismatch;
    element.allocation = .allocated;
    for (&indices, 0..) /*index, index_num| {
        const inserted = index.insert(&element.tuple); ①
        if (!inserted) {
            for (0..index_num) |inserted_idx| { ②
                _ = indices[inserted_idx].remove(&element.tuple);
            }
            element.allocation = .free;
            return RelVarError.DuplicateTuple;
        }
    }
    universe.set(element_index);
}
```

① Tuples are not indexed until the storage slot is activated.

② If insertion into one of the indices fails, we must remove all the previously indexed elements from their respective index. Note we cannot blindly remove the tuple from all the identification indices since that would also remove the existing tuple which was not a duplicate. So activating is an all-or-nothing operation with respect to the indices and failure frees the offending tuple itself.

Free

```
<<variable private functions>>=
/// The `free` function sets the status of a relation variable storage
```

```

/// to be `free`, removes the tuple indicated by `element_index` from
/// the indices of the relation variable, and from the universe of
/// allocated tuples.
fn free(
    element_index: RelationCapacity,
) void {
    const element = &storage[element_index];
    if (element.allocation == .free) @panic("attempt to double free relvar tuple");
    element.allocation = .free;
    for (&indices) |*index| {
        _ = index.remove(&element.tuple); ①
    }
    universe.unset(element_index);
}

```

- ① Ignore the fact that the removal might fail. One way or the other, the tuple no longer appears in the index and its storage slot is marked as `free`.

Tests

```

<<relvar module tests>>=
test RelVar {
    // In words, `RVar` is a Relation Variable type with whose heading
    // contains three attributes: name, length, and weight. The RelVar
    // has two identifiers. The first identifier consists of the
    // .name and .length attributes. The second identifier consists
    // of the .name and .weight attributes. Note there is a non-empty
    // intersection between the attributes of the two identifiers.
    // The maximum number of tuples the variable can hold is 10.
    // No hash table load factor is given, so the default is used.
    // The initial tuple population is empty.
    const RVar = RelVar(
        struct { // heading
            name: []const u8,
            length: usize,
            weight: f32,
        },
        &.{ // identifiers
            &{.name, .length}, // id 0
            &{.name, .weight}, // id 1
        },
        10, // capacity
        null, // load factor
        &{}, // initial population
    );

    // rvar_1 is an ordinary Zig variable of type, RVar.
    const rvar_1 = RVar.init();
    try testing.expectEqual(@as(usize, 0), rvar_1.cardinality());
}

```

```

// Because RVar is a namespace struct which contains no fields,
// Zig variables of RVar type occupy no space. All the space required
// to store the tuples of the relation variable is part of a
// container variable in RVar.
try testing.expectEqual(@as(usize, 0), @sizeOf(RVar));

const created_1 = try RVar.reserveAndUpdate(.{
    .name = "foo",
    .length = 20,
    .weight = 100.0,
});
try testing.expectEqual(@as(usize, 0), rvar_1.cardinality());
try RVar.activate(created_1);
try testing.expectEqual(@as(usize, 1), rvar_1.cardinality());

// Within the same scope, creating a second variable of type RVar
// results in access to the same relation variable data since
// the data stored in a container variable of the class.
// So adding a tuple with the same attribute values causes a
// duplicate tuple error.
const rvar_2: RVar = {};
const created_2 = try RVar.reserveAndUpdate(.{
    .name = "foo",
    .length = 20,
    .weight = 100.0,
});
try testing.expectError(RelVarError.DuplicateTuple, RVar.activate(created_2));
try testing.expectEqual(.free, RVar.storage[created_2].allocation);
// And the original tuple added via `rvar_1` remains.
try testing.expectEqual(@as(usize, 1), rvar_2.cardinality());

RVar.free(created_1);
try testing.expectEqual(@as(usize, 0), rvar_1.cardinality());
try testing.expectEqual(@as(usize, 0), rvar_2.cardinality());
}

```

Relation Variable Indices

For Relation Values, all the attribute values of each Relation Tuple contributed to the identity of the tuples. For Relation Variables, we use minimal identifiers which means that one or more subsets of attributes is used to identify a tuple. Placing indices on the identifiers improves the search time when enforcing identity constraints and when selecting a tuple based on identifying attribute values. Since we are only concerned about finding tuples with equal identifiers, we create a [hash table index](#) for each identifier.

For each identifier, a hash table index is provided to ensure uniqueness of the tuples with respect to each identifier. This index is discussed [below](#). For now, we can define an index type that is suitable for enforcing tuple uniqueness.

<<variable declarations>>

```
const Index = RelVarIndex(capacity, load_factor);
```

The indices used for identification are held in an array. The default value of the `indices` field creates an empty index. The hash and equal functions are set to functions generated at compiletime.

```
<<variable variables>>=
var indices: [identifiers.len]Index = initIndices();
```

```
<<variable private functions>>=
fn initIndices() [identifiers.len]Index {
    var idxs: [identifiers.len]Index = undefined;
    for (identifiers, 0...) |identifier, id_index| {
        idxs[id_index] = .{
            .hash_fn = IndexHashFunction(identifier), ①
            .eq_fn = IndexEqFn(identifier),
        };
    }
    return idxs;
}
```

① The `hash_fn` and `eq_fn` are the only fields of the index that do not have default initializers.

Overview

The Zig standard library contains a wide array of hashed data structures. The use case here is much simpler than those supported by the standard library and we have specifically tailored the hashing scheme for this usage. Using a specific hash table scheme allows us to take advantage of the knowledge that the upper bound on the size of the relation variable storage is fixed at compile time.

The following considerations are important for this use case.

- Sizes of tuple sets held in relation variables are bound at compile time and so memory for hash table storage can also be sized for worst case usage at compile time.
- Searching for an array element is based strictly upon identity is sufficient. This index does not support any other types of searching, e.g. searching on a range of values.
- A relation variable must have at least one *identifier*. An identifier consists of one or more tuple attributes. Multiple indices may be created on a given tuple if it has multiple identifiers. For example, multiple indices can be used to create a bijective function that is based on identifier values.

There are many different hashing techniques and an important consideration in hashing is how to handle collisions. Since we don't change the storage size dynamically, some self-contained form of collision handling is most convenient. This suggests using an [open addressing](#) approach for collision resolution rather than a chaining approach.

[Robin Hood hashing](#) is a collision resolution technique that uses open addressing and has the further twist of moving keys around based on the number of probes used to find a hash bucket. It is a relatively simple [algorithm](#) with predictable worst case performance, and it can deal effectively with removing entries from the hash table without special markings or other meta-data.

The following section shows an implementation of relation variable tuple indexing using Robin Hood hashing implemented in Zig. We do not explain how the algorithm works outside of the given Zig

implementation. The Internet contains many examples and the previously referenced links should be consulted if additional background is desired.

Implementation

We define some convenient data types to hold values of interest to the index.

The SequenceCount type is used to set a limit on how big the hash table can be and to store the probe counts used by the Robin Hood hash algorithm. We have already limited the maximum tuple set size for relation variables, so we use the same type to limit counts and storage for the indices.

```
<<relvar module declarations>>=
pub const SequenceCount = RelationCapacity;
```

For hash tables, there is a space / speed trade off when it comes to collision probability. A hash table larger than the indexed array reduces the probability of collisions and, consequently, the number of additional comparisons that must be made when a collision occurs. The Loading type holds a load factor value. Robin Hood hashing works reasonably even with load factors of 100%. Depending upon the frequency of search, additional index space used to lower the collision possibilities may not be worth it. Experience and anecdote advises load factors in the 80% to 90% range. Measurement under realistic work loads is the only sure way to choose a load factor value if the trade-off is important to consider.

```
<<relvar module declarations>>=
pub const Loading = math.IntFittingRange(0, 100);
```

```
<<variable private functions>>=
/// The `RelVarIndex` function is a type function which generates a data
/// type suitable for creating an index of an array whose element type
/// correspond to that type used store a tuple set in a Relation Variable.
/// The `indx_capacity` gives the maximum number of tuple elements that
/// can be handled. The `indx_load_factor` is an optional percentage
/// which specifies the maximum number of buckets in the hash table.
/// This argument allows for having a greater number of buckets to
/// reduce the probability of hash collisions. The `indx_load_factor`
/// must be greater than zero. A `indx_load_factor` of 100 indicates
/// that the number of buckets in the hash table is to be the same as
/// the number of elements in the indexed array. A `indx_load_factor`
/// of 50 indicates that twice as many hash table buckets are to be
/// used as there are elements in the array. If the `index_load_factor`
/// is given as `null`, then a default loading of 90% is used.
fn RelVarIndex(
    comptime indx_capacity: SequenceCount,
    comptime indx_load_factor: ?Loading,
) type {
    const default_load_factor: usize = 90;
```

```

const load = idx_load_factor orelse default_load_factor;
if (load > 100)
    @compileError("load factor must be less than or equal to 100%: got '" ++
        fmt.comptimePrint("{d}", .{load}) ++ "%'");
if (load == 0)
    @compileError("load factor cannot be 0%");

return struct {
    const IndexSelf = @This();

    <<index fields>>
    <<index declarations>>
    <<index public functions>>
    <<index private functions>>
};

}

```

The total number of hash table buckets is adjusted for the load factor.

```

<<index declarations>>=
const bucket_count = math.divCeil(SequenceCount, idx_capacity * 100, load)
    catch |err| @compileError("failed to calculate bucket count: '" ++
        @errorName(err) ++ "'");

```

Each bucket in the hash table stores an index into the tuple storage for the relation variable and the number of probing sequences it took to find the bucket when inserting the element reference into the hash table. The element pointer is optional and `null` is used to indicate an unoccupied bucket.

```

<<index declarations>>=
const Bucket = struct {
    value: ?*Tuple = null,
    sequence: SequenceCount = 0,
};

```

The index is an array of Bucket elements and a count of the number of buckets that are occupied. The occupied count is convenient to ensure that we don't launch off looking for a bucket when one will never be found. In this case, is it worth the small price to maintain the redundant data.

```

<<index fields>>=
buckets: [bucket_count]Bucket = @splat(.{}),
occupied: SequenceCount = 0,

```

Operating the hash table implies that we need a function to compute the hash and another function to compare elements. This design holds those function pointers as fields of the RelVarIndex structure. It would be tempting to include these functions as part of the comptime arguments for RelVarIndex. However, we want to store the hash tables for the identifiers in an array. That

requires that the generated types of the hash tables be the same. Since the `hash_fn` and `eql_fn` compute the hash and equality over different tuple attributes, the types these functions across multiple identifiers is *not* the same. They have different types depending upon the attributes which form the identifier. So, if the required hash and equality functions are part of the type for the index, then it is not possible to hold an array of them as part of the `RelVar` structure. The solution is to use fields in the index and place the responsibility for initializing the function pointers on the initialization of a `RelVar`. Another alternative would be to hold the hash indices as an array of tagged unions which would allow the array elements to be the same type.

```
<<index fields>>=
hash_fn: *const fn(tuple: *const Tuple) u32,
eql_fn: *const fn(one: *const Tuple, other: *const Tuple) bool,
```

To make initializing the above fields more convenient, we use a `comptime` function to generate the hash and equality functions that are specific to the attributes that form the identifier for the relation variable. The Zig standard library has a large selection of hashes that satisfy a wide range of requirements. We only need simple non-cryptographic hash and use the “1a” variation of the [Fowler-Noll-Vo hash function](#). This hash function is available from the Zig standard library, `hash`.

```
<<variable private functions>>=
fn IndexHashFunction(
    comptime attr_ids: []const AttributeId,
) *const fn(*const Tuple) u32 {
    return struct {
        fn hash(
            tuple: *const Tuple,
        ) u32 {
            var hasher = std.hash.Fnv1a_32.init();
            inline for (attr_ids) |attr_id| {
                hasher.update(mem.asBytes(&tuple.extract(attr_id)));
            }
            return hasher.final();
        }
    }.hash ;
}
```

Attribute equality is provided by the `reltuple` module and we use it here to generate a function that tests two array elements for equality.

```
<<relvar module declarations>>=
const attributeEql = reltuple.attributeEql;
```

```
<<variable private functions>>=
fn IndexEqlFunction(
    comptime attr_ids: []const AttributeId,
) *const fn (*const Tuple, *const Tuple) bool {
    return struct {
```

```

fn eql(
    one: *const Tuple,
    other: *const Tuple,
) bool {
    inline for (attr_ids) |attr_id| {
        if (!attributeEql(❶
            one.extract(attr_id),
            other.extract(attr_id),
        )) {
            return false;
        }
    }
    return true;
}
}.eql;
}

```

- ❶ Note it is the tuple, not the element, which is tested for equality.

Initialization of a RelVarIndex requires only the two functions described previously since the other fields have default values.

```

<<index public functions>>=
pub fn init(
    hash_fn: *const fn(*const Tuple) u32,
    eql_fn: *const fn(*const Tuple, *const Tuple) bool,
) IndexSelf {
    return .{
        .hash_fn = hash_fn,
        .eql_fn = eql_fn,
    };
}

```

Probing

A common operation required by the three primary hash table functions is the notion of *probing*. In Robin Hood hashing, the number of comparisons required to find an open addressing slot is used to determine the boundaries of an ordered sequence chain. Probing is the act of determining the status of a key.

After hashing the key, the probing starts at the index given by the hash function. There are three possible outcomes to probing the index with a key:

1. An empty slot is found. This is indicated by the `value` field of the bucket being `null`. An empty slot is potentially the start of a probe sequence.
2. A match is found, *i.e.* a non-empty slot referring to a tuple whose identifying attributes match those of the key.
3. The end of a probe chain is found. Since we are keeping the number of probes it took to insert an entry into the index, if we have probed from a starting place and the number of probe attempts is now greater than the sequence number found in an index entry, then we have reached the end of the probe sequence. For insertion, we will “steal” the entry and move its current occupant to a different place. For removal, we have reached the end of the “back

shuffle" that is required when an index entry is removed.

```
<<index declarations>>=
const ProbeStatus = enum {
    empty,
    found,
    end,
};
```

The probe of the index does not modify the index array elements. Rather, the status of the probe is returned. Depending upon the operation being performed, the result of the probe is used to modify the index elements or request additional probes of the index.

```
<<index declarations>>=
const ProbeResult = struct {
    // which status the probe found
    status: ProbeStatus,
    // the index of the bucket where probing stopped
    bucket_index: SequenceCount,
    // the number of probe attempts required to determine the status,
    // equivalent to the number of key comparisons required
    probe_count: SequenceCount,
};
```

Bucket addition is, of course, modulo the `bucket_count`.

```
<<index private functions>>=
fn nextBucket(
    bucket_index: SequenceCount,
) SequenceCount {
    return (bucket_index +% 1) % bucket_count;
}
```

```
<<index public functions>>=
/// The `probe` function traverses the index array to determine the status of
/// the `key` with respect to the index and returns that status.
/// Probing stops at the first situation where the status can be determined to be
///
/// 1. an empty bucket,
/// 2. a match of `key`, or
/// 3. the end of a probing sequence.
///
/// The arguments are:
///
```

```

/// - `index_start`
/// The bucket index in the hash table where probing is to begin.
/// If `index_start` is `null`,
/// then this is an initial probe of the hash table and the starting bucket
/// is determined by the value of the hash of the `key` identifying attributes.
/// Otherwise, this is a secondary probe being used complete one of the
/// operations on the index.
///
/// - `probe_start`
/// The starting value of the probe count.
/// If `index_start` is `null`, `probe_start` must be 0.
/// Otherwise, the counter used to track the number of probes, _i.e._ the number
/// of key comparisons, is set to `probe_start`.
///
/// - `key`
/// A pointer to a memory object of type, `T`. The fields of the structure that
/// are used as an identifiers must be set to proper values.
/// Other fields of `T` may be left undefined.
/// A key comparison determines if the key identifier fields match the identifier
/// fields of the elements of the indexed array.

fn probe(
    self: IndexSelf,
    index_start: ?SequenceCount,
    probe_start: SequenceCount,
    key: *const Tuple,
) ProbeResult {
    var result: ProbeResult = .{
        .status = undefined,
        .bucket_index = index_start orelse
            @intCast(self.hash_fn(key) % @as(u32, @intCast(bucket_count))), ①
        .probe_count = probe_start,
    };

    if (index_start == null) {
        assert(probe_start == 0);
        result.probe_count = 0;
    }
    var bucket = &self.buckets[result.bucket_index];

    assert(result.probe_count < self.buckets.len);

    while (result.probe_count < self.buckets.len) {
        if (bucket.value == null) {
            result.status = .empty;
            break;
        } else if (self.eql_fn(bucket.value.?, key)) {
            result.status = .found;
            break;
        } else if (result.probe_count > bucket.sequence) {
            result.status = .end;
            break;
        }
    }
}

```

```

    // N.B. no else clause. Continue looking for one of the
    // three conditions that terminates the probe.
    result.probe_count += 1;
    result.bucket_index = IndexSelf.nextBucket(result.bucket_index);
    bucket = &self.buckets[result.bucket_index];
}

return result;
}

```

- ① We use the *lazy mod mapping method* to limit the 32-bit hashed value to the required range.

Insert

```

<<index public functions>>
/// The `insert` function inserts `value` into a hash table. It returns
/// `true` if the insertion succeeds and `false` otherwise. The `value`
/// argument is a pointer to an array element. Upon successful insertion,
/// this pointer is placed in the hash table bucket of the array index.
/// _N.B._ the `value` argument must point to the object being indexed
/// and some care must be used when obtaining the object reference.
/// Pointers to automatic variables _must_ be avoided. The indexing
/// code does no management of the object referenced by `value` and
/// assumes no ownership of the memory referenced by `value`.
pub fn insert(
    self: *IndexSelf,
    value: *Tuple,
) bool {
    if (self.occupied >= self.buckets.len) {
        return false;
    }

    var insert_value = value;
    var probe_result = self.probe(null, 0, insert_value);

    while (true) {
        const bucket = &self.buckets[probe_result.bucket_index];
        switch (probe_result.status) {
            .empty => {
                bucket.sequence = probe_result.probe_count;
                bucket.value = insert_value;
                self.occupied += 1;
                return true;
            },
            .found => return false,
            .end => {
                // The bucket is stolen by swapping out the value we are trying to insert
                // for the one in the bucket.
                const occupant_value = bucket.value.?;
                bucket.value = insert_value;
                insert_value = occupant_value;
            }
        }
    }
}

```

```

        // The steal is completed by swapping the bucket sequence count with the
        // probe count.
        const occupant_sequence = bucket.sequence;
        bucket.sequence = probe_result.probe_count;
            // +1 as we are moving the occupant to the next slot
            // from is initial placement
        const next_probe = occupant_sequence + 1;

        // Continue probing at the next bucket with the stolen value and
        // probe count.
        const next_bucket = IndexSelf.nextBucket(probe_result.bucket_index);
        probe_result = self.probe(next_bucket, next_probe, insert_value);
    },
}
}

```

A couple of things to note:

- If we are fortunate to find an empty bucket on the first attempt, the probe count is zero, i.e. we have not made any key comparisons in the attempt to find an empty slot. For perfect hashing, all the probe counts are zero.
- All successful insertions go through the `.empty` prong and return out of the probing loop there. All the unsuccessful insertions go through the `.found` prong, assuming the index is not full in the beginning. Because the probe chain is sequentially ordered, if we steal a bucket, we are guaranteed that the value is not in the index, i.e. we would have already found the match, and after a “steal” probing continues to find the nearest empty bucket for the displaced value.

Search

```

<<index public functions>>=
/// The `search` function searches for an item in a hash table given a
/// key value which identifies the item. If the search is successful, a
/// reference to the matching array value is returned. Otherwise, `null`
/// is returned. The `key` argument is an array element which is used
/// as the key for the search. Only the `tuple` field of key is used.
pub fn search(
    self: IndexSelf,
    key: *const Tuple,
) ?*Tuple {
    const probe_result = self.probe(null, 0, key);
    return if (probe_result.status == .found)
        self.buckets[probe_result.bucket_index].value
    else
        null;
}

```

Remove

The removal algorithm can be summarized as:

1. Find the entry to remove. If it's not found, we have failed and return null to indicate so.
2. Starting from the found entry, iterate over the remaining probe chain to shuffle entries on the chain backward.
 - a. Look ahead to the next entry. Essentially, we keep a cursor into the probe chain.
 - b. Check if the next entry is the end of the probe chain. A probe chain ends when we find an empty bucket or the beginning of a new probe chain. All buckets with a non-null value continue the probe chain. All buckets with a sequence value of zero, are the start of a new probe chain even if the bucket is occupied.
 - c. If we haven't reached the end of a probe chain, shift the entry back one bucket by assigning the next bucket into the current bucket. Note the sequence number of the current bucket is decremented to account for the fact that the `next_bucket` is now one closer to the beginning of the probe chain.
 - d. Continue probing the next bucket to determine the end of the probe chain. When the next bucket is the end of a chain, we empty out the current bucket to mark the end of the probe chain from which the removal took place.

```

<<index public functions>>
/// The `remove` function removes an item from an array index.
/// If successful, it returns the pointer to the array element originally
/// inserted into the index. Unsuccessful removals, _i.e._ attempting to
/// remove a reference to an array element value that is not identified by
/// `key`, returns `null`. The `key` argument is a reference to the array
/// element to be removed from the index. Only the `tuple` field is used.
pub fn remove(
    self: *IndexSelf,
    key: *const Tuple,
) ?*Tuple {
    const probe_result = self.probe(null, 0, key);

    var removed: ?*Tuple = null;

    if (probe_result.status == .found) {
        var bucket_index = probe_result.bucket_index;
        removed = self.buckets[bucket_index].value;
        self.occupied -= 1;

        var done = false;
        while (!done) {
            var bucket = &self.buckets[bucket_index];
            const next_index = IndexSelf.nextBucket(bucket_index);
            const next_bucket = &self.buckets[next_index];

            if (next_bucket.value == null or next_bucket.sequence == 0) {
                bucket.value = null;
                bucket.sequence = 0;
                done = true;
            } else {
                bucket.value = next_bucket.value;
                bucket.sequence = next_bucket.sequence - 1; ①
            }
        }
    }
}

```

```

        bucket_index = next_index;
    }
}

return removed;
}

```

- ① Moving the bucket value back means it is one step closer to the beginning of the probe sequence.

References to Tuples in a Relation Variable

An identifier for a tuple storage slot is the combination of the index into the storage array which locates the storage slot and the usage generation of the slot which tracks its usage lifetime. Tuple references play an important role to the internal operations on relation variables. In this section, we show the definition of tuple references and their methods.

A tuple reference plays the role in relation variables analogous to the role an ordinary relational tuple plays in a relational value. It gives us a way to access a single tuple value. Many operations on relation variables consider just a single tuple and a tuple reference is a convenient means to handle those operations.

A direct pointer is avoided in favor of the index of the tuple in the relation variable storage array. However, we must account for the lifetime of a tuple reference. Tuple references are used in the execution management portion of ReSEE. We need to detect the circumstance where deleting the tuple invalidates a reference that may have been stored away by the execution mechanisms. To detect any difference in lifetime, each reference includes a *generation number* for the tuple storage element. Each time a tuple storage slot is *reserved*, its associated generation number is changed. Dereferencing the tuple then checks for equality of the generations. References held for longer than the lifetime of the tuple are considered an analysis error and it is the responsibility of the analysis model to ensure reference lifetimes match those of the referenced tuple. Attempting to dereference a tuple with an incorrect generation number creates a panic condition.

```

<<variable declarations>>=
/// The structure of a reference to a relation variable tuple.
pub const TupleRef = struct {
    /// The index into the tuple storage array for this tuple.
    tuple_id: RelationCapacity = 0,
    /// An indication of the reuse of a tuple storage slot. Each time a
    /// storage slot is reserved, the `generation` field of the storage
    /// element is modified. Each time a reference to the tuple is taken
    /// this field records the current generation number of the storage
    /// slot. When a tuple is dereferenced, this field and the current
    /// value in the storage slot must match.
    gen_id: GenerationNumber = undefined_generation,
    <<tuple reference declarations>>
};

```

The following section describe the operations available on TupleRef types.

Initialization

Obtaining a `TupleRef` for a tuple is done primarily by knowing the index of the tuple in the relation variable storage. Each time a reference is initialized, its lifetime generation number is also stored in the reference. Note that is a panic condition to attempt to make reference to a `.free` storage slot.

```
<<tuple reference declarations>>=
/// The `initFromIndex` function returns an initialized `TupleRef` based on
/// the `element_index` into the relation variable storage. It is not allowed
/// to create a tuple reference to a `free` storage slot.
pub fn initFromIndex(
    element_index: usize,
) TupleRef {
    if (storage[element_index].allocation == .free) panic(
        "attempt to create a reference to a free tuple storage element: '{d}'",
        .{element_index},
    );

    return .{
        .tuple_id = @intCast(element_index),
        .gen_id = storage[element_index].generation,
    };
}
```

```
<<tuple reference declarations>>=
/// The `initFromInitialInstance` function returns an initialized `TupleRef` based
/// on the initial instance moniker, `initial_inst`. The value of `initial_inst`
/// is a `comptime` generated enumeration for each initial instance specified when
/// the `RelVar` was created.
pub fn initFromInitialInstance(
    initial_inst: InitialInstanceId,
) TupleRef {
    return TupleRef.initFromIndex(@intFromEnum(initial_inst));
}
```

```
<<tuple reference declarations>>=
pub fn tupleRefSetFromTupleRef(
    tuple_ref: TupleRef,
) TupleRefSet {
    return .initFromTupleRef(tuple_ref);
}
```

Referencing Tuples by Creating

Relational tuple held in relational variable are created one at a time. The functions in this section support creating a tuple in a relation variable and returning a reference to the newly created tuple.

Two different interfaces are provided. One returns an error when things go wrong and the other creates a panic condition. This pattern is used in many of the operations on tuple references. Circumstances of the program semantics determine whether it is better to bubble up an error or to assume everything is “perfect” until demonstrated otherwise.

```

<<tuple reference declarations>>=
/// The `createOrError` function creates a new relation variable tuple based on the
/// `tuple` value and returns a `TupleRef` to the newly created tuple.
/// If the creation fails, a `RelVarError` is returned.
pub fn createOrError(
    tuple: TupleBase,
) RelVarError!TupleRef {
    const element_index = try Self.reserveAndUpdate(tuple);
    try Self.activate(element_index); ①
    return TupleRef.initFromIndex(element_index);
}

```

- ① If the activate fails, it will have backed out all the index entries and freed the tuple storage.

```

<<tuple reference declarations>>=
/// The `create` function creates a new relation variable tuple based on the
/// `tuple` value and returns a `TupleRef` to the newly created tuple.
/// Failing to create the tuple results in a panic condition.
pub fn create(
    tuple: TupleBase,
) TupleRef {
    return TupleRef.createOrError(tuple) catch |err| {
        panic("failed to create relvar tuple, '{s}'", .{@errorName(err)});
    };
}

```

As mentioned in the section of relation variable storage, support for asynchronous creation as a side effect of signaling an event is required to support our domain model. To support creating relvar tuples asynchronously, functions are provided that give finer control over the allocation of tuple storage.

```

<<tuple reference declarations>>=
/// The `reserve` function finds an empty tuple storage slot,
/// updates the storage slot with the `tuple` value, and returns
/// a reference to the new tuple. Any error results in returning
/// a `RelVarError`.
pub fn reserve() RelVarError!TupleRef {
    const element_index = try Self.reserve();
    return TupleRef.initFromIndex(element_index);
}

```

```

<<tuple reference declarations>>=
/// The `updateReserved` function updates the tuple storage slot referenced
/// by `tuple_ref` with the value of `tuple`.
pub fn updateReserved(
    tuple_ref: TupleRef,
    tuple: TupleBase,
) RelVarError!void {
    try Self.updateReserved(tuple_ref.tuple_id, tuple);
}

```

```

<<tuple reference declarations>>=
/// The `reserveAndUpdate` function is a convenience function that reserves
/// a tuple storage slot and immediately update the slots contents with the
/// value of `tuple`. This is useful for those cases where the value in `tuple`
/// does not depend upon the slot where the tuple is stored.
pub fn reserveAndUpdate(
    tuple: TupleBase,
) RelVarError!TupleRef {
    const element_index = try Self.reserveAndUpdate(tuple);
    return TupleRef.initFromIndex(element_index);
}

```

```

<<tuple reference declarations>>=
/// The `activate` function marks the tuple referenced by `tuple_ref` to
/// be allocated, inserts the referenced tuple into the identification
/// indices, and add the tuple to the universe of tuples for the relation variable.
/// Any failure to activate the tuple returns a `RelVarError` value.
pub fn activate(
    tuple_ref: TupleRef,
) RelVarError!void {
    try Self.activate(tuple_ref.tuple_id);
}

```

Dereferencing a Tuple Reference

By analogy to dereferencing a pointer, a tuple reference may also be dereferenced to obtain the tuple value stored in relation variable.

```

<<tuple reference declarations>>=
/// The `dereference` function returns the tuple from the storage slot
/// referenced by the `tuple_ref` argument.
pub fn dereference(
    tuple_ref: TupleRef,
) Tuple {
    tuple_ref.verifyAsNot(.free);
    return storage[tuple_ref.tuple_id].tuple;
}

```

```

<<tuple reference declarations>>=
/// The `readAttribute` attribute function returns the value of the
/// attribute given by `attr_id` from the tuple reference given by `tuple_ref`.
pub fn readAttribute(
    tuple_ref: TupleRef,
    comptime attr_id: AttributeId,
) AttributeType(attr_id) {
    const element_index = tuple_ref.tuple_id;
    assert(storage[element_index].allocation == .allocated);
    return storage[element_index].tuple.extract(attr_id);
}

```

```

<<tuple reference declarations>>=
/// The `updateAttribute` function sets the attribute given by `attr_id` to
/// the value given by `value` for the tuple referenced by `tuple_ref`.
/// The update changes the attribute values in situ.
/// It is an error to attempt to update an identifying attribute.
pub fn updateAttribute(
    tuple_ref: TupleRef,
    comptime attr_id: AttributeId,
    value: AttributeType(attr_id),
) void {
    comptime validateAttrAsNonIdentifying(attr_id);
    const element_index = tuple_ref.tuple_id;
    @field(storage[element_index].tuple.reltuple, @tagName(attr_id)) = value;
}

```

Destroy

Tuples may be deleted from a Relation Variable. The storage management design handles the bookkeeping to reuse a storage slot.

```

<<tuple reference declarations>>=
/// The `destroy` function deletes the tuple given by the `tuple_ref`
/// argument from the given relation variable.
pub fn destroy(
    tuple_ref: TupleRef,
) void {
    tuple_ref.verifyAsNot(.free);
    free(tuple_ref.tuple_id);
}

```

Validating References

Since a tuple reference plays an important role and has some properties of a raw pointer, we must make sure the lifetime of the reference corresponds to the state of the corresponding tuple storage slot. The functions in this section provide the necessary operations to ensure that a `TupleRef` value actually refers to the correct entity.

```

<<tuple reference declarations>>=
/// The `verifyAs` function checks that the allocation status of `tuple_ref`
/// matches the `expected_status` value. Failure to match results in a
/// panic condition.
pub fn verifyAs(
    tuple_ref: TupleRef,
    expected_status: AllocationStatus,
) void {
    const tuple_index = tuple_ref.tuple_id;
    const actual_status = storage[tuple_index].allocation;
    if (expected_status != actual_status) {
        panic("tuple reference verification failed: expected '{any}' != '{any}'",
              expected_status,

```

```

        actual_status,
    });
}
tuple_ref.verifyGeneration();
}

```

```

<<tuple reference declarations>>=
/// The `verifyAsNot` function checks that the allocation status of `tuple_ref`
/// _does not match_ the `disallowed_status` value. Failure to match results in a
/// panic condition.
pub fn verifyAsNot(
    tuple_ref: TupleRef,
    disallowed_status: AllocationStatus,
) void {
    const tuple_index = tuple_ref.tuple_id;
    const actual_status = storage[tuple_index].allocation;
    if (disallowed_status == actual_status) {
        panic("tuple reference verification failed: should not be '{any}'", .{
            disallowed_status,
        });
    }
    tuple_ref.verifyGeneration();
}

```

```

<<tuple reference declarations>>=
/// The `verifyGeneration` function checks that the generation number of the
/// tuple referenced by `tuple_ref` is the same as the generation number stored
/// in the tuple storage element. Failure to match creates a panic condition.
pub fn verifyGeneration(
    tuple_ref: TupleRef,
) void {
    const tuple_index = tuple_ref.tuple_id;
    const expected_gen = tuple_ref.gen_id;
    const actual_gen = storage[tuple_index].generation;
    if (expected_gen != actual_gen) {
        panic("tuple reference generation mismatch: expected {d}, got {d}", .{
            expected_gen,
            actual_gen,
        });
    }
}

```

```

<<tuple reference declarations>>=
/// The `eql` function return `true` if the tuple referred to by `one_ref` is
/// the same tuple referred to by `other_ref` and `false` otherwise.
pub fn eql(
    one_ref: TupleRef,
    other_ref: TupleRef,
) bool {
    return if (one_ref.tuple_id != other_ref.tuple_id or

```

```

        one_ref.gen_id != other_ref.gen_id)
    false
else
    true;
}

```

```

<<tuple reference declarations>>=
/// The `isReserved` function returns `true` if the tuple storage slot given
/// by `tuple_ref` is in the `reserved` state and `false` otherwise.
pub fn isReserved(
    tuple_ref: TupleRef,
) bool {
    const tuple_index = tuple_ref.tuple_id;
    const actual_status = storage[tuple_index].allocation;
    return if (actual_status == .reserved) true else false;
}

```

Formatting

A custom formatter a TupleRef ensures both the tuple index and generation number are displayed.

```

<<tuple reference declarations>>=
/// A custom `format` function for `TupleRef` values.
pub fn format(
    tuple_ref: TupleRef,
    writer: *std.IO.Writer,
) std.IO.Writer.Error!void {
    try writer.print("{d}:{d}", .{ tuple_ref.tuple_id, tuple_ref.gen_id });
}

```

Tests

```

<<relvar module tests>>=
test "creating as TupleRef" {
    const RvarTest = RelVar(
        struct { // heading
            name: []const u8,
            length: usize,
            weight: f32,
        },
        &.{ // identifiers
            &{.name, .length}, // id 0
            &{.name, .weight}, // id 1
        },
        10, // capacity
        null, // load factor
        &.{ // ...

```

```

        .foo, .{ .name = "foo", .length = 42, .weight = 13.7 },
    },
}, // initial population
);
const rvar_test = RvarTest.init();

// Attempt to create a duplicate tuple containing all the same attribute values.
try testing.expectError(RelVarError.DuplicateTuple, RvarTest.TupleRef.createOrError(.{
    .name = "foo",
    .length = 42,
    .weight = 13.7,
}));
// The original tuple still remains.
try testing.expectEqual(@as(usize, 1), rvar_test.cardinality());

// Duplicate tuple for the 1st identifier.
try testing.expectError(RelVarError.DuplicateTuple, RvarTest.TupleRef.createOrError(.{
    .name = "foo",
    .length = 42,
    .weight = 13.8,
}));
try testing.expectEqual(@as(usize, 1), rvar_test.cardinality());

// Duplicate tuple for the 2nd identifier.
try testing.expectError(RelVarError.DuplicateTuple, RvarTest.TupleRef.createOrError(.{
    .name = "foo",
    .length = 43,
    .weight = 13.7,
}));
try testing.expectEqual(@as(usize, 1), rvar_test.cardinality());

// Add more tuples to the relation variable.
const bar_ref =
    try RvarTest.TupleRef.createOrError(.{ .name = "bar", .length = 43, .weight = 13.8
});
_ = try RvarTest.TupleRef.createOrError(.{ .name = "baz", .length = 44, .weight = 13.9
});
_ = try RvarTest.TupleRef.createOrError(.{ .name = "fuzz", .length = 45, .weight = 14.0
});
try testing.expectEqual(@as(usize, 4), rvar_test.cardinality());

bar_ref.destroy();
try testing.expectEqual(@as(usize, 3), rvar_test.cardinality());
}

```

Sets of Tuple References

Just as Relation Values are sets of Relation Tuples, it is useful to have sets of tuple references. Most often we deal with tuple reference sets since this allows us to handle the empty set case as well as multiple tuple references in a convenient manner. The design uses the standard library `StaticBitSet` to hold tuple reference sets.

The *universe* of a Relation Variable is the tuple reference set of all possible tuples. The bit offset corresponds to the index into the tuple storage for the relation variable. If a particular bit is set, then the corresponding storage slot is allocated and in use. Conversely, an unset bit in the bit set indicates the storage slot is either `.free` or `.reserved`.

The universe of a Relation Variable is used;

1. To find unallocated slots during creation operations.
2. To compute the *complement* of a set of particular bit set.

```
<<variable declarations>>=
/// A type used to indicate a set of tuple references in a relation variable.
pub const TupleRefSet = struct {
    tset: std.StaticBitSet(capacity),

    pub const TupleBitSetType = @FieldType(TupleRefSet, "tset");

    <<tuple ref set declarations>>
};
```

Initialization

There are several convenient ways to initialize the value of a tuple reference set.

```
<<tuple ref set declarations>>=
pub fn initEmpty() TupleRefSet {
    return .{ .tset = .initEmpty() };
}
```

```
<<tuple ref set declarations>>=
pub fn initFromBitSet(
    bit_set: TupleBitSetType,
) TupleRefSet {
    return .{ .tset = bit_set };
}
```

```
<<tuple ref set declarations>>=
/// The `initFromTupleRef` function returns an `TupleRefSet` of cardinality one
/// that references the same tuple as given by `tuple_ref`.
/// Any attempt to access an unallocated tuple or any mismatch between
/// the reference and the tuple generation causes a panic condition.
pub fn initFromTupleRef(
    tuple_ref: TupleRef,
) TupleRefSet {
    tuple_ref.verifyAsNot(.free);
    var tuple_set: TupleRefSet = .initEmpty();
    tuple_set.set(tuple_ref.tuple_id);
    return tuple_set;
```

```
}
```

In the same manner that a relation tuple may be extracted from a relation value of cardinality one, we can obtain tuple reference from a tuple reference set of cardinality one.

```
<<tuple ref set declarations>>=
/// The `tupleRefFromTupleRefSet` function returns a tuple reference from the
/// reference set given by `tuple_set`. The cardinality of `tuple_set` must
/// be one.
pub fn tupleRefFromTupleRefSet(
    tuple_set: TupleRefSet,
) TupleRef {
    const card = tuple_set.count();
    if (card != 1)
        panic("require a single cardinality set, got {d}", .{card});

    const element_index = tuple_set.findFirstSet().?;
    return .initFromIndex(element_index);
}
```

Tuple Reference Set Operations

Many of the methods of `TupleRefSet` are simply methods of the `tset` field “raised” one level to make them more conveniently accessible. The following are the raised bit set functions. Note the list is not exhaustive. The functions are given with no further explanation.

```
<<tuple ref set declarations>>=
pub fn set(
    tuple_set: *TupleRefSet,
    index: usize,
) void {
    tuple_set.tset.set(index);
}
```

```
<<tuple ref set declarations>>=
pub fn unset(
    tuple_set: *TupleRefSet,
    index: usize,
) void {
    tuple_set.tset.unset(index);
}
```

```
<<tuple ref set declarations>>=
pub fn isSet(
    tuple_set: TupleRefSet,
    index: usize,
) bool {
    return tuple_set.tset.isSet(index);
}
```

```
<<tuple ref set declarations>>=
pub fn eql(
    tuple_set: TupleRefSet,
    other: TupleRefSet,
) bool {
    return tuple_set.tset.eql(other.tset);
}
```

```
<<tuple ref set declarations>>=
pub fn setUnion(
    one: *TupleRefSet,
    other: TupleRefSet,
) void {
    return one.tset.setUnion(other.tset);
}
```

```
<<tuple ref set declarations>>=
pub fn complement(
    tuple_set: TupleRefSet,
) TupleRefSet {
    return .{
        .tset = tuple_set.tset.complement(),
    };
}
```

```
<<tuple ref set declarations>>=
pub fn setIntersection(
    one: *TupleRefSet,
    other: TupleRefSet,
) void {
    one.tset.setIntersection(other.tset);
}
```

```
<<tuple ref set declarations>>=
pub fn intersectWith(
    one: TupleRefSet,
    other: TupleRefSet,
) TupleRefSet {
    return .{
        .tset = one.tset.intersectWith(other.tset),
    };
}
```

```
<<tuple ref set declarations>>=
pub fn count(
    tuple_set: TupleRefSet,
) usize {
    return tuple_set.tset.count();
```

```
}
```

```
<<tuple ref set declarations>>=
pub fn setRangeValue(
    tuple_set: *TupleRefSet,
    range: std.bit_set.Range,
    value: bool,
) void {
    return tuple_set.tset.setRangeValue(range, value);
}
```

```
<<tuple ref set declarations>>=
pub fn findFirstSet(
    tuple_set: TupleRefSet,
) ?usize {
    return tuple_set.tset.findFirstSet();
}
```

```
<<tuple ref set declarations>>=
pub fn supersetOf(
    one: TupleRefSet,
    other: TupleRefSet,
) bool {
    return one.tset.supersetOf(other.tset);
}
```

```
<<tuple ref set declarations>>=
pub fn subsetOf(
    one: TupleRefSet,
    other: TupleRefSet,
) bool {
    return one.tset.subsetOf(other.tset);
}
```

Iteration

An **iterator** function is defined for a tuple reference set. The iterator returns successive **TupleRef** values for each member of the **TupleRefSet**. This reinforces the notion that a **TupleRefSet** is a set of **TupleRef** values and is analogous to iterating over a Relation Value to obtain the individual Relation Tuples.

```
<<tuple ref set declarations>>=
/// The `iterator` function returns an iterator type over the `tuple_set`.
/// The iterator type has a single `next` method which returns a
/// `TupleRef` value to access a tuple in a relation variable.
pub fn iterator(
    tuple_set: *const TupleRefSet,
    comptime options: std.bit_set.IteratorOptions,
```

```

) Iterator(options) {
    return .{
        .bit_set_iter = tuple_set.tset.iterator(options),
    };
}

```

```

<<tuple ref set declarations>>=
pub fn Iterator(
    comptime options: std.bit_set.IteratorOptions,
) type {
    return struct {
        const IterSelf = @This();
        bit_set_iter: TupleBitSetType.Iterator(options),

        pub fn next(
            self: *IterSelf,
        ) ?TupleRef {
            return if (self.bit_set_iter.next()) |tuple_index|
                TupleRef.initFromIndex(tuple_index)
            else
                null;
        }
    };
}

```

Destruction

```

<<tuple ref set declarations>>=
/// The `destroy` function deletes all the tuples given by the `tuples_set`
/// argument from the given relation variable.
pub fn destroy(
    tuple_set: *const TupleRefSet,
) void {
    var set_iter = tuple_set.iterator(.{});
    while (set_iter.next() |tuple_ref| {
        tuple_ref.destroy();
    })
}

```

Relation Variable Algebra

In this section, the relation variable algebra is given. Note that the number of algebraic operations on relation variable is considerably less than that for relation values. For relation variables, we are concerned with primarily with creating, deleting, selecting, and updating the tuples in the relation variable. These functions either return or accept a `TupleRefSet` which references the tuples affected. A function to obtain a relation value for a subset of relation variable tuples is provided to yield the full power of the relation value algebra for computations. Other functions are provided to update a relation variable from a relation value. The usual pattern is to select a starting `TupleRefSet`, obtain the corresponding `RelValue`, perform any required computation, and update the `RelVar` from the `RelValue` result.

Relation Variable Introspection

We provide the usual set of introspection functions along with a few other useful ones.

Cardinality

```
<<variable public functions>>=
/// The `cardinality` function returns the number of activated tuples currently
/// stored in the relation value.
pub fn cardinality(
    _: Self,
) usize {
    return universe.count();
}
```

Degree

```
<<variable public functions>>=
/// The `degree` function returns the number of attributes in the heading of
/// the relation value.
pub fn degree(
    _: Self,
) usize {
    return Tuple.tuple_degree;
}
```

Creation

As part of the execution management scheme we discuss in a later [chapter](#), Relation Tuples are sometimes added to a Relation Variable asynchronously. Asynchronous creation arises in the context of state machine event dispatch. For synchronous creation the newly created tuple is immediately available after return from the creation function. For asynchronous creation, the tuple is created when an association event is dispatched. Thus asynchronous creation has “do and forget it” style semantics and implies the logic does not need access to the newly created tuple. This design manages the asynchronous requirement by using the following steps:

1. A slot for a new tuple in the relvar is allocated as, `.reserved`.
2. The tuple value is then inserted into the reserved tuple slot.
3. When an event is received, the tuple is then activated ensuring it passes identity constraints and is then ready for use.

For synchronous creation, all three steps are performed in the same function. For asynchronous creation, the first two steps are performed. This leaves the allocation status of the new tuple as `.reserved`. The `.reserved` status ensures that space is not reused, but that the instance is not yet part of the universe of the tuples for the relation variable. Later, the tuple is activated and is available for use. The notion of “reserved” is used to solve the problem of where to find the memory for the new tuples attribute values by using a slot that will eventually be put into use.

The second step above may seem superfluous. However, when we consider Relation Classes in the following section, we find it helpful when generated class-based identifiers.

In almost all cases, codes executing in a ReSEE environment assumes the environment is “perfect”. This is to say that errors don’t occur when relation variable tuples are created.

```
<<variable public functions>>=
pub fn createOrError(
    _: Self,
    tuple: TupleBase,
) RelVarError!TupleRefSet {
    var created_set: TupleRefSet.TupleBitsetType = .initEmpty();
    const element_index = try Self.reserveAndUpdate(tuple);
    try Self.activate(element_index);
    created_set.set(element_index);
    return TupleRefSet.initFromBitSet(created_set);
}
```

```
<<variable public functions>>=
pub fn create(
    self: Self,
    tuple: TupleBase,
) TupleRefSet {
    return self.createOrError(tuple) catch |err| {
        panic("failed to create tuple reference set: '{s}'", {@errorName(err)});
    };
}
```

```
<<variable public functions>>=
pub fn reserveOrError(
    _: Self,
) RelVarError!TupleRefSet {
    var reserved_set: TupleRefSet.TupleBitsetType = .initEmpty();
    const element_index = try Self.reserve();
    reserved_set.set(element_index);
    return TupleRefSet.initFromBitSet(reserved_set);
}
```

```
<<variable public functions>>=
pub fn reserveForAsync(
    self: Self,
) TupleRefSet {
    return self.reserveOrError() catch |err| {
        panic("failed to reserve tuple for async creation: '{s}'", {@errorName(err)});
    };
}
```

```
<<variable public functions>>=
pub fn reserveAndUpdateOrError(
```

```

_: Self,
tuple: TupleBase,
) RelVarError!TupleRefSet {
    var reserved_set: TupleRefSet.TupleBitsetType = .initEmpty();
    const element_index = try Self.reserveAndUpdate(tuple);
    reserved_set.set(element_index);
    return TupleRefSet.initFromBitSet(reserved_set);
}

```

```

<<variable public functions>>=
pub fn reserveAndUpdateForAsync(
    self: Self,
    tuple: TupleBase,
) TupleRefSet {
    return self.reserveAndUpdateOrError(tuple) catch |err| {
        panic("failed to reserve/update tuple value for async creation: '{s}'",
{@errorName(err)} );
    };
}

```

```

<<variable public functions>>=
pub fn updateReservedOrError(
    _: Self,
    tuple_set: TupleRefSet,
    tuple: TupleBase,
) RelVarError!void {
    const card = tuple_set.count();
    if (card != 1)
        panic("require a set of cardinality 1, got {d}", .{card});

    const element_index = tuple_set.findFirstSet().?;
    try Self.updateReserved(element_index, tuple);
}

```

```

<<variable public functions>>=
pub fn updateReservedForAsync(
    self: Self,
    tuple_set: TupleRefSet,
    tuple: TupleBase,
) void {
    self.updateReservedOrError(tuple_set, tuple) catch |err| {
        panic("failed to update tuple value for async creation: '{s}'",
{@errorName(err)} );
    };
}

```

```

<<variable public functions>>=
pub fn activateOrError(
    _: Self,

```

```

        tuple_set: TupleRefSet,
) RelVarError!void {
    const card = tuple_set.count();
    if (card != 1)
        panic("require a set of cardinality 1, got {d}", .{card});

    const element_index = tuple_set.findFirstSet().?;
    try Self.activate(element_index);
}

```

```

<<variable public functions>>=
pub fn activateForAsync(
    self: Self,
    tuple: TupleBase,
) TupleRefSet {
    return self.activateOrError(tuple) catch |err| {
        panic("failed to activate tuple for async creation: '{s}'", {@errorName(err)});
    };
}

```

It is convenient to be able to create multiple tuples at a time. This is useful at initialization time to create an initial instance population for a relation variable.

```

<<variable public functions>>=
/// The `populate` function adds zero or more tuples to a relation variable.
/// The return value is a `TupleRefSet` value recording all the added tuples.
/// The function is "consistent" in the sense that either all tuples are added
/// successfully or none of them are.
pub fn populate(
    _: Self,
    tuples: []const TupleBase,
) RelVarError!TupleRefSet {
    var tuple_set: TupleRefSet = .initEmpty();
    errdefer tuple_set.destroy();

    for (tuples) |tuple| {
        const element_index = try Self.reserve();
        try Self.updateReserved(element_index, tuple);
        try Self.activate(element_index);
        tuple_set.set(element_index);
    }

    return tuple_set;
}

```

Tests

```

<<relvar module tests>>=
test "creating as TupleRefSet" {
    const RvarTest = RelVar(
        struct { // heading
            name: []const u8,
            length: usize,
            weight: f32,
        },
        &.{ // identifiers
            &{.name, .length}, // id 0
            &{.name, .weight}, // id 1
        },
        10, // capacity
        null, // load factor
        &.{ // initial population
            .{
                .foo, .{ .name = "foo", .length = 42, .weight = 13.7 },
            },
        },
    );
    const rvar_test = RvarTest.init();

    // Attempt to create a duplicate tuple containing all the same attribute values.
    try testing.expectError(RelVarError.DuplicateTuple, rvar_test.createOrError(.{
        .name = "foo",
        .length = 42,
        .weight = 13.7,
    }));
    // The original tuple still remains.
    try testing.expectEqual(@as(usize, 1), rvar_test.cardinality());

    // Duplicate tuple for the 1st identifier.
    try testing.expectError(RelVarError.DuplicateTuple, rvar_test.createOrError(.{
        .name = "foo",
        .length = 42,
        .weight = 13.8,
    }));
    try testing.expectEqual(@as(usize, 1), rvar_test.cardinality());

    // Duplicate tuple for the 2nd identifier.
    try testing.expectError(RelVarError.DuplicateTuple, rvar_test.createOrError(.{
        .name = "foo",
        .length = 43,
        .weight = 13.7,
    }));
    try testing.expectEqual(@as(usize, 1), rvar_test.cardinality());

    // Add more tuples to the relation variable.
    const added_set = try rvar_test.populate(&.{ // new tuples
        .{ .name = "bar", .length = 43, .weight = 13.8 },
        .{ .name = "baz", .length = 44, .weight = 13.9 },
    });
}

```

```

const instset_fuzz = try rvar_test.createOrError(.{ .name = "fuzz", .length = 45,
.weight = 14.0 });
try testing.expectEqual(@as(usize, 4), rvar_test.cardinality());

// Populate with duplicate.
try testing.expectError(RelVarError.DuplicateTuple, rvar_test.populate(&.{.
{ .name = "fur", .length = 46, .weight = 15.0 },
.{ .name = "fuzz", .length = 45, .weight = 14.0 }, // duplicate
}));
// Populate is an all-or-nothing operation. The duplicate caused a
// roll-back to the previous state.
try testing.expectEqual(@as(usize, 4), rvar_test.cardinality());

// Destroy 1 tuple
instset_fuzz.destroy();
try testing.expectEqual(@as(usize, 3), rvar_test.cardinality());

// Destroy 2 tuples
added_set.destroy();
try testing.expectEqual(@as(usize, 1), rvar_test.cardinality());
}

```

Selection

Once you are able to create and destroy tuples in a relation variable, the next important operation is to select subsets of those tuples. Selection almost always precedes further computations on the tuples. In this section, we present selection operations based on tuple attribute values. In a following [section](#), we show how tuple sets can be computed by applying relationships.

Requesting all the tuples is the easiest selection.

```

<<variable public functions>>=
/// The `selectAll` function returns an `TupleRefSet` that references all the
/// tuples currently in the given relation variable.
pub fn selectAll(
    _: Self,
) TupleRefSet {
    return universe;
}

```

Selection by identifier attributes is factored as a special case since it allows us to use the relation variable indices.

```

<<variable public functions>>=
/// The `selectByIdentifier` function returns a `TupleRefSet` whose
/// cardinality is at most one. The set references that tuple in
/// the relation variable whose attributes and values, as given by
/// the `id_attr_values` argument match the identifier given by the

```

```

/// `ident_num`. The `ident_num` is a small integer value that gives
/// the identifier on which the search is based. The `id_attr_values`
/// argument is a anonymous `struct` literal containing the attributes
/// and values that make up the identifiers specified by `ident_num`.
/// If no tuple in the relation variable has attribute values that match
/// the identifying attributes, then the cardinality of the returned
/// `TupleRefSet` value is zero.
pub fn selectByIdentifier(
    : Self,
    comptime ident_num: usize,
    id_attr_values: Self.IdentifierAttributes(ident_num),
) RelVarError!TupleRef {
    // Supply values for the attributes of the identifier from the argument.
    var key: Tuple = undefined;
    inline for (comptime meta.fieldNames(Self.IdentifierAttributes(ident_num))) |
        field_name| {
        @field(key.reltuple, field_name) = @field(id_attr_values, field_name);
    }

    const maybe_found = indices[ident_num].search(&key);
    if (maybe_found) |found| {
        const found_element: *RelVarStorageElement = @fieldParentPtr("tuple", found); ①
        const element_index = found_element - &storage[0];
        assert(storage[element_index].allocation == .allocated);
        return .initFromIndex(element_index);
    } else {
        return RelVarError.NoSuchTuple;
    }
}

```

- ① Since 0.14.0, Zig supports pointer difference within an array to compute the array index of an element. Since the value returned from `search` is a pointer to a tuple, it is necessary to determine the pointer to the enclosing storage element. Then we can compute the corresponding element index by subtracting off the beginning of the array. Note that experimentally, `found - &storage[0].tuple` yields incorrect results. We need to make sure we are computing in `RelVarStorageElement` units to get a proper index into the storage array.

To support selection by an identifier, it is convenient to supply the values of the attributes of an identifier as a simple tuple. A type function computes the required structure.

```

<<variable public functions>>=
/// The `IdentifierAttributes` function returns a `struct` type whose fields
/// are the same as the attributes given by `ident_num`.
pub fn IdentifierAttributes(
    comptime ident_num: usize,
) type {
    var fields: []const StructField = &[_]StructField{};
    inline for (identifiers[ident_num]) |id_attr| {
        fields = fields ++ [_]StructField{.
            .name = @tagName(id_attr),
            .type = AttributeType(id_attr),
            .default_value_ptr = null,
            .is_comptime = false,
        };
    }
}

```

```

        .alignment = @alignOf(AttributeType(id_attr)),
    });
}

return @Type(.{
    .@"struct" = .{
        .layout = .auto,
        .backing_integer = null,
        .fields = fields,
        .decls = &.{},
        .is_tuple = false,
    },
}),
}

```

```

<<variable public functions>>=
/// The `IdentifierValues` function returns a `struct` whose fields match those
/// of the identifier given by `ident_num` and whose values are taken from
/// the corresponding fields of `tuple_value`.
pub fn IdentifierValues(
    comptime ident_num: usize,
    tuple_value: anytype, ①
) Self.IdentifierAttributes(ident_num) {
    if (comptime !reltuple.isRelTupleType(@TypeOf(tuple_value)))
        @compileError("identifier value must come from a Tuple type");

    var id_values: Self.IdentifierAttributes(ident_num) = undefined;
    inline for (comptime meta.fieldNames(@TypeOf(id_values))) |field_name| {
        @field(id_values, field_name) = @field(tuple_value.reltuple, field_name);
    }
    return id_values;
}

```

- ① The use of `anytype` here allows the required tuple values to be acquired from different type functions as long as they have the same attributes. The expectation is that the tuple value is a `Tuple` type with the same attributes as the identifier given by `ident_num`.

Since initial instances are tagged with enumeration literals, it is also possible to select an initial instance by “dead reckoning”, i.e. we have its index into storage hidden as an enumeration literal.

```

<<variable public functions>>=
/// The `selectInitialInstance` function returns the initial instance given
/// by `inst_tag`. All initial instances created in the call to `RelVar`
/// are given an enumeration literal tag which can be used with this
/// function to obtain a `TupleRefSet` containing just instance referred to
/// by `inst_tag`.
pub fn selectInitialInstance(
    _: Self,
    inst_tag: InitialInstanceId,
) TupleRef {
    const element_index = @intFromEnum(inst_tag);
    assert(storage[element_index].allocation == .allocated);
    return .initFromIndex(element_index);
}

```

```
}
```

Selection is also accomplished by examining the tuples with a `selector` function.

```
<<variable public functions>>=
/// The `select` function returns a `TupleRefSet` value that contains
/// a subset of the tuples from the given relation variable. Which
/// tuples are included is determined by the function pointed to by
/// `selector`. The `selector` function is invoked for each tuple in the
/// relation value and is passed the `selector_args` argument. If the
/// return value from an invocation of `selector` is `true`, then the
/// argument tuple to `selector` is included in the returned `TupleRefSet`.
pub fn select(
    _: Self,
    selector: *const fn(tuple: Tuple, args: anytype) bool,
    selector_args: anytype,
) TupleRefSet {
    var tuple_set: TupleRefSet = .initEmpty();

    var set_iter = universe.iterator(.{});
    while (set_iter.next() |tuple_ref| {
        tuple_ref.verifyAs(.allocated);

        const tuple = tuple_ref.dereference();
        if (selector(tuple, selector_args)) {
            tuple_set.set(tuple_ref.tuple_id);
        }
    }

    return tuple_set;
}
```

The `selector` functions for Relation Variables have the same interface as those for Relation Values. A set of functions are provided to supply the syntactic sugar for reusing the relation value selector functions.

```
<<variable public functions>>=
/// The `selectEqual` function returns a `TupleRefSet` value that references those
/// tuples where the value of the `attr_id` attribute is equal to the
/// value of the `attr_value` argument.
pub fn selectEqual(
    self: Self,
    comptime attr_id: AttributeId,
    attr_value: AttributeType(attr_id),
) TupleRefSet {
    return self.select(relvalue.equalitySelector, .{ attr_id, attr_value });
}
```

```

<<variable public functions>>=
/// The `selectAllEqual` function returns a `TupleRefSet` value that indicates those
/// tuples where all of the attribute id / attribute value pairs of the `av_pairs`
/// argument are true. The `av_pairs` argument is assumed to be a list of
/// attribute id / value pairs.
pub fn selectAllEqual(
    self: Self,
    av_pairs: anytype,
) TupleRefSet {
    return self.select(relvalue.conjunctiveEqualitySelector, av_pairs);
}

```

```

<<variable public functions>>=
/// The `selectAnyEqual` function returns a `TupleRefSet` value that indicates
/// those tuples where at least one of the attribute id / attribute value pairs of
/// the `av_pairs` argument is true. The `av_pairs` argument is assumed to be a list
/// of attribute id / value pairs.
pub fn selectAnyEqual(
    self: Self,
    av_pairs: anytype,
) TupleRefSet {
    return self.select(relvalue.disjunctiveEqualitySelector, av_pairs);
}

```

Following the same pattern as with equality selector functions, we provide simpler interfaces which make using the comparative selector function convenient.

```

<<variable public functions>>=
/// The `selectComparison` function returns a `TupleRefSet` value that indicates
/// those tuples where the value of the attribute given by `attr_id` when compared to
/// the `attr_value` by the `op` operation is true.
pub fn selectComparison(
    self: Self,
    comptime attr_id: AttributeId,
    op: math.CompareOperator,
    attr_value: AttributeType(attr_id),
) TupleRefSet {
    return self.select(relvalue.comparativeSelector, .{ attr_id, op, attr_value});
}

```

```

<<variable public functions>>=
/// The `selectAllComparison` function returns a `TupleRefSet` value that indicates

```

```

/// those tuples where the logical *AND* of all the comparisons given by `exprs` are true.
/// The `exprs` argument is taken as a list of triples.
/// Each triple is as described for the `comparativeSelector` function.
pub fn selectAllComparison(
    self: Self,
    exprs: anytype,
) TupleRefSet {
    return self.select(relvalue.conjunctiveComparativeSelector, exprs);
}

```

```

<<variable public functions>>=
/// The `selectAnyComparison` function returns a `TupleRefSet` value that indicates
/// those tuples where the logical *OR* of all the comparisons given by `exprs` are true.
/// The `exprs` argument is taken as a list of triples.
/// Each triple is as described for the `comparativeSelector` function.
pub fn selectAnyComparison(
    self: Self,
    exprs: anytype,
) TupleRefSet {
    return self.select(relvalue.disjunctiveComparativeSelector, exprs);
}

```

Dereferencing Tuple Sets

Computations on tuple subsets of a Relation Variable are done by extracting the subset into a Relation Value and then applying relation value algebra operations from the last section to compute a result.

```

<<tuple ref set declarations>>=
/// The `relValueFromTupleSet` returns a `RelValue` value containing those tuples
/// included in the `tuples` argument.
pub fn relValueFromTupleSet(
    tuples: TupleRefSet,
) RelValueType {
    var builder = RelValueType.Builder.init(tuples.count()); ①

    var set_iter = tuples.iterator(.{});
    while (set_iter.next()) |tuple_ref| {
        tuple_ref.verifyAs(.allocated);
        const inserted = builder.insert(tuple_ref.dereference());
        assert(inserted);
    }

    return builder.finalize();
}

```

① This function is the primary motivation for the Builder interface included in the RelValue type.

Tests

```
<<relvar module tests>>=
test "relValueFromTupleSet tests" {
    relvalue.relExprBegin();
    defer relvalue.relExprEnd();

    const Rvar1 = RelVar(
        struct {
            name: []const u8,
            length: usize,
            weight: f32,
        },
        &.{&{&{.name, .length}, &{.name, .weight}},&},
        10,
        null,
        &.{&{.foo, &{.name = "foo", .length = 42, .weight = 13.7}},&{.bar, &{.name = "bar", .length = 43, .weight = 13.8}},&{.baz, &{.name = "baz", .length = 44, .weight = 13.9}},&{.fuzz, &{.name = "fuzz", .length = 45, .weight = 14.0}},&},
    );
    const rvar_1 = Rvar1.init();

    try testing.expectEqual(@as(usize, 4), rvar_1.cardinality());

    const inst_value = rvar_1.selectAll().relValueFromTupleSet();
    try testing.expectEqual(@as(usize, 4), inst_value.cardinality());
    try testing.expectEqual(@as(usize, 42 + 43 + 44 + 45), inst_value.sum(.length));
    try testing.expectEqual(@as(f32, 13.7 + 13.8 + 13.9 + 14.0), inst_value.sum(.weight));
}
```

Updating Tuples in a Relation Variable

When more extensive computations are required, the `relValueFromTupleSet` function in the previous section provides a technique to obtain the relation value corresponding to a `TupleRefSet`. The corresponding relation value can be used with the relation value algebra from the last section to obtain the required result. Any calculated relation value can then updated back into the relation variable. There are three use cases for the disposition of the relation value.

1. The relation variable can be updated from a relation value. In this case, all tuples in the relation value are required to be present in the relation variable and the update changes the non-identifying attributes.
2. The tuples of the relation variable can be inserted into the relation variable and any duplicate tuples cause an error.
3. The relation variable can be updated from the relation value and all tuples that are present in

the relation variable have their non-identifying attributes updated and those tuples not already in the relation variable are inserted.

N.B. it is *not* possible to change the value of identifying attributes in any of the used cases just described. Either non-identifying attributes are updated or new tuples are inserted. Identifying attributes can only be set when a tuple is created in the relation variable and changes to identifying attributes can only be obtained by deleting the old tuple and re-creating a new tuple.

The following three functions are provided for the described use cases. Note also that if an error is detected in the update functions, it is possible for the the relation variable to be partially updated. There are no guarantees that the updates are "all or nothing."

```
<<variable public functions>>
/// The `updateFromRelValue` function updates all the tuples in the given
/// `RelVar` using values from the `updates` argument. The identifying
/// attributes of each tuple in `updates` are used to locate the tuple
/// in the relation variable which is then overwritten with the tuple
/// from `updates`. Note, it is an error if a tuple from `updates` is
/// not found in the relation variable.
pub fn updateFromRelValue(
    _: Self,
    updates: RelValueType,
) void {
    for (updates.body()) |update_tuple| {
        var maybe_found: ?*Tuple = null;
        for (&indices) |*index| {
            maybe_found = index.search(&update_tuple);
            if (maybe_found == null) {
                panic("unable to find update tuple in relation variable", .{});
            }
        }

        const found = maybe_found.?;
        found.* = update_tuple; ①
    }
}
```

- ① Note there is no need to "reindex" the replacement tuple. All attributes that participate in any identifier have the same value since we were able to locate the tuple in the all of its indices. Rehashing the update tuple would then necessarily yield the same hash value.

```
<<variable public functions>>
/// The `insertFromRelValue` function inserts all the tuples in the
/// `inserts` argument into the given `RelVar` and returns a `TupleRefSet`
/// identifying the inserted tuples. It is an error to attempt
/// to insert a duplicate tuple.
pub fn insertFromRelValue(
    self: Self,
    inserts: RelValueType,
```

```

) TupleRefSet {
    var tuple_set: TupleRefSet = .initEmpty();
    errdefer tuple_set.destroy();

    for (inserts.body()) |insert_tuple| {
        const new_set = self.create(insert_tuple.reltuple);
        tuple_set.setUnion(new_set);
    }

    return tuple_set;
}

```

```

<<variable public functions>>=
/// The `updateOrInsertFromRelValue` function updates all the tuples in the
/// given `RelVar` using values from the updates argument. Any tuple not
/// found in the relation variable is inserted. The function returns a
/// `TupleRefSet` of the inserted tuples (but not the updated tuples).
pub fn updateOrInsertFromRelValue(
    self: Self,
    updates: RelValueType,
) TupleRefSet {
    var tuple_set: TupleRefSet = .initEmpty();

    for (updates.body()) |update_tuple| {
        var maybe_found: ?*Tuple = null;
        var found_count: usize = 0;
        for (&indices) |*index| {
            maybe_found = index.search(&update_tuple);
            if (maybe_found != null) {
                found_count += 1;
            }
        }
    }

    // Make sure any tuple to be updated is found in all
    // of the indices.
    if (found_count > 0 and found_count < indices.len)
        panic("found tuple in only {d} indices: expected {d}", .{
            found_count,
            indices.len,
        });

    if (maybe_found) |found| {
        found.* = update_tuple;
    } else {
        const new_set = self.create(update_tuple.reltuple); ①
        tuple_set.setUnion(new_set);
    }
}

return tuple_set;

```

```
}
```

- ① The tuple was not found in any of the RelVar indices, so if the creation fails, something is terribly wrong.

Tests

```
<<relvar module tests>>=
test "updateFromRelValue tests" {
    relvalue.relExprBegin(); defer relvalue.relExprEnd();

    const Rvar_1 = RelVar(
        struct {
            name: []const u8,
            length: usize,
            weight: f32,
        },
        &.{&{.name, .length},
    },
    10,
    null,
    &.{&{.foo, &{.name = "foo", .length = 42, .weight = 13.7 }},
        &{.bar, &{.name = "bar", .length = 43, .weight = 13.8 }},
        &{.baz, &{.name = "baz", .length = 44, .weight = 13.9 }},
        &{.fuzz, &{.name = "fuzz", .length = 45, .weight = 14.0 }},
    },
);
const rvar_1 = Rvar_1.init();

// Change the .weight attribute by creating a relation value of the
// proper type and then updating the relation variable from the
// values contained in the relation value.
const Rvar1RelValue = Rvar_1.RelValueType;
const update = Rvar1RelValue.create(&.&{
    .{.name = "bar", .length = 43, .weight = 20.8 },
});
rvar_1.updateFromRelValue(update);
const bar_inst = try rvar_1.selectByIdentifier(0, &{&{.name = "bar", .length = 43}});
try testing.expectEqual(@as(f32, 20.8), bar_inst.readAttribute(.weight));

// Add new tuples.
const inserts = Rvar1RelValue.create(&.&{
    .{.name = "zoom", .length = 46, .weight = 30.8 },
});
const zoom_inst = rvar_1.insertFromRelValue(inserts);
```

```

try testing.expectEqual(@as(f32, 30.8), zoom_inst.readAttribute(.weight));

// Update and add new tuples.
const upserts = Rvar1RelValue.create(&.{.
    .{ .name = "baz", .length = 44, .weight = 40.8 },
    .{ .name = "zaz", .length = 50, .weight = 10.8 },
});
_= rvar_1.updateOrInsertFromRelValue(upserts);
const baz_inst = try rvar_1.selectByIdentifier(0, .{
    .name = "baz",
    .length = 44,
});
try testing.expectEqual(@as(f32, 40.8), baz_inst.readAttribute(.weight));
const zaz_inst = try rvar_1.selectByIdentifier(0, .{
    .name = "zaz",
    .length = 50,
});
try testing.expectEqual(@as(f32, 10.8), zaz_inst.readAttribute(.weight));
}

```

```

<<tuple ref set declarations>>=
/// The `updateAttribute` function sets the attribute given by `attr_id` to
/// the value given by `value` for all tuples referenced in `tuples`.
/// The update changes the attribute values in situ.
/// It is an error to attempt to update an identifying attribute.
pub fn updateAttribute(
    tuples: TupleRefSet,
    comptime attr_id: AttributeId,
    value: AttributeType(attr_id),
) void {
    comptime validateAttrAsNonIdentifying(attr_id);

    var set_iter = tuples.iterator(.{});
    while (set_iter.next() |tuple_ref| {
        tuple_ref.verifyAs(.allocated);
        @field(storage[tuple_ref.tuple_id].tuple.reltuple, @tagName(attr_id)) = value;
    }
}

```

```

<<variable private functions>>=
fn validateAttrAsNonIdentifying(
    comptime attr_id: AttributeId,
) void {
    for (relvar_identifiers) |identifier| {
        for (identifier) |identifier_attr| {
            if (identifier_attr == attr_id) {
                @compileError(
                    "updates to identifying attributes are not allowed: '" ++
                    @tagName(attr_id) ++

```

Attribute Access

There are several common use cases where it is convenient to have access to read and update attributes of a relation variable directly.

```
<<tuple ref set declarations>>=
/// The `readAttribute` attribute function returns the value of the
/// attribute given by `attr_id` from the tuple set given by `tuple`.
/// The cardinality of the tuple set given by `tuple` must be one.
pub fn readAttribute(
    tuple: TupleRefSet,
    comptime attr_id: AttributeId,
) AttributeType(attr_id) {
    assert(tuple.count() == 1);

    const element_index = tuple.findFirstSet()?.?;
    assert(storage[element_index].allocation == .allocated);
    return storage[element_index].tuple.extract(attr_id);
}
```

Tests

```
<<relvar module tests>>=
test "attribute access tests" {
    const Rvar_1 = RelVar(
        struct {
            name: []const u8,
            length: usize,
            weight: f32,
        },
        &.{&{.name, .length},
        },
        10,
        null,
        &.{&{.foo, &{.name = "foo", .length = 42, .weight = 13.7 }},
        &{.bar, &{.name = "bar", .length = 43, .weight = 13.8 }},
        &{.baz, &{.name = "baz", .length = 44, .weight = 13.9 }},
        &{.fuzz, &{.name = "fuzz", .length = 45, .weight = 14.0 }},
    },
}
```

```

);
const rvar_1 = Rvar_1.init();

const bar_inst = try rvar_1.selectByIdentifier(0, .{
    .name = "bar",
    .length = 43,
});
try testing.expectEqual(@as(f32, 13.8), bar_inst.readAttribute(.weight));

const bar_baz_insts = rvar_1.selectAnyEqual(&.{.
    .{ .name, "bar" },
    .{ .name, "baz" },
});
bar_baz_insts.updateAttribute(.weight, 15.0);
const baz_inst = try rvar_1.selectByIdentifier(0, .{
    .name = "baz",
    .length = 44,
});
try testing.expectEqual(@as(f32, 15.0), baz_inst.readAttribute(.weight));
try testing.expectEqual(@as(f32, 15.0), bar_inst.readAttribute(.weight));
}

```

Code layout

```

<<relvar.zig>>=
///! The `relvar.zig` module implements the concept of a *Relation
///! Variable*. A Relation Variable is a set of Relation Tuples, all of
///! which have the same heading. The heading is a set of attribute names
///! / attribute data types as for a Relation Tuple. Relation variables
///! are mutable and can be created, destroyed, read, and updated.
///! Relation variable have one or more identifiers, each identifier
///! consisting of one or more attributes. A relation variable is a set
///! and the identifiers are used to ensure that no duplicate tuples are
///! present in a relation variable. The attributes of a relation heading
///! have no inherent _order_. Relation Variables have a lifetime that
///! is the same as the lifetime of the program itself and are the primary
///! mechanism used to store application data.
<<edit warning>>
<<copyright info>>

const std = @import("std");
const builtin = @import("builtin");
const testing = if (builtin.os.tag == .freestanding)
    @import("resee_testing")
else
    std.testing;
const mem = std.mem;
const math = std.math;
const fmt = std.fmt;
const meta = std.meta;
const enums = std.enums;

```

```

const assert = std.debug.assert;
const panic = std.debug.panic;

const StructField = std.builtin.Type.StructField;
const EnumField = std.builtin.Type.EnumField;

pub const version = "1.0.0-a1";

<<relvar module declarations>>

<<relation variable>>

<<relation index>>

<<relvar module public functions>>

<<relvar module private functions>>

<<relvar module tests>>

```

Binary Relations Between Relation Variables

In this section, we show how relation variables can have relationships amongst themselves. These relationships are an encoding of subject matter associations in the same sense that relation variable headings are an encoding of a logical predicate about the characteristics of the entities of the same subject matter. We attribute meaning to relationships using simple verb phrases. Combining relation variables and their relationships gives a precise vocabulary for the subject matter.

There are two types of relationships, associations and generalizations. In the next section, we discuss associations relationships from several viewpoints to arrive at a design for handling them. Then, we consider generalization relationships and show how the design concepts extend to them.

Association Relationships

We start with a small example that is visualized using a UML diagram.

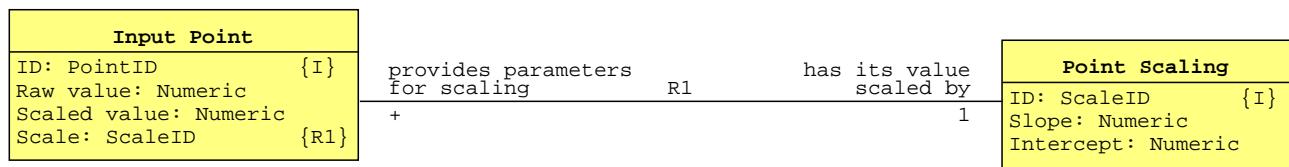


Figure 32. Example class diagram

Note, that we are abusing UML graphical symbols^[12]. Since the graphical symbols used by UML for denoting multiplicity and conditionality are clunky, we use a notation inspired by common regular expression syntax. The notation also facilitates writing relationships in-line in the text. The following table shows the correspondence.

Regular Expression Notation	UML Notation	Meaning
1	1	Exactly one
?	0..1	At most one
+	1..*	At least one
*	0..* or *	Any cardinality

Table 9. Notation Legend for Association Conditionality and Multiplicity

The diagram is a fragment of a subject matter domain which manages obtaining input from the system environment by reading transducer values into an **Input Point** and converting those raw device values into engineering units by linear scaling. **Input Point** and **Point Scaling** are *classes* by the definition presented by the ReSEE execution model. Each class has one identifier (as shown by the {I} annotation). An identifier may consist of multiple attributes, but in this example only one attribute is needed. **R1** is an *association* between the classes. The meaning of the association is contained in the verb phrases on each side of the association line. The formalization of the association is indicated by the {R1} annotation for the **Input Point.Scale** attribute. The association conditionality and multiplicity state that each instance of **Input Point** has its value scaled by exactly one instance of **Point Scaling** and each instance of **Point Scaling** provides parameters for scaling at least one **Input Point** instance. We know which **Point Scaling** to use for a particular instance because the value of the **Input Point.Scale** attribute matches the value of the **Point Scaling.ID** attribute for exactly one instance of **Point Scaling**. The converse is also true, we know which instances of **Input Point** are scaled by any given instance of **Point Scaling** because the value of **Point Scaling.ID** matches the value of **Input Point.Scale** for at least one instance of **Input Point**.

Given an instance of **Input Point**, the execution model provides operations that allow you to find the associated instance of **Point Scaling** by *traversing* the **R1** association from **Input Point** to **Point Scaling**. It is fitting to emphasize this is an analytical modeling formalism and *not* a direct specification for a program design.

The Relational Model of Data Viewpoint

The ideas of the relational model of data are often conflated with Relational Database Management Systems (RDMS), particularly those that use the SQL query language. This is not surprising since the first proposed use of the relational model was for RDMS and, for some, exposure to the relational model is limited to brief references as the underlying theory of RDMS. But the relational model stands apart from database management.

The relational model has — despite its origins — nothing intrinsically to do with databases. Rather it is an elegant approach to structuring data, a means for manipulating such data, and a mechanism for maintaining integrity and consistency of state. These features are applicable to state and data in any context.

— Mosely and Marks, Out of the Tar Pit

In terms of the relational model, associations operate by having referential attribute values of a tuple in a relation variable match the identifying attribute values of a tuple in the associated (but possibly the same) relation variable^[13]. We have already seen the operations that can be performed on a relation value. To perform the equivalent of traversing an association, we could use the relational algebraic operation of *semijoin* on the relation values held in the corresponding relation variables. For our example, this operation would *join* the relation values held in the **Input Point** and **Point Scaling** relation variables across tuples where **Input Point.Scale** and **Point Scaling.ID**

are equal and then *project*, from the joined result, the attributes of **Point Scaling**. The net effect is to find the instances of **Point Scaling** that are related to a set of **Input Point** across the **R1** association. The equality of relation variable referential attributes and relation variable identifying attributes form a binary relation.

It is convenient to represent the contents of relation variables in a tabular manner^[14]. For our example, assume the following small populations of **Input Point** and **Point Scaling**.

ID	Raw value	Scaled value	Scale
0	27	58	0
1	42	88	0
2	10	37	1

Table 10. Input Point Relation

ID	Slope	Intercept
0	2	4
1	4	-3

Table 11. Point Scaling Relation

This population shows that **Input Points** 0 and 1 are scaled linearly by a line of slope 2 and intercept 4. **Input Point** 2 is scaled by a line of slope 4 and intercept -3. Again, we emphasize this is model formalism and not a specification of a design or implementation.

The Binary Relations View

The relational model of data, first described by [Edgar Codd](#), is an application of the mathematical theory of [binary relations](#) to structuring computer data. The vocabulary of binary relations uses terms such as *domain*, *codomain*, *Cartesian product* and many others. In this context, we use the term *association relation* to mean the mapping from a set called the *domain* to a set called the *codomain*. Often, that mapping is a mathematical function and we reserve the unadorned term, *function*, for that specific usage.

A binary relation is a set of ordered pairs that specifies a mapping from the domain to the codomain. Since we are dealing with computing machinery of finite size, we assume the domain and codomain are finite, the binary relation is then finite, and the ordered pairs can be completely enumerated. The following table is the enumeration of the binary relation for our example population shown in tabular form.

Input Point	Point Scaling
0	0
1	0
2	1

Table 12. R1 as a Binary Relation

The *image* of the domain under the relation is the set of related codomain elements. The *preimage* of the codomain under the relation is the set of related domain elements. These are conventional

terms used to describe the actions of functions.

In the binary relation view, we equate the relation variable containing the referential attributes as the domain and the relation variable containing the identifying attributes as the codomain. For those binary relations which require an associative class in the relational model view, the choice of domain and codomain classes just needs to be made and used consistently. Again, we tend to choose the class acting as a subject as the domain and the class acting as the object of the association verb phrase as the codomain. This is convention, not a strict requirement.

Once the domain and codomain sets are identified and the ordered pairs are determined, we can dispense with all the attributes since they have no effect on the binary relation between the sets^[15].

Adjacency Matrix Viewpoint

A binary relation is a directed graph where the graph nodes are class instances and the graph arcs represent the ordered pairs in the relation.

There are several ways to represent the directed graph of a binary relation. The representation shown in the previous figure is essentially an *adjacency list*. If we replace the referential attributes with direct memory pointers to a class instances, then we see that the common way of implementing a binary relation using programming language pointers is essentially an adjacency list view of the relation.

Another convenient representation of a binary relation is as an *adjacency matrix*. This allows us to use matrix algebra to compute the relation over the participating sets. By indexing the rows of the matrix with the domain elements and the columns with the codomain elements, our example, **R1**, can be represented as a matrix of boolean values where any cell that contains *true* indicates that the domain element is related to the codomain element.

$$R1 = \begin{bmatrix} true & false \\ true & false \\ false & true \end{bmatrix}$$

Note, that each row of the matrix has *exactly one* true value and each column of the matrix has *at least one* value of *true* in keeping with the specification of the multiplicity and conditionality for **R1**.

We can compute the **Point Scaling** associated with **Input Point** 0 by using a row vector to represent a subset^[16] of **Input Point** and multiply it (on the right) by the adjacency matrix of the association relation. The matrix multiplication uses the standard dot product definition where boolean multiplication is *logical and* and boolean addition is *logical or*. The multiplication yields a subset of **Point Scaling** instances.

$$\begin{bmatrix} true_0 & false_1 & false_2 \end{bmatrix} \times \begin{bmatrix} true & false \\ true & false \\ false & true \end{bmatrix} = \begin{bmatrix} true_0 & false_1 \end{bmatrix}$$

If we use the normal encoding of *true* and *false* as the integers 1 and 0, respectively, and use integer arithmetic, we can represent the mapping by **R1** as:

$$\begin{bmatrix} 1_0 & 0_1 & 0_2 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1_0 & 0_1 \end{bmatrix}$$

In words: if you have a subset of **Input Point** consisting of only **Input Point** instance, 0, and apply it to **R1** then the result is a subset of **Point Scaling** consisting of only the **Point Scaling** instance,

0. When considering how to scale an instance of **Input Point**, we can compute, by matrix multiplication, the associated instance of **Point Scaling** (and there is exactly one) to obtain the **Slope** and **Intercept** numbers needed for the scaling operation.

It is insightful to consider the converse traversal of **R1**, i.e. from **Point Scaling** to **Input Point**. The converse relation, i.e. the relation which associates codomain elements to domain elements, is computed using the transpose of the relation matrix. Traversing from **Point Scaling** instance, 0, to **Input Point** is given by the matrix multiplication of the transpose.

$$\begin{bmatrix} 1_0 & 0_1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1_0 & 1_1 & 0_2 \end{bmatrix}$$

In words: **Point Scaling** instance, 0, is used to scale the values of both **Input Point**, 0 and 1.

Referential Consistency Viewpoint

An integral part of the manner which the data model uses the relational model is to specify the multiplicity and conditionality of each side of the association. Coupled with verb phrases on each side of the association that state the semantic intent, the multiplicity and conditionality provide a precise way to relate the association to the requirements of the problem domain.

In this example, **Input Point** +--1 **Point Scaling**, the association is a surjective function from the **Input Point** set to the **Point Scaling** set. This means that all instances of **Input Point** must be related to exactly one instance of **Point Scaling** (that makes it a function) and that each instance of **Point Scaling** must be associated with at least one element of **Input Point** (that makes it surjective).

When the system runs, elements of **Input Point** and / or **Point Scaling** may be created, deleted, or re-associated. At some point in time, unspecified here, the system comes to stasis and we want to ensure that any processing has not compromised the referential consistency of the association as given by its multiplicity and conditionality. We consider a transaction on the data model as a period where the association may be temporarily inconsistent, but must be brought into a consistent state by the end of the transaction.

It is important to remember that ongoing processing that modifies the association between the elements of the participating sets is actually creating a new binary relation between the sets^[17]. In the example, if a new instance of **Input Point** is created, then it is necessary to relate that instance to an instance of **Point Scaling**. That action updates the bits in the adjacency matrix and therefore the association relation is now different. We require that the properties of the binary relation as stated by the multiplicity and conditionality are *invariant* to any changes in the association relation adjacency matrix.

We are *not* defining the scope of the presumed transaction, nor do we specify any actions that happen if an inconsistency is discovered. Those issues are not handled by a simple concrete data structure, but are important enough to an overall execution mechanism that we give functions later to determine if an association relation is consistent with its stated properties.

This is a slightly different view of binary relations than is usually taken by mathematical texts. There, the sets are specified (often infinite) and it is usually the properties of the binary relation that are of interest. Here, the properties of the binary relation are fixed *a priori*, and those properties are used to *constrain* the execution of the model.

The determination of referential consistency is a matter of counting the number of true values found in the rows and columns of the adjacency matrix. For the **R1** adjacency matrix shown above, the constraint of each **Input Point** instance being associated to exactly one **Point Scaling** is equivalent to insisting that each row of the **R1** adjacency matrix have exactly one *true* value in the

row. Conversely, the constraint that each **Point Scaling** instance be used to scale at least one **Input Point** is equivalent to insisting that the count of *true* cells in each column of the **R1** adjacency matrix must be greater than or equal to one.

Should the program execution lead to a referentially inconsistent state of the data model, that result is considered an analysis error. However, we also consider it a requirement on the model execution mechanisms to detect the inconsistency during execution. Some model execution mechanisms may choose not to enforce the consistency constraints. This is analogous to whether array indexing constraints are enforced. For some class of applications, the cost of the computation may be deemed too great and the burden of ensuring the referential consistency is placed solely upon the analysts. For systems that have high safety requirements, the constraints must be enforced. The model execution mechanism may choose to provide referential consistency checks on an optional basis with the usual understanding that thorough testing lowers, but does not eliminate, the probability of run-time consistency violations and the unpredictable behavior that may result.

Generalization Relationships

There is another form of a relationship called, generalization. Partitioning the domain into a family of sets is the fundamental idea behind a generalization. Each domain partition is injective into the codomain and the union of the images of the domain partitions under the generalization is disjoint. The disjoint union is a complementary idea to the Cartesian product (sometimes called its *dual*). To mathematicians, this is an important insight. To software engineers, it's a well-founded concept for encoding the distinctions between sets and still maintain strict referential constraints. Associations are fundamentally derived from subsets of the extended Cartesian product of the participating classes. Generalization are fundamentally derived from partitioning the domain set.

A generalization relationship partitions its domain into a family of sets, but it has a single codomain. We call the domain partitions, *subclasses*, and the codomain is called the *superclass*.

The rules for generalization relationships are:

1. Each domain element in each partition of the domain must be associated to exactly one codomain element. In terminology words, the domain is injective under the generalization. In modeling words, each subclass tuple must be related to exactly one superclass tuple.
2. The union of the images of the domain partitions must be disjoint. In modeling words, every subclass tuple belongs to only one subclass.
3. The union of the images of the domain partitions must equal the codomain. In modeling words, every subclass tuple must participate in the generalization.

Partitioning is not inheritance

For those readers familiar with standard UML notation, the generalization graphic we show below is usually interpreted as some form of inheritance, particularly structural data inheritance along with some form of polymorphism. *N.B.* our use of the term generalization and the manner in which it is defined here does *not* imply any type of inheritance or polymorphism. There is polymorphism in this execution model, but it arises in a different context than this discussion. This approach is abstract data type oriented. Conventional object orientation takes a different view of the problem^[18]. An [essay](#) by William R. Cook clarifies the differences. We admit a certain discordance associated with the meaning we apply to the UML graphical symbol, but introducing new graphical symbols offers additional precision at the cost of additional cognitive burden. Alas, there is no solution that is without drawbacks.

Forms of a Generalization

There are three forms of generalization that arise from the roles played by the domain and codomain partitions. The figures below are redrawn from [mtoc], p. 163.

Repeated generalization

is the case when a domain partition serves as the codomain of a different generalization.

In the following figure, the domain of **R1** is partitioned into three subsets and the **Truck** partition serves as the codomain for **R2**. So, **Truck** undergoes repeated generalization. A single level of generalization with no additional repeated generalization is the most common form found in analysis models. A large number of levels is usually indicative of analysis problems.

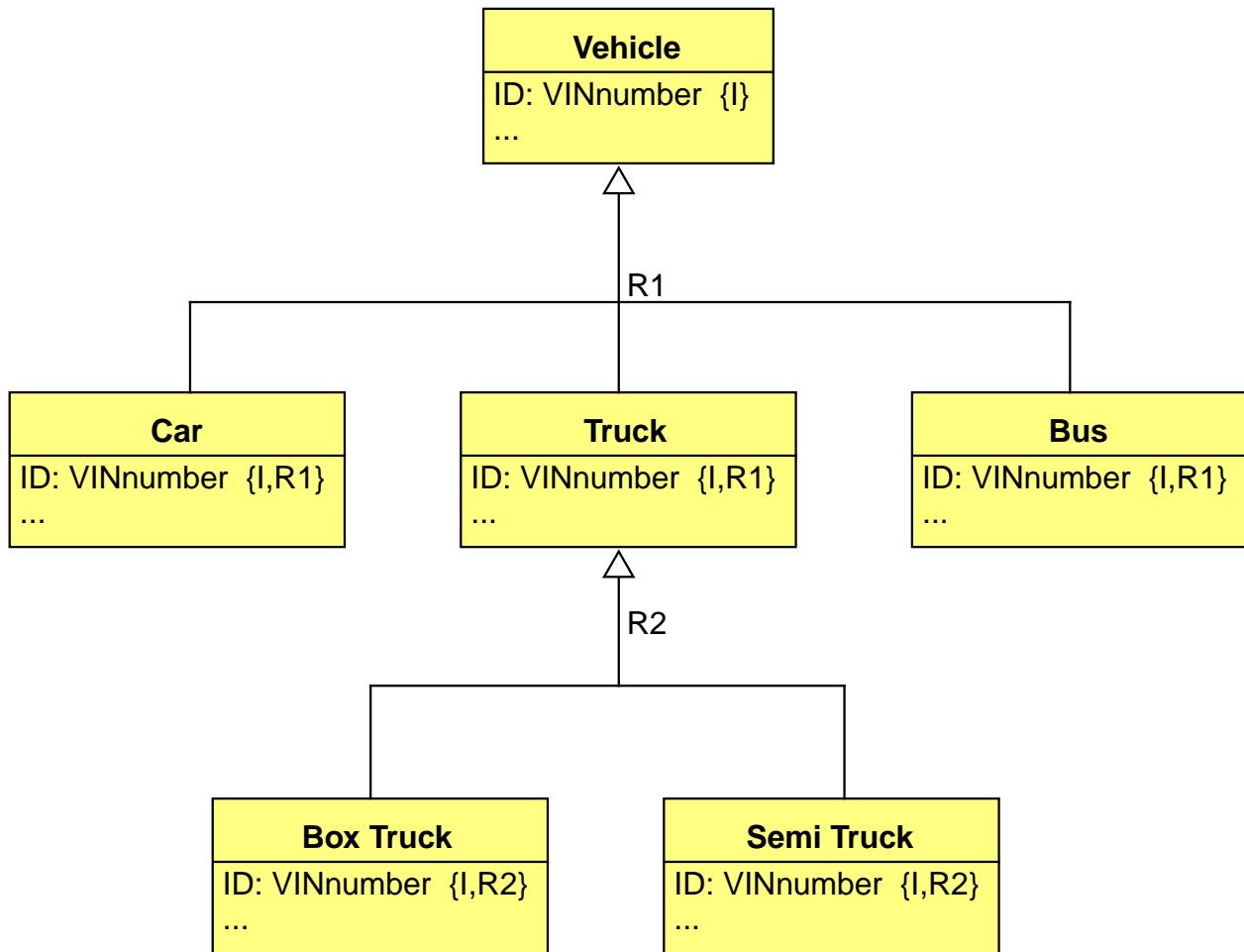


Figure 33. Repeated Generalization Example

Multiple generalization

is this case, a domain partition serves as a domain partition in more than one generalization.

In the following figure, the **VTOL Aircraft** is a domain partition for the two generalizations specified by **R1** and **R2**. For every instance of **VTOL Aircraft** there are instances of both **Rotary Wing Aircraft** and **Fixed Wing Aircraft**.

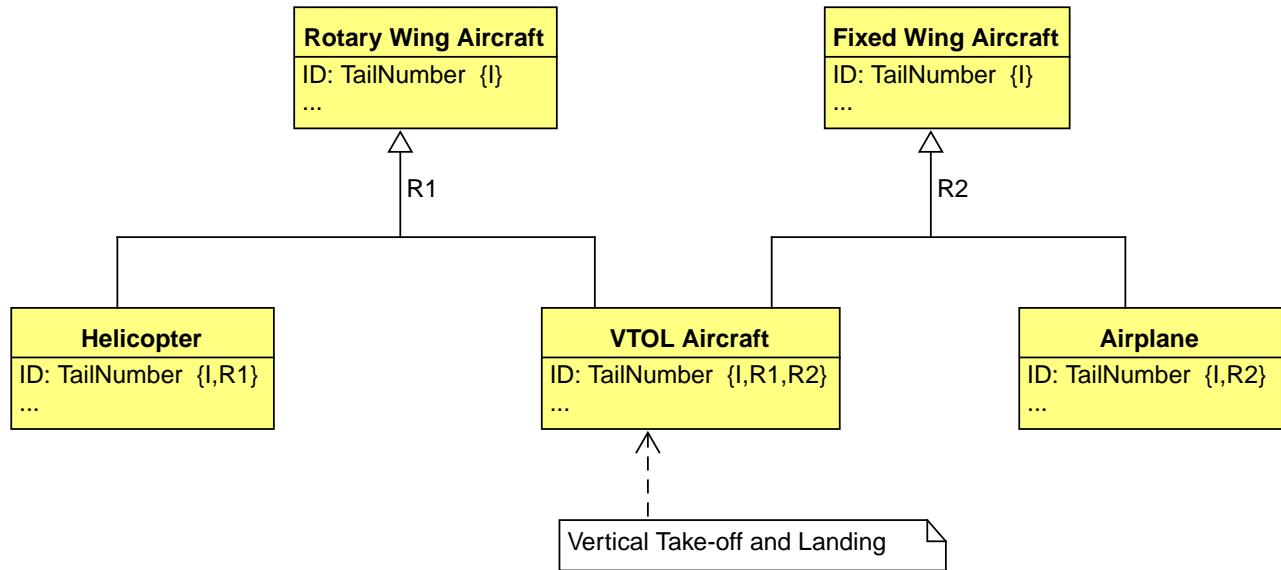


Figure 34. Multiple Generalization Example

Compound generalization

is the case when a codomain serves as the codomain for more than one generalization.

The following figure shows that **Mars Probe** is partitioned in two orthogonal ways, both by its mobility and its power source. For each instance of **Mars Probe** there are related instances in both **R1** and **R2**. In real-world models of concrete entities, the strict partitioning of a compound generalization rarely holds. In this example, a thorough analyst would want to make sure that there are indeed nuclear powered rovers. If not, then the situation should be treated as a repeated generalization of stationary probes.

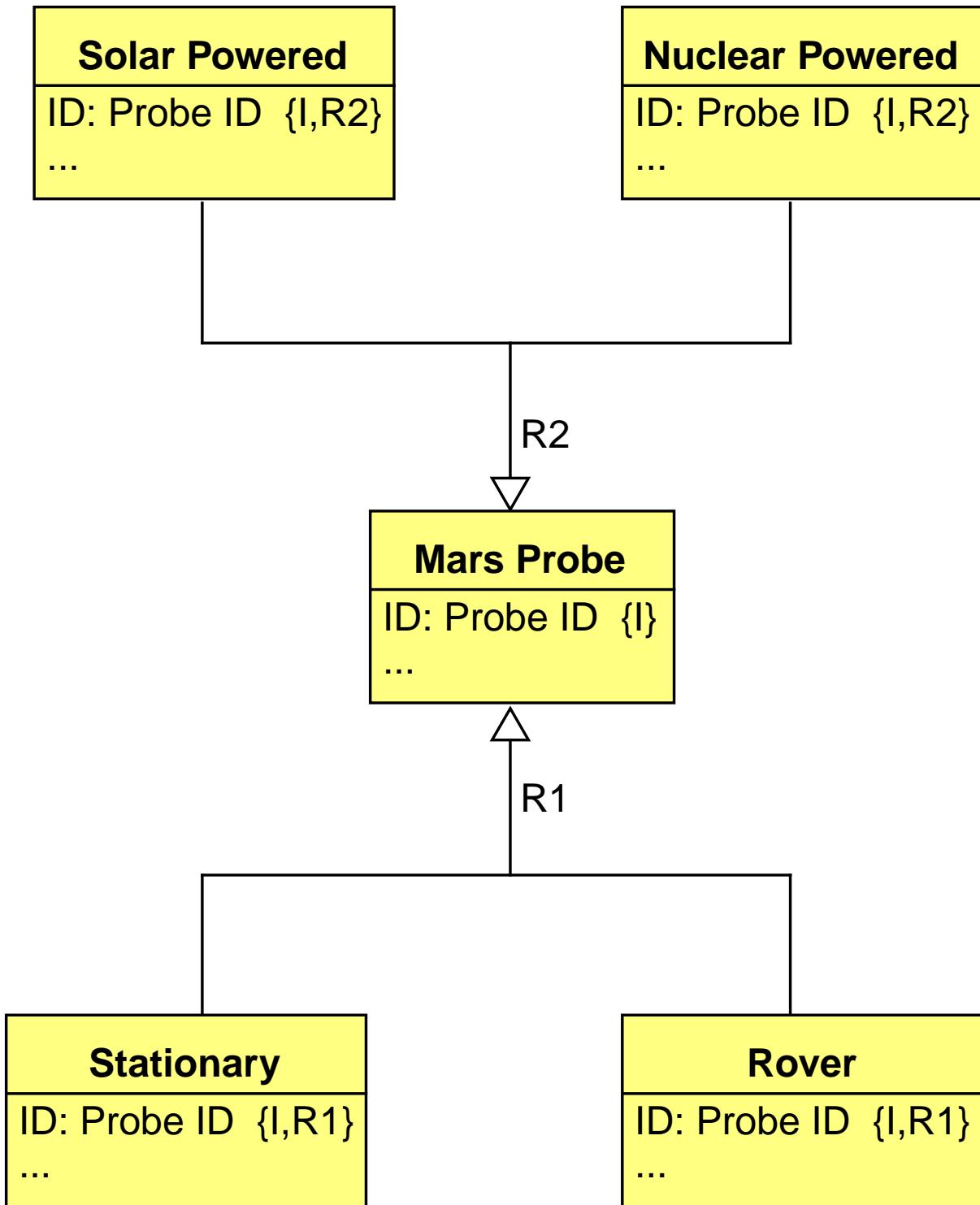


Figure 35. Compound Generalization Example

Adjacency Matrices for Associations

From the discussion in the last section, we intend to implement relationships between classes using an adjacency matrix. There are several designs that could be used to implement the matrix. The adjacency matrix is a matrix of boolean values. After all, it only takes a single bit of information to record whether two set elements are associated. The first impulse might be to have a two-dimensional matrix of booleans, *i.e.*

```
bool[domain_capacity][codomain_capacity]
```

Such a design would implement the adjacency matrix as we have described it. Consider trying to find the associated set of a single domain element as shown above. The multiplication consists of a boolean row matrix that is 1 by *domain_cardinality* as the left term. Since we are considering a single element, only one of the values in the row matrix is set to true. The right term of the multiplication is the adjacency matrix for the association. The traditional algorithm for matrix multiplication then computes the dot products to yield a 1 by *codomain_cardinality* matrix. It would be necessary to index into the adjacency matrix and iterate across each boolean in the row (or column, if computing the inverse set) determining which entries are true. Using the usual integer representation, such a naive algorithm would require a lot of iteration and integer multiplication appears to be a poor choice for our purposes. However, this is speculative and might be a reasonable avenue of experimentation.

Since computers are quite good at bit-wise operations on memory words and since we are dealing with boolean values which can be represented as a single bit, storing the rows of the adjacency matrix as a bit-set seems a reasonable choice for this problem. We also use a bit-set to store the transpose of the relation adjacency matrix. The considerations for that choice are discussed [later](#). The [Zig Standard Library](#) provides several implementation variations of bit-sets. For this design, we are interested in a `StaticBitSet`.

There are two other important requirements for the design to discuss. Although we have allocated a maximum number of elements in the domain and codomain sets, typically not all the storage locations implied by the size of the domain or codomain set may be in use simultaneously. Dynamic allocation of domain and codomain elements happens and not all elements of the domain or codomain may be in use. This implies we need a mechanism to know which of the domain and codomain elements of the adjacency matrix are actually in use at any given time and should be considered in computing the relation. Consider two design ideas for this requirement.

1. Pushing the problem off to the user of the data type. In this case, the user would have to provide input sets that have elements known to be in use.
2. Save usage information that records which elements are in use and provide an interface for managing the element usage and status.

We use the first strategy here. This design places a burden on other code to keep track of the elements of the domain and codomain that are currently in use. For some operations, we use the term *universe* for the subset of the domain or codomain that is a bit-set containing those bits which are considered the valid subset and only those elements are considered in the computation. As we have seen, Relation Variables have the concept of a universe that tracks the tuples in use.

The other important consideration in the design is the need to be able to conveniently compute the preimage of a binary relation, *i.e.* we need to compute the converse relation of the association relation. Applying a set of codomain elements to the converse relation finds the corresponding elements of the domain. This is equivalent to multiplying a set of codomain elements by the transpose of the relation adjacency matrix.

In theory, the minimum necessary storage for the adjacency matrix is an array of bit-sets. The minimum array would be `domain_capacity` in length and each bit-set in the array would have `codomain_capacity` number of bits in the set. This arrangement is convenient enough when applying the association relation to a domain element set. It is just a matter of indexing into the array for the given element to obtain the codomain element set. However, going from the codomain to the domain implies iterating over the entire array, multiple times, to find all the associated domain elements and constructing the resultant bit-set.

This design directly stores the transpose of the relation adjacency matrix. Although storing the transpose of the matrix doubles the space required for the association adjacency matrix, we deem this a beneficial space / speed trade-off. Otherwise, we must either compute the transpose as necessary or traverse the adjacency matrix in a transpose-like order. This design uses additional space to limit additional computation. The decision does have the consequence that we store redundant data and must ensure there is no discrepancy in the redundant data.



One potential optimization is not to store either the relation or transpose matrix for those associations when there is no traversal in the corresponding direction. Although associations are considered traversable in either direction, in practice many times the actions of the processing do not actually use both directions. We do *not* consider such an optimization here, but record the idea for future consideration.

We define a public function used at `comptime` to create a type that will hold the adjacency matrix for the association.

```
<<binary_relation module public functions>>=
/// The `AssociationAdjMatrix` function is a type function that returns
/// a data type appropriate for tracking an association relationship
/// between two Relation Variables one of which serves as the `domain`
/// of the binary relation and the other as the `codomain`. The returned
/// type is a namespace `struct` whose container variables and constants
/// describe the association. The `domain` and `codomain` arguments
/// must be objects of a type returned by a `RelVar` function invocation.
/// The `domain_ref` and `codomain_ref` arguments give the multiplicity
/// and conditionality constraints on the association. _N.B._ the order
/// of the `codomain_ref` and `domain_ref` arguments. This is intended
/// to be mnemonic of the manner in which we write associations in line.
/// For example, the association A ---> B, the referential constraint for
/// domain elements is `.".@["1` as each *A* must be related to one *B* and
/// for codomain elements each *B* must be related to `.".@["+` *A*.
pub fn AssociationAdjMatrix(
    comptime DomainRelVar: type,
    comptime codomain_ref: RefConstraint,
    comptime domain_ref: RefConstraint,
    comptime CodomainRelVar: type,
    comptime initial_assocs: []const struct {
        DomainRelVar.InitialInstanceId, CodomainRelVar.InitialInstanceId},
) type {
    if (comptime !relvar.isRelVarType(DomainRelVar))
        @compileError("DomainRelVar argument is not a RelVar");
    if (comptime !relvar.isRelVarType(CodomainRelVar))
```

```

    @compileError("CodomainRelVar argument is not a RelVar");

    return struct {
        const Self = @This();

        <<association matrix variables>>
        <<association matrix declarations>>
        <<association matrix public functions>>
        <<association matrix private functions>>
    };
}

```

```

<<binary_relation module declarations>>=
/// Importing the RelationCapacity type from the relation variable module.
const relvar = @import("relvar.zig");
const RelVar = relvar.RelVar;
pub const RelationCapacity = relvar.RelationCapacity;

```

The adjacency matrix size is known at compile time. This is in keeping with a memory allocation strategy which does worst-case memory allocation known at compile time. For the purposes here, relation variable instances are simply equated to sequential natural numbers starting at zero.

The matrix multiplication implied by computing the image of the association relation is reduced to taking the bit-wise union of a sequence of bit-sets, *i.e.* with this data type we computing the dot products with bit-wise `or` operations on memory words.

It is convenient for this adjacency matrix type to have the defining dimensions of the adjacency matrix available as part of the type.

```

<<association matrix declarations>>=
pub const Domain = DomainRelVar;
pub const domain: DomainRelVar = .{};
const domain_capacity = DomainRelVar.relvar_capacity;
pub const domain_constraint = domain_ref;

pub const Codomain = CodomainRelVar;
pub const codomain: CodomainRelVar = .{};
const codomain_capacity = CodomainRelVar.relvar_capacity;
pub const codomain_constraint = codomain_ref;

```

We need some data types to handle the dimensions of the matrix.

```

<<association matrix declarations>>=
/// The `DomainTupleRefSet` type is a data type used to hold sets of domain
/// elements. This is used as input to computing the image of the binary
/// relation over the domain.
pub const DomainTupleRefSet = DomainRelVar.TupleRefSet;

```

```

const DomainAdjSet = @FieldType(DomainTupleRefSet, "tset");
const DomainTupleRef = DomainRelVar.TupleRef;

```

```

<<association matrix declarations>>=
/// The `CodomainTupleRefSet` is a data type used conversely to `DomainTupleRefSet`,
/// _i.e._ it holds codomain elements and is used as an input type for
/// computing the preimage of the binary relation over the codomain.
pub const CodomainTupleRefSet = CodomainRelVar.TupleRefSet;
const CodomainAdjSet = @FieldType(CodomainTupleRefSet, "tset");
const CodomainTupleRef = CodomainRelVar.TupleRef;

```

The empty sets are useful to have at hand.

```

<<association matrix declarations>>=
pub const emptyDomainTupleRefSet = DomainTupleRefSet.initEmpty();
const emptyDomainAdjSet = DomainAdjSet.initEmpty();
pub const emptyCodomainTupleRefSet = CodomainTupleRefSet.initEmpty();
const emptyCodomainAdjSet = CodomainAdjSet.initEmpty();

```

As discussed [previously](#), we store both the adjacency matrix for the binary relation and its transpose.

```

<<association matrix declarations>>=
/// `RelationMatrix` is a data type used to describe an adjacency
/// matrix used to compute the image of the domain under the association
/// relation.
const RelationMatrix = [domain_capacity]CodomainAdjSet;

/// `TransposeMatrix` is, conversely, a data type used to describe an
/// adjacency matrix used to compute the preimage of the codomain under
/// the association relation.
const TransposeMatrix = [codomain_capacity]DomainAdjSet;

```

Note that the **RelationMatrix** is an array of **CodomainAdjSet** and the **TransposeMatrix** is an array of **DomainAdjSet**.

These two types are used for the data fields in the **AssociationAdjMatrix** structure.

```

<<association matrix variables>>=
/// `relation_matrix` is an array where each element holds a bit-set
/// consisting of codomain elements that are associated to the domain
/// element of the array index.
var relation_matrix: RelationMatrix = initRelationMatrix();

```

```

/// `transpose_matrix` is an array where each element holds a bit-set
/// consisting of domain elements that are associated to the codomain
/// element of the array index.
var transpose_matrix: TransposeMatrix = initTransposeMatrix();

```

```

<<association matrix private functions>>=
fn initRelationMatrix() RelationMatrix {
    var rel_matrix: RelationMatrix = @splat(emptyCodomainAdjSet);
    inline for (initial_assocs) |initial_assoc| {
        const domain_index = @intFromEnum(initial_assoc@"0");
        const codomain_index = @intFromEnum(initial_assoc@"1");
        rel_matrix[domain_index].set(codomain_index) ;
    }

    return rel_matrix;
}

```

```

<<association matrix private functions>>=
fn initTransposeMatrix() TransposeMatrix {
    var xpose_matrix: TransposeMatrix = @splat(emptyDomainAdjSet);
    inline for (initial_assocs) |initial_assoc| {
        const domain_index = @intFromEnum(initial_assoc@"0");
        const codomain_index = @intFromEnum(initial_assoc@"1");
        xpose_matrix[codomain_index].set(domain_index) ;
    }

    return xpose_matrix;
}

```

```

<<binary_relation module tests>>=
const ami_tests = struct {
    const A = RelVar(
        struct { id: u8, which: u8 = 'A' },
        &.{&.{.id}},
        4,
        null,
        &.{.
            .{ .a0, .{ .id = 0 } },
            .{ .a1, .{ .id = 1 } },
            .{ .a2, .{ .id = 2 } },
            .{ .a3, .{ .id = 3 } },
        },
    );
    const B = RelVar(
        struct { id: u8, which: u8 = 'B' },
        &.{&.{.id}},
        4,
        null,
        &.{.
    );

```

```

        .{ .b0, .{ .id = 0 } },
        .{ .b1, .{ .id = 1 } },
        .{ .b2, .{ .id = 2 } },
        .{ .b3, .{ .id = 3 } },
    },
);

const R33 = AssociationAdjMatrix(
    A,
    @"1",
    @"1",
    B,
    &{},
);
const r33: R33 = .{};

}

```

```

<<binary_relation module tests>>=
test "adjacency matrix initialization" {
    const A = ami_tests.A;
    const B = ami_tests.B;
    const R33 = ami_tests.R33;
    _ = A.init();
    _ = B.init();

    try testing.expectEqual(@as(usize, 4), R33.domain_capacity);
    try testing.expectEqual(@as(usize, 4), R33.codomain_capacity);

    for (&R33.relation_matrix) /*set| {
        try testing.expectEqual(@as(usize, 0), set.count());
    }
    for (&R33.transpose_matrix) /*set| {
        try testing.expectEqual(@as(usize, 0), set.count());
    }
}

```

Functions to Relate and Unrelate Elements

One primary function of `AssociationAdjMatrix` is to keep track of ordered pairs of the binary relation. The functions in this section are used to implement the bookkeeping required to record the binary relation between domain and codomain elements.

```

<<association matrix public functions>>=
/// The `relate` function modifies the binary relation so as to bind the
/// `domain_element` to the `codomain_element` and vice versa_. After
/// invoking `relate`, computing the image of the binary relation on a
/// set that contains `domain_element` will include the `codomain_element`

```

```

/// in the result. Conversely, computing the preimage of the association
/// relation on a set that contains the `codomain_element` will include
/// the `domain_element` in the result.
pub fn relate(
    _: Self,
    domain_tuple: DomainTupleRef,
    codomain_tuple: CodomainTupleRef,
) void {
    domain_tuple.verifyAsNot(.free);
    codomain_tuple.verifyAsNot(.free);

    const domain_index = domain_tuple(tuple_id);
    const codomain_index = codomain_tuple(tuple_id);
    relation_matrix[domain_index].set(codomain_index) ; ①
    transpose_matrix[codomain_index].set(domain_index);
}

```

- ① Here we ensure that the redundant data implied by storing both the relation matrix and its transpose is maintained consistently. The code is remarkably symmetric.

```

<<binary_relation module tests>>=
test "relating elements" {
    const A = ami_tests.A;
    const B = ami_tests.B;

    const R33 = ami_tests.R33;
    const r33 = ami_tests.r33;

    const a0 = A.TupleRef.initFromInitialInstance(.a0);
    const b1 = B.TupleRef.initFromInitialInstance(.b1);
    r33.relate(a0, b1);
    try testing.expect(R33.relation_matrix[0].isSet(1));
    try testing.expect(R33.transpose_matrix[1].isSet(0));

    const a1 = A.TupleRef.initFromInitialInstance(.a1);
    const b2 = B.TupleRef.initFromInitialInstance(.b2);
    r33.relate(a1, b2);
    try testing.expect(R33.relation_matrix[1].isSet(2));
    try testing.expect(R33.transpose_matrix[2].isSet(1));
}

```

The inverse of relating an ordered pair is to unrelated them. Functions are provided for both a single pair and a slice of pairs.

```

<<association matrix public functions>>=
/// The `unrelate` function modifies the association relation so
/// as to remove the binding between the `domain_element` and the
/// `codomain_element`. After invoking `unrelate`, computing the

```

```

/// image of the association relation over a set that contains the
/// `domain_element` will _not_ include the `codomain_element` in
/// the result. Conversely, computing the preimage of the association
/// relation over a set containing the `codomain_element` will _not_
/// include the `domain_element` in the result.
pub fn unrelated(
    _: Self,
    domain_tuple: DomainTupleRef,
    codomain_tuple: CodomainTupleRef,
) void {
    domain_tuple.verifyAsNot(.free);
    codomain_tuple.verifyAsNot(.free);

    const domain_index = domain_tuple.tuple_id;
    const codomain_index = codomain_tuple.tuple_id;
    relation_matrix[domain_index].unset(codomain_index) ; ❶
    transpose_matrix[codomain_index].unset(domain_index);
}

```

- ❶ Again, there is redundant data between the relation matrix and the transpose matrix bit-sets that must be managed.

There are times when you need to unrelated a domain element and do not have handy its related codomain elements. For example, when a relation variable tuple is deleted, all its associations must also be deleted. Those association can always be found by computing the image. But this use case is common enough to warrant providing a function that finds any related codomain elements and unrelates them as part of the action to unrelated the domain element.

```

<<association matrix public functions>>=
// The `unrelateDomainTuple` function unrelateds `domain_element` from any
// codomain elements to which it might be related. The function returns
// the set of codomain elements that were unrelated.
pub fn unrelatedDomainTuple(
    _: Self,
    domain_tuple: DomainTupleRef,
) CodomainTupleRefSet {
    domain_tuple.verifyAs(.allocated);
    const domain_index = domain_tuple.tuple_id;
    var unrelated = emptyCodomainTupleRefSet;

    var codomain_iter = relation_matrix[domain_index].iterator(.{});
    while (codomain_iter.next() |codomain_index| {
        transpose_matrix[codomain_index].unset(domain_index);
        unrelated.set(codomain_index);
    }
    relation_matrix[domain_index] = emptyCodomainAdjSet; ❶

    return unrelated;
}

```

- ① We could call the `unrelate` function in the while loop, but since we are unrelated all codomain elements that might be related, we save access to the relation matrix and just set it to empty at the end.

What can be done for the domain needs also to be done for the codomain. The strategy is the same and the code shows the symmetry that occurs many times.

```
<<association matrix public functions>>=
// The `unrelateCodomainTuple` function unrelates `codomain_element` from any
// domain elements to which it might be related. The function returns
// the set of domain elements that were unrelated.
pub fn unrelateCodomainTuple(
    : Self,
    codomain_tuple: CodomainTupleRef,
) DomainTupleRefSet {
    codomain_tuple.verifyAs(.allocated);
    const codomain_index = codomain_tuple.tuple_id;
    var unrelated = emptyDomainTupleRefSet;

    var domain_iter = transpose_matrix[codomain_index].iterator(.{});
    while (domain_iter.next()) |domain_index| {
        relation_matrix[domain_index].unset(codomain_index);
        unrelated.set(domain_index);
    }
    transpose_matrix[codomain_index] = emptyDomainAdjSet;

    return unrelated;
}
```

```
<<binary_relation module tests>>=
test "unrelating instances" {
    const A = ami_tests.A;
    const B = ami_tests.B;

    const R23 = AssociationAdjMatrix(A, .@"?", .@"1", B, &.{});
    const r23: R23 = .{};

    const a0 = A.TupleRef.initFromInitialInstance(.a0);
    const b1 = B.TupleRef.initFromInitialInstance(.b1);
    r23.relate(a0, b1);
    try testing.expect(R23.relation_matrix[0].isSet(1));

    r23.unrelate(a0, b1);
    try testing.expect(!R23.relation_matrix[0].isSet(1));
    try testing.expect(!R23.transpose_matrix[1].isSet(0));

    const a1 = A.TupleRef.initFromInitialInstance(.a1);
    const b2 = B.TupleRef.initFromInitialInstance(.b2);
```

```

r23.relate(a1, b2);
try testing.expect(R23.relation_matrix[1].isSet(2));
try testing.expect(R23.transpose_matrix[2].isSet(1));

r23.unrelate(a1, b2);
try testing.expect(!R23.relation_matrix[1].isSet(2));
try testing.expect(!R23.transpose_matrix[2].isSet(1));

const b0 = B.TupleRef.initFromInitialInstance(.b0);
r23.relate(a0, b0);
r23.relate(a0, b1);
r23.relate(a0, b2);
try testing.expect(R23.relation_matrix[0].isSet(0));
try testing.expect(R23.relation_matrix[0].isSet(1));
try testing.expect(R23.relation_matrix[0].isSet(2));
try testing.expect(R23.transpose_matrix[0].isSet(0));
try testing.expect(R23.transpose_matrix[1].isSet(0));
try testing.expect(R23.transpose_matrix[2].isSet(0));

const ude_set = r23.unrelateDomainTuple(a0);
try testing.expect(ude_set.isSet(0));
try testing.expect(ude_set.isSet(1));
try testing.expect(ude_set.isSet(2));
try testing.expect(!R23.relation_matrix[0].isSet(0));
try testing.expect(!R23.relation_matrix[0].isSet(1));
try testing.expect(!R23.relation_matrix[0].isSet(2));
try testing.expect(!R23.transpose_matrix[0].isSet(0));
try testing.expect(!R23.transpose_matrix[1].isSet(0));
try testing.expect(!R23.transpose_matrix[2].isSet(0));

r23.relate(a0, b2);
r23.relate(a1, b2);
try testing.expect(R23.relation_matrix[0].isSet(2));
try testing.expect(R23.relation_matrix[1].isSet(2));
try testing.expect(R23.transpose_matrix[2].isSet(0));
try testing.expect(R23.transpose_matrix[2].isSet(1));

const uce_set = r23.unrelateCodomainTuple(b2);
try testing.expect(uce_set.isSet(0));
try testing.expect(uce_set.isSet(1));
try testing.expect(!R23.relation_matrix[0].isSet(2));
try testing.expect(!R23.relation_matrix[1].isSet(2));
try testing.expect(!R23.transpose_matrix[2].isSet(0));
try testing.expect(!R23.transpose_matrix[2].isSet(1));
}

```

Sometimes the execution situation makes it more convenient to update an adjacency matrix to change a relation rather than have to unrelated and then relate again elements. Trivial convenience functions are provided.

```

<<association matrix public functions>>=
/// The `rereleaseDomainTuple` function updates the binary relation
/// to unrelated `domain_element` and `curr_codomain_element` and relate
/// `domain_element` and new_codomain_element`.
pub fn rereleaseDomainTuple(
    self: Self,
    domain_tuple: DomainTupleRef,
    curr_codomain_tuple: CodomainTupleRef,
    new_codomain_tuple: CodomainTupleRef,
) void {
    self.unrelate(domain_tuple, curr_codomain_tuple);
    self.relate(domain_tuple, new_codomain_tuple);
}

```

The same circumstances hold for the codomain.

```

<<association matrix public functions>>=
/// The `relrelateCodomainTuple` function updates the association
/// relation to disassociate `codomain_element` and `curr_domain_element`
/// and create a new association between `codomain_element` and
/// new_domain_element`.
pub fn relrelateCodomainTuple(
    self: Self,
    codomain_tuple: CodomainTupleRef,
    curr_domain_tuple: DomainTupleRef,
    new_domain_tuple: DomainTupleRef,
) void {
    self.unrelate(curr_domain_tuple, codomain_tuple);
    self.relate(new_domain_tuple, codomain_tuple);
}

```

Applying the Association Relation

Until now, we have been presenting functions that perform little more than bookkeeping. Relating and unrelating elements is necessary to ensure the correct bits are set in the bit-sets used for the adjacency matrix. These functions are supplied to the data management mechanism managing the data.

However, the more interesting operations are those that apply the association relation, as encoded in the adjacency matrix, to compute the image of the domain under the relation. Since the adjacency matrix represents a binary relation between two sets, the result of computing the image or its preimage has well characterized properties. These properties are related to the multiplicity and conditionality of the binary relation.

Given a bit-set of domain elements, the following `image` function performs a union over the `relation_matrix` entries for each element in the `domain_set` argument. The result is a `CodomainTupleRefSet` contains the codomain elements related across the association to the domain elements in `domain_set`. When traversing an association starting with a single instance, the image of interest is for the subset of domain elements at the beginning of the relationship “chain”. Computing a subset of the total association image is logically the same as traversing an association

from a referring relation variable (*i.e.* the association participant that contains referential attributes that realize the association) to a referenced relation variable (*i.e.* the association participant that contains identifying attributes whose values match those of the referring relation variable).

```
<<association matrix public functions>>=
/// The `image` function computes the image of the binary relation over the
/// domain elements in `domain_set`. The function returns a `CodomainTupleRefSet`
/// referring to the related tuples.
pub fn image(
    _: Self,
    domain_set: DomainTupleRefSet
) CodomainTupleRefSet {
    var codomain_set = emptyCodomainAdjSet;
    var domain_iter = domain_set.tset.iterator(.{}));
    while(domain_iter.next()) |domain_element| {
        codomain_set.setUnion(relation_matrix[domain_element]);
    }
    return .initFromBitSet(codomain_set);
}
```

The complementary primitive function computes the domain elements given a set of codomain elements.

```
<<association matrix public functions>>=
/// The `preimage` function computes the preimage of the binary relation
/// over the codomain elements in `codomain_set`.
pub fn preimage(
    _: Self,
    codomain_set: CodomainTupleRefSet
) DomainTupleRefSet {
    var domain_set = emptyDomainAdjSet;
    var codomain_iter = codomain_set.tset.iterator(.{}));
    while(codomain_iter.next()) |codomain_element| {
        domain_set.setUnion(transpose_matrix[codomain_element]);
    }
    return .initFromBitSet(domain_set);
}
```

Also interesting is the complement of the image. This answers the question, “What are the elements of the codomain that are unrelated to a subset of the domain?” This helps in situations where the codomain represents the abstraction of a resource and the complement of the image gives the available resources.

```
<<association matrix public functions>>=
/// The `imageComplement` function computes the complement of the image
/// of the association relation over the `domain_set`. The returned
```

```

/// bit-set contains those codomain elements that are contained in the
/// `codomain_universe` but are _not_ related to any domain element in
/// `domain_set`.
pub fn imageComplement(
    self: Self,
    domain_set: DomainTupleRefSet,
) CodomainTupleRefSet {
    return self.image(domain_set)
        .complement()
        .intersectWith(codomain.selectAll());
}

```

```

<<association matrix public functions>>=
/// The `preimageComplement` function computes the complement of the
/// preimage of the association relation over the `codomain_set`.
/// The returned bit-set contains those domain elements that are
/// contained in the `domain_universe` but are _not_ related to any
/// codomain element in `codomain_set`.
pub fn preimageComplement(
    self: Self,
    codomain_set: CodomainTupleRefSet,
) DomainTupleRefSet {
    return self.preimage(codomain_set)
        .complement()
        .intersectWith(domain.selectAll());
}

```

The following tests show how an adjacency matrix is used to implement a *serial relation* using a `?--?` reflexive association. This can be considered as the binary relation equivalent of an ordered list.

```

<<binary_relation module tests>>=
test "images and preimages" {
    // Consider a reflexive association (also called a homogeneous binary relation)
    // that is A ?--? A. This association be used as a serial relation, imposing an
    // order on the domain elements, by properly relating the set elements.
    // Start with the sequence: 0 -- 1 -- 2 -- 3
    const A = ami_tests.A;
    const a: A = .{};

    const R44 = AssociationAdjMatrix(
        A,
        .@"?",
        .@ "?",
        A,
        &.{.
            .{ .a0, .a1 },
            .{ .a1, .a2 },

```

```

        .{ .a2, .a3 },
    }
);
const r44: R44 = .{};

try testing.expect(r44.isAssociationConsistent());
std.log.debug("{any}\n? -- ? Initial Order\n", .{r44});

// Note that the element at the head never appears in the
// set of related codomain elements and the element at the tail
// never appears in the set of related domain elements.
// For this example, 0 is the head of the sequence and 3
// is the tail.

// To never appear in the image, means the head must appear in the
// complement of the image.
var head_element = r44.imageComplement(a.selectAll());
try testing.expectEqual(@as(usize, 1), head_element.count());
try testing.expect(head_element.isSet(0));

// Inverting the logic computes the element at the tail.
var tail_element = r44.preimageComplement(a.selectAll());
try testing.expectEqual(@as(usize, 1), tail_element.count());
try testing.expect(tail_element.isSet(3));

// The element after the head is just the image of the head element,
// Image() is the successor function.
var after_head = r44.image(head_element);
try testing.expectEqual(@as(usize, 1), after_head.count());
try testing.expect(after_head.isSet(1));

// There is no element before the head.
// preimage() is the predecessor function.
var before_head = r44.preimage(head_element);
try testing.expectEqual(@as(usize, 0), before_head.count());

// There is no element after the tail.
var after_tail = r44.image(tail_element);
try testing.expectEqual(@as(usize, 0), after_tail.count());

// The element before the tail is 2.
var before_tail = r44.preimage(tail_element);
try testing.expectEqual(@as(usize, 1), before_tail.count());
try testing.expect(before_tail.isSet(2));

// By rerelating the elements in the sets, we can obtain a new ordering.
// Previous order: 0 -- 1 -- 2 -- 3
// New order:      0 -- 2 -- 1 -- 3
const a0 = A.TupleRef.initFromInitialInstance(.a0);
const a1 = A.TupleRef.initFromInitialInstance(.a1);
const a2 = A.TupleRef.initFromInitialInstance(.a2);
const a3 = A.TupleRef.initFromInitialInstance(.a3);
r44.rerelateDomainTuple(a0, a1, a2);      // 0 -- 1 is now 0 -- 2

```

```

r44.rerelateDomainTuple(a1, a2, a3);      // 1 -- 2 is now 1 -- 3
r44.rerelateDomainTuple(a2, a3, a1);      // 2 -- 3 is now 2 -- 1
try testing.expect(r44.isAssociationConsistent());
// Interestingly, the new ordering is created by flipping bit values.
std.log.debug("{any}\n? -- ? New Order\n", .{r44});

// Now the element after the head element is 2.
after_head = r44.image(head_element);
try testing.expectEqual(@as(usize, 1), after_head.count());
try testing.expect(after_head.isSet(2));

// The element before the tail is 1.
before_tail = r44.preimage(tail_element);
try testing.expectEqual(@as(usize, 1), before_tail.count());
try testing.expect(before_tail.isSet(1));
}

```

Referential Consistency

When the adjacency matrix is used to store a relationship, we need to be able to evaluate whether ongoing processing has updated the adjacency matrix in a manner that maintains the prescribed multiplicity and conditionality. Now we consider functions which perform the necessary counting of the binary relation to determine if the multiplicity and conditionality constraints are maintained. There are only four cases to consider. We encode the constraints as an enum.

```

<<binary_relation module declarations>>=
pub const RefConstraint = enum {
    @"?", // at most one, i.e. optional, 0 or 1
    @"1", // exactly one, i.e. one and only one
    @"+", // at least one, i.e. one or more
    @"*", // any, i.e. zero or more

    <<ref constraint declarations>>
};

```

The strategy for deciding if an binary relation has consistent multiplicity and conditionality properties is simply to count the number of related domain and codomain elements. Given the count, the determination of consistency is given by the following function.

```

<<ref constraint declarations>>=
/// The `evalConsistency` function determines if the value of the `ref_count`
/// argument is consistent with the referential constraint given by the
/// `constraint` argument.
fn evalConsistency(
    constraint: RefConstraint,
    ref_count: usize,
) bool {

```

```

    return switch(constraint) {
        .@ "?" => ref_count <= 1,
        .@ "1" => ref_count == 1,
        .@ "+" => ref_count >= 1,
        .@ "*" => true,
    };
}

```

To count the references, we iterate across the appropriate bit-set and use the `count` function of the bit-set to compare against the constraint.

```

<<association matrix public functions>>=
/// The `isDomainConsistent` function evaluates the referential consistency
/// for all elements of the `domain_universe` set. The `domain_universe`
/// represents those elements of the total domain set that are to be considered.
/// Each member of the `domain_universe` set has the number of codomain set
/// elements it references counted and then compared to the consistency
/// constraint defined for the domain of the association.
/// The function returns `true` if all `domain_universe` members reference the
/// proper number of codomain elements as given by the domain set
/// referential constraint and `false` otherwise.
fn isDomainConsistent(
    domain_universe: DomainTupleRefSet,
) bool {
    if (domain_constraint != .@ "*") { ❶
        var univ_iter = domain_universe.iterator(.{});
        while (univ_iter.next() |domain_tuple_ref| {
            const related_count = relation_matrix[domain_tuple_ref.tuple_id].count();
            const consistent = domain_constraint.evalConsistency(related_count);
            if (!consistent) {
                return false;
            }
        })
    }

    return true;
}

```

❶ If the constraint is `.@ "*"` the result is always `true` and there is no reason to perform the computation.

The counting for the codomain is complementary.

```

<<association matrix public functions>>=
/// The `isCodomainConsistent` function evaluates the referential consistency
/// for all elements of the `codomain_universe` set. The `codomain_universe`
/// represents those elements of the total codomain set that are to be considered.
/// Each member of the `codomain_universe` set has the number of domain set

```

```

/// elements it references counted and then compared to the consistency
/// constraint defined for the codomain of the association.
/// The function returns `true` if all `codomain_universe` members reference the
/// proper number of domain elements as given by the codomain set
/// referential constraint and `false` otherwise.
fn isCodomainConsistent(
    codomain_universe: CodomainTupleRefSet,
) bool {
    if (codomain_constraint != .@") {
        var univ_iter = codomain_universe.iterator(.{});
        while (univ_iter.next()) |codomain_tuple_ref| {
            const related_count = transpose_matrix[codomain_tuple_ref.tuple_id].count();
            const consistent = codomain_constraint.evalConsistency(related_count);
            if (!consistent) {
                return false;
            }
        }
    }

    return true;
}

```

The conjunction of the domain consistency and the codomain consistency determines the consistency for the association.

```

<<association matrix public functions>>=
/// The `isAssociationConsistent` function returns `true` if both the
/// `domain_universe` and `codomain_universe` sets are consistent with
/// their given constraints and `false` otherwise.
pub fn isAssociationConsistent(
    _: Self,
) bool {
    //const domain_consistent = isDomainConsistent(domain.selectAll());
    //const codomain_consistent = isCodomainConsistent(codomain.selectAll());
    //std.debug.print("domain = {}, codomain = {}\n", .{ domain_consistent,
codomain_consistent });
    //return domain_consistent and codomain_consistent;
    return isDomainConsistent(domain.selectAll()) and
        isCodomainConsistent(codomain.selectAll());
}

```

Platform Specific Considerations

The choice of an adjacency matrix as the implementation mechanism affects the way in which we must construct associations for those classes which use an associative class to formalize the association. One view of an associative class is that all associations use them to hold the ordered pairs which underlie the binary relation. As we have seen, associations that are mathematical functions and for which the associative class has no additional attributes or behavior can be simplified by eliminating the associative class and incorporating referential attributes into the

domain class.

The particular characteristics of an adjacency matrix means that associative classes for all associations may be eliminated as long as the association class has no additional attributes beyond those referential attributes that formalize the association and has no other behavior.

For those cases where the associative class either participates in other relationships, has descriptive attributes of its own, or has a state model for its life cycle, it is necessary when using an adjacency matrix to implement associations, to *decompose* the association into two associations and have an adjacency matrix for each part of the decomposition. This is because the adjacency matrix itself has no way to carry other data in the manner an associative class can.

This characteristic of the adjacency matrix implementation implies that when the mapping from the model onto the platform specific mechanisms, the mapping must accommodate the two types of association classes differently.

Debugging Support

To aid in visualization at debug time, we provide a way to print a simple character based rendition of the association matrix. The printout uses an “x” character to mark locations in the bit-set that are not in the domain or codomain sets and a “.” character for those that are included. If two instances are associated a “1” is printed in the corresponding cell, otherwise a “-” is printed. Rows and columns are labeled with the value of the domain or codomain element index modulo 10. The type name of the matrix is printed and an optional title are printed at the end.

Example Adjacency Matrix Printout

```
=====
 0 1 2 3 4 5 6 7 8 9
 . . . . x x x x x
0 . - 1 - - - - - -
1 . - 1 - - - - - -
2 . - - 1 - - - - -
3 . - - - 1 - - - -
4 . - - - - - - - -
5 x - - - - - - - -
6 x - - - - - - - -
7 x - - - - - - - -
8 x - - - - - - - -
9 x - - - - - - - -
=====
```

```
<<association matrix public functions>>=
pub fn format(
    : Self,
    writer: *std.Io.Writer,
) std.Io.Writer.Error!void {
    const space_per_cell: usize = 2;
    const lead_in_space: usize = 3;
    var leader: [lead_in_space]u8 = @splat(' ');
    const boundary_len = (Self.codomain_capacity * space_per_cell) +
```

```

        lead_in_space;
const boundary: [boundary_len]u8 = @splat('=');

try writer.print("{s}\n", .{boundary});

var row_buf: [codomain_capacity * space_per_cell]u8 = @splat(' ');

for (0..codomain_capacity) |codomain_element| {
    const digit = fmt.digitToChar(@intCast(codomain_element % 10), .lower);
    row_buf[codomain_element * space_per_cell + 1] = digit;
}
try writer.print("{s}{s}\n", .{leader, row_buf});

for (0..codomain_capacity) |codomain_element| {
    row_buf[codomain_element * space_per_cell + 1] =
        if (codomain.selectAll().isSet(codomain_element)) '.' else 'x';
}
try writer.print("{s}{s}\n", .{leader, row_buf});

for (0..domain_capacity) |domain_element| {
    leader[0] = fmt.digitToChar(@intCast(domain_element % 10), .lower);
    leader[2] = if (domain.selectAll().isSet(domain_element)) '.' else 'x';
    for (0..codomain_capacity) |codomain_element| {
        var mark: u8 = undefined;
        if (relation_matrix[domain_element].isSet(codomain_element)) {
            mark = '1';
            assert(transpose_matrix[codomain_element].isSet(domain_element));
        } else {
            mark = '-';
            assert(!relation_matrix[domain_element].isSet(codomain_element));
        }
        row_buf[codomain_element * space_per_cell + 1] = mark;
    }
    try writer.print("{s}{s}\n", .{leader, row_buf});
}

try writer.print("{s}", .{boundary});
}

```

Adjacency Matrices for Generalizations

In this section, we show how generalization relationships are implemented using adjacency matrices. From the previous discussion, generalizations are characterized by the partitioning the domain of the relationships such that the partitions form a disjoint union. We reiterate that this usage of the term, generalization, does not imply any form of type inheritance.

A generalization assumes that the domain for the generalization is partitioned into disjoint sets. For this implementation, partitioning implies that the domain element numbering is only unique within a particular partition. For example, if there are three domain partitions, call them **A**, **B**, and **C**, then the cardinality of all three may differ and each starts their element numbering at 0. You might be tempted to assume that the number of elements in each domain partition must equal the number of codomain elements. This would be the case if all domain partitions have an equal probability of

being related to a codomain element and generalizations are modified at run-time. In general, it is not true because static populations of domain elements can be defined at system build time.

Representing a Generalization

Following the pattern used for associations, a type function is supplied that returns a type to hold a partitioned generalization adjacency matrix.

```
<<binary_relation module public functions>>=
/// The `GeneralizationAdjMatrix` function is a type function that
/// returns a namespace `struct` data type which is appropriate to track a
/// generalization relationship between a set of relation variables which
/// form a partitioned domain and a relation variable that serves as the
/// codomain of the generalization.
/// The `codomain` argument must be a pointer to relation variable object
/// that represents the codomain of the generalization.
/// The `partitions` argument must be a list of pairs. The first member
/// of the pair must be an enumeration literal giving a tag used to
/// identify the partition. The second member of the pair must be a
/// pointer to a relation variable object that represents one of the
/// partitions of the domains of the generalization.
/// The generalization is initialized
/// using the `initial_gens` argument. This argument is a three field
/// tuple. The first field is an enumeration literal that corresponds
/// to one of the domain partition tags given in the `partitions`
/// argument. The second field of the triple is an enumeration literal
/// corresponding to an `InitialInstanceId` for a tuple in the domain
/// partition referenced by the first field. The third element is an
/// `InitialInstanceId` for a tuple in the codomain.
pub fn GeneralizationAdjMatrix(
    comptime CodomainRelVar: type,
    comptime partitions: []const struct { @Type(.enum_literal), type },
    comptime initial_gens: []const struct {
        @Type(.enum_literal), // partition tag
        @Type(.enum_literal), // InitialInstanceId tag value corresponding to the partition
        CodomainRelVar.InitialInstanceId, // InitialInstanceId enum value for the codomain
    },
) type {
    if (comptime !relvar.isRelVarType(CodomainRelVar))
        @compileError("CodomainRelVar argument is not a RelVar");
    if (partitions.len < 2)
        @compileError("generalization adj matrix requires at least 2 domain partitions");
    inline for (partitions) |partition_desc| {
        const PartitionDomain = partition_desc.@"1";
        if (!relvar.isRelVarType(PartitionDomain))
            @compileError("domain partition '" ++ @tagName(partition_desc.@"0") ++ "' is not
a RelVar");
    }

    return struct {
```

```

const Self = @This();

<<generalization matrix variables>>
<<generalization matrix declarations>>
<<generalization matrix public functions>>
<<generalization matrix private functions>>
};

}

```

This implementation adopts the view that a generalization is just a set of **?--1** associations between each of the domain partitions and the codomain, coupled with a rule about the disjointness of the domain partitions. This simplifies, in some sense, the code used to manage a Generalization since the code for Associations can be used for individual partitions.

The data types of a domain partitions for a generalization are dependent upon the types of Relation Variables used for the individual partitions. We need to have ready access to the partition domains. It is not possible to hold the partition associations is a simple array since they are not all of the same data type. The solution used here is to create a tagged union type which can then be stored in an array.

Domain partitions are known by an enumerated type. The values of the enumeration are the same as the `class_id` values of the relation variables that make up the partition. Thus, if a partition contains a relation variable whose `class_id` value is `.apple`, then the partition that contains the `.apple` relational variable is also called `.apple`. The type of the enumeration is `PartitionId` and is obtained from the tagged catalog containing the domain partitions

```

<<generalization matrix declarations>>=
/// An enumeration type used to identify the partitions of the generalization.
pub const PartitionId = TagPartitions();

```

```

<<generalization matrix private functions>>=
fn TagPartitions() type {
    const EnumField = std.builtin.Type.EnumField;
    var fields: []const EnumField = &[_]EnumField{};
    inline for (partitions, 0..) |partition_desc, partition_index| {
        fields = fields ++ &[_]EnumField{
            .{ .name = @tagName(partition_desc.@"0"), .value = partition_index }
        };
    }

    return @Type(.{
        .@"enum" = .{
            .tag_type = u8,
            .fields = fields,
            .decls = &.{},
            .is_exhaustive = true,
        },
    });
}

```

```
<<generalization matrix declarations>>=
const PartitionEntry = DefinePartitionEntry();
```

```
<<generalization matrix private functions>>=
fn DefinePartitionEntry() type {
    const UnionField = std.builtin.Type.UnionField;
    var fields: []const UnionField = &[_]UnionField{};
    inline for (partitions) |partition_desc| {
        const AdjMatrixType = MakePartitionAdjMatrix(partition_desc);
        fields = fields ++ &[_]UnionField{.
            .name = @tagName(partition_desc.@"0"),
            .type = AdjMatrixType,
            .alignment = @alignOf(AdjMatrixType),
        }};
    }

    return @Type(.{
        .@"union" = .{
            .layout = .auto,
            .tag_type = PartitionId,
            .fields = fields,
            .decls = &.{},
        },
    });
}

fn MakePartitionAdjMatrix(
    comptime partition_desc: anytype,
) type {
    const partition_id = partition_desc.@"0";
    const PartitionRelVar = partition_desc.@"1";
    const InitialInstSpec = struct {
        PartitionRelVar.InitialInstanceId,
        CodomainRelVar.InitialInstanceId,
    };
    var initial_pop: []const InitialInstSpec = &[_]InitialInstSpec{};
    inline for (initial_gens) |initial_gen| {
        if (initial_gen.@"0" == partition_id) {
            initial_pop = initial_pop ++ &[_]InitialInstSpec{
                .{ initial_gen.@"1", initial_gen.@"2" },
            };
        }
    }
    return AssociationAdjMatrix(
        PartitionRelVar,
        .@"?",
        .@"1",
        CodomainRelVar,
        initial_pop,
    );
}
```

```
<<generalization matrix declarations>>=
pub const PartitionsCatalog = enums.EnumArray(PartitionId, PartitionEntry);
```

```
<<generalization matrix variables>>=
const partitions_catalog = initPartitionsCatalog();
```

```
<<generalization matrix private functions>>=
fn initPartitionsCatalog() PartitionsCatalog {
    var catalog: PartitionsCatalog = .initUndefined();
    inline for (partitions) |partition_desc| {
        const partition_name = @tagName(partition_desc.@"0");
        const AdjMatrixType = @FieldType(PartitionEntry, partition_name);
        const partition_adj_matrix: AdjMatrixType = .{};
        catalog.set(
            @field(PartitionId, partition_name),
            @unionInit(PartitionEntry, partition_name, partition_adj_matrix),
        );
    }
    return catalog;
}
```

Similar to AssociationAdjMatrix types, it is necessary to be able to construct bit-sets in order to manage a GeneralizationAdjMatrix type. We include class declarations to handle domain and codomain sets and to have easy reference to the element counts. We start with the codomain.

```
<<generalization matrix variables>>=
pub const codomain: CodomainRelVar = .{};
```

```
<<generalization matrix declarations>>=
pub const CodomainTupleRef = CodomainRelVar.TupleRef;
```

```
<<generalization matrix declarations>>=
pub const CodomainTupleRefSet = CodomainRelVar.TupleRefSet;
```

```
<<generalization matrix declarations>>=
pub const emptyCodomainTupleRefSet: CodomainTupleRefSet = .initEmpty();
```

Because each bit-set for a domain partition is potentially a different data type, obtaining the bit-set type for a domain partition now becomes a `comptime` type function.

```
<<generalization matrix private functions>>=
fn DomainRelVar(
    comptime partition_id: PartitionId,
```

```
) type {
    const partition_adj_matrix = get_partition_adj_matrix(partition_id);
    const PartitionAdjMatrix = @TypeOf(partition_adj_matrix);
    return @TypeOf(PartitionAdjMatrix.domain);
}
```

```
<<generalization matrix public functions>>=
pub fn DomainTupleRef(
    comptime partition_id: PartitionId,
) type {
    return DomainRelVar(partition_id).TupleRef;
}
```

```
<<generalization matrix public functions>>=
pub fn DomainTupleRefSet(
    comptime partition_id: PartitionId,
) type {
    return DomainRelVar(partition_id).TupleRefSet;
}
```

An empty domain set also require a type function.

```
<<generalization matrix public functions>>=
pub fn emptyDomainTupleRefSet(
    comptime partition_id: PartitionId,
) DomainTupleRefSet(partition_id) {
    return DomainTupleRefSet(partition_id).initEmpty();
}
```

```
<<generalization matrix public functions>>=
pub fn get_partition_assoc(
    _: Self,
    comptime partition_id: PartitionId,
) @FieldType(PartitionEntry, @tagName(partition_id)) {
    return get_partition_adj_matrix(partition_id);
}
```

```
<<generalization matrix private functions>>=
fn get_partition_adj_matrix(
    comptime partition_id: PartitionId,
) @FieldType(PartitionEntry, @tagName(partition_id)) {
    return @field(partitions_catalog.get(partition_id), @tagName(partition_id));
}
```

Tests

```
<<binary_relation module tests>>=
const vehicles = struct {
    const Vehicle = RelVar(
        struct {
            id: u8,
            make: []const u8,
            model: []const u8,
        },
        &.{{&.{.id}}},
        10,
        null,
        &.{.
            .{ .car1 , .{ .id = 1, .make = "Ford", .model = "F150" } },
            .{ .car2 , .{ .id = 2, .make = "Toyota", .model = "Camry" } },
            .{ .car3 , .{ .id = 3, .make = "Honda", .model = "Accord" } },
            .{ .truck1 , .{ .id = 4, .make = "Peterbuilt", .model = "P-1000" } },
            .{ .truck2 , .{ .id = 5, .make = "Kenworth", .model = "K-1000" } },
            .{ .bus1 , .{ .id = 6, .make = "Champion Bus", .model = "Challenger" } },
            .{ .bus2 , .{ .id = 7, .make = "Featherlite Coaches", .model = "Vantare" } },
        },
    );
    const vehicle: Vehicle = .{};

    const Car = RelVar(
        struct { id: u8, door_count: u8 },
        &.{{&.{.id}}},
        5,
        null,
        &.{.
            .{ .car1 , .{ .id = 1, .door_count = 2 } },
            .{ .car2 , .{ .id = 2, .door_count = 4 } },
            .{ .car3 , .{ .id = 3, .door_count = 4 } },
        },
    );
    const car: Car = .{};

    const Truck = RelVar(
        struct { id: u8, gross_weight: u32 },
        &.{{&.{.id}}},
        5,
        null,
        &.{.
            .{ .truck1 , .{ .id = 4, .gross_weight = 40_000 } },
            .{ .truck2 , .{ .id = 5, .gross_weight = 35_000 } },
        },
    );
    const truck: Truck = .{};

    const Bus = RelVar(
        struct { id: u8, passenger_capacity: u8 },

```

```

    &.{&.{.id}},  

    10,  

    null,  

    &.{  

        .{ .bus1 , .{ .id = 6, .passenger_capacity = 50 } },  

        .{ .bus2 , .{ .id = 7, .passenger_capacity = 25 } },  

    },  

);  

const bus: Bus = .{};  
  

const R1 = GeneralizationAdjMatrix(  

    Vehicle,  

    &.{  

        .{ .car, Car },  

        .{ .truck, Truck },  

        .{ .bus, Bus },  

    },  

    &.{  

        .{ .car, .car1, .car1 },  

        .{ .car, .car2, .car2 },  

        .{ .car, .car3, .car3 },  

        .{ .truck, .truck1, .truck1 },  

        .{ .truck, .truck2, .truck2 },  

        // N.B. buses are not part of the initial generalization.  

        // Tests below add them.  

    },  

);
const r1: R1 = .{};  
  

pub fn init() void {  

    _ = Vehicle.init();  

    _ = Car.init();  

    _ = Truck.init();  

    _ = Bus.init();  

}
};
```

```

<<binary_relation module tests>>=
test "generalization initialization" {
    vehicles.init();

    const r1 = vehicles.r1;

    const car_adj_matrix = r1.get_partition_assoc(.car);
    try testing.expectEqual(@as(usize, 5), @TypeOf(car_adj_matrix).domain_capacity);
    try testing.expectEqual(@as(usize, 10), @TypeOf(car_adj_matrix).codomain_capacity);

    const truck_adj_matrix = r1.get_partition_assoc(.truck);
    try testing.expectEqual(@as(usize, 5), @TypeOf(truck_adj_matrix).domain_capacity);
    try testing.expectEqual(@as(usize, 10), @TypeOf(truck_adj_matrix).codomain_capacity);
```

```

const bus_adj_matrix = r1.get_partition_assoc(.bus);
try testing.expectEqual(@as(usize, 10), @TypeOf(bus_adj_matrix).domain_capacity);
try testing.expectEqual(@as(usize, 10), @TypeOf(bus_adj_matrix).codomain_capacity);
}

```

Functions to Relate and Unrelate Elements

When relating domain elements to codomain elements, for the generalization we must include the partition number.

```

<<generalization matrix public functions>>=
/// The `relate` function modifies the generalization relation so as to
/// bind the `domain_element` in `partition` to the `codomain_element`
/// and _vice versa_. After invoking `relate`, computing the image of
/// the generalization on the `partition` of the domain which contains
/// `domain_element` will include the `codomain_element` in the result.
/// Conversely, computing the preimage of the generalization for
/// `partition` on a set that contains the `codomain_element` will
/// include the `domain_element` in the result.
pub fn relate(
    _: Self,
    comptime partition_id: PartitionId,
    domain_tuple: DomainTupleRef(partition_id),
    codomain_tuple: CodomainTupleRef,
) void {
    const partition_assoc = get_partition_adj_matrix(partition_id);
    partition_assoc.relate(domain_tuple, codomain_tuple);
}

```

Notably, the `relate` function uses the `relate` function of the `AssociationAdjMatrix` to perform the work on the selected domain partition.

The corresponding un-relate functions are also provided.

```

<<generalization matrix public functions>>=
pub fn unrelated(
    _: Self,
    comptime partition_id: PartitionId,
    domain_tuple: DomainTupleRef(partition_id),
    codomain_tuple: CodomainTupleRef,
) void {
    const partition_assoc = get_partition_adj_matrix(partition_id);
    partition_assoc.unrelate(domain_tuple, codomain_tuple);
}

```

Tests

```
<<binary_relation module tests>>=
test "generalization relating" {
    const r1 = vehicles.r1;
    const vehicle = vehicles.vehicle;
    const Vehicle = vehicles.Vehicle;
    const bus = vehicles.bus;
    const Bus = vehicles.Bus;

    const bus_1 = Bus.TupleRef.initFromInitialInstance(.bus1);
    const vehicle_bus_1 = Vehicle.TupleRef.initFromInitialInstance(.bus1);
    const bus_assoc = r1.get_partition_assoc(.bus);

    r1.relate(.bus, bus_1, vehicle_bus_1);
    try testing.expect(@TypeOf(bus_assoc)
        .relation_matrix[@intFromEnum(@TypeOf(bus).InitialInstanceId.bus1)]
        .isSet(@intFromEnum(@TypeOf(vehicle).InitialInstanceId.bus1)));
    try testing.expect(@TypeOf(bus_assoc)
        .transpose_matrix[@intFromEnum(@TypeOf(vehicle).InitialInstanceId.bus1)]
        .isSet(@intFromEnum(@TypeOf(bus).InitialInstanceId.bus1)));

    r1.unrelate(.bus, bus_1, vehicle_bus_1);
    try testing.expect(!@TypeOf(bus_assoc)
        .relation_matrix[@intFromEnum(@TypeOf(bus).InitialInstanceId.bus1)]
        .isSet(@intFromEnum(@TypeOf(vehicle).InitialInstanceId.bus1)));
    try testing.expect(!@TypeOf(bus_assoc)
        .transpose_matrix[@intFromEnum(@TypeOf(vehicle).InitialInstanceId.bus1)]
        .isSet(@intFromEnum(@TypeOf(bus).InitialInstanceId.bus1)));

    const bus_2 = Bus.TupleRef.initFromInitialInstance(.bus2);
    const vehicle_bus_2 = Vehicle.TupleRef.initFromInitialInstance(.bus2);

    r1.relate(.bus, bus_2, vehicle_bus_2);
    try testing.expect(@TypeOf(bus_assoc)
        .relation_matrix[@intFromEnum(@TypeOf(bus).InitialInstanceId.bus2)]
        .isSet(@intFromEnum(@TypeOf(vehicle).InitialInstanceId.bus2)));
    try testing.expect(@TypeOf(bus_assoc)
        .transpose_matrix[@intFromEnum(@TypeOf(vehicle).InitialInstanceId.bus2)]
        .isSet(@intFromEnum(@TypeOf(bus).InitialInstanceId.bus2)));

    r1.unrelate(.bus, bus_2, vehicle_bus_2);
    try testing.expect(!@TypeOf(bus_assoc)
        .relation_matrix[@intFromEnum(@TypeOf(bus).InitialInstanceId.bus2)]
        .isSet(@intFromEnum(@TypeOf(vehicle).InitialInstanceId.bus2)));
    try testing.expect(!@TypeOf(bus_assoc)
        .transpose_matrix[@intFromEnum(@TypeOf(vehicle).InitialInstanceId.bus2)]
        .isSet(@intFromEnum(@TypeOf(bus).InitialInstanceId.bus2)));
}
```

There are use cases in dealing with generalizations where we need to know which domain partition contains the domain element that is related to a given codomain element. We would like to know the domain partition without having to compute the preimage of each partition and determine which one is not empty. A sequential search is sufficient as we do not expect large numbers of domain partitions.

```
<<generalization matrix public functions>>=
/// The `classify` function determines which domain partition holds the
/// domain element related to `codomain_element`.
pub fn classify(
    _: Self,
    codomain_tuple: CodomainTupleRef,
) PartitionId {
    const partition_ids = comptime enums.values(PartitionId);
    inline for (partition_ids) |partition_id| {
        const partition_assoc = get_partition_adj_matrix(partition_id);
        if (@TypeOf(partition_assoc).transpose_matrix[codomain_tuple.tuple_id].count() !=
0) {
            return partition_id;
        }
    }

    unreachable;
}
```

As with associations, there are times when only the domain element (and its partition) are known and we wish to have the generalization code find the related codomain element for us.

```
<<generalization matrix public functions>>=
pub fn unrelateDomainTuple(
    _: Self,
    comptime partition_id: PartitionId,
    domain_tuple: DomainTupleRef(partition_id),
) CodomainTupleRefSet {
    domain_tuple.verifyAsNot(.free);
    const partition_adj_matrix = get_partition_adj_matrix(partition_id);
    return partition_adj_matrix.unrelateDomainTuple(domain_tuple);
}
```

When unrelating a codomain element in isolation, the generalization code must first find the partition which contains the domain element to which the codomain element is related.

```
<<generalization matrix public functions>>=
pub fn unrelateCodomainTuple(
    _: Self,
    codomain_tuple: CodomainTupleRef,
```

```

) void {
    codomain_tuple.verifyAsNot(.free);
    const codomain_element = codomain_tuple.tuple_id;
    const partition_ids = comptime.enums.values(PartitionId);
    inline for (partition_ids) |partition_id| {
        const partition_adj_matrix = get_partition_adj_matrix(partition_id);
        if (@TypeOf(partition_adj_matrix).transpose_matrix[codomain_element]
            .findFirstSet()) |domain_element|
    {
        const Domain = @TypeOf(partition_adj_matrix).Domain;
        const domain_tuple = Domain.TupleRef.initFromIndex(domain_element);
        partition_adj_matrix.unrelate(domain_tuple, codomain_tuple);
        return;
    }
}
unreachable;
}

```

Tests

```

<<binary_relation module tests>>=
test "generalization unrelating elements" {
    const r1 = vehicles.r1;
    const vehicle = vehicles.vehicle;
    const Vehicle = vehicles.Vehicle;
    const car = vehicles.car;
    const Car = vehicles.Car;

    // Check they are already related.
    // The initial generalizations perform relating action.
    const car_assoc = r1.get_partition_assoc(.car);
    try testing.expect(@TypeOf(car_assoc)
        .relation_matrix[@intFromEnum(@TypeOf(car).InitialInstanceId.car3)]
        .isSet(@intFromEnum(@TypeOf(vehicle).InitialInstanceId.car3)));
    try testing.expect(@TypeOf(car_assoc)
        .transpose_matrix[@intFromEnum(@TypeOf(vehicle).InitialInstanceId.car3)]
        .isSet(@intFromEnum(@TypeOf(car).InitialInstanceId.car3)));

    try testing.expect(@TypeOf(car_assoc)
        .relation_matrix[@intFromEnum(@TypeOf(car).InitialInstanceId.car2)]
        .isSet(@intFromEnum(@TypeOf(vehicle).InitialInstanceId.car2)));
    try testing.expect(@TypeOf(car_assoc)
        .transpose_matrix[@intFromEnum(@TypeOf(vehicle).InitialInstanceId.car2)]
        .isSet(@intFromEnum(@TypeOf(car).InitialInstanceId.car2)));

    const car_3 = Car.TupleRef.initFromInitialInstance(.car3);
    _ = r1.unrelateDomainTuple(.car, car_3);

    try testing.expect(!@TypeOf(car_assoc)

```

```

    .relation_matrix[@intFromEnum(@TypeOf(car).InitialInstanceId.car3)]
    .isSet(@intFromEnum(@TypeOf(vehicle).InitialInstanceId.car3)));
try testing.expect(!@TypeOf(car_assoc)
    .transpose_matrix[@intFromEnum(@TypeOf(vehicle).InitialInstanceId.car3)]
    .isSet(@intFromEnum(@TypeOf(car).InitialInstanceId.car3)));

const vehicle_car_2 = Vehicle.TupleRef.initFromInitialInstance(.car2);
r1.unrelateCodomainTuple(vehicle_car_2);

try testing.expect(!@TypeOf(car_assoc)
    .relation_matrix[@intFromEnum(@TypeOf(car).InitialInstanceId.car2)]
    .isSet(@intFromEnum(@TypeOf(vehicle).InitialInstanceId.car2)));
try testing.expect(!@TypeOf(car_assoc)
    .transpose_matrix[@intFromEnum(@TypeOf(vehicle).InitialInstanceId.car2)]
    .isSet(@intFromEnum(@TypeOf(car).InitialInstanceId.car2)));
}

```

Another common generalization operation is to have a subclass instance *migrate* to another subclass of the generalization. Migration can be used to model modal behavior. The operation is logically equivalent to a domain element being unrelated from its codomain element followed by a new domain element in (presumably) a new partition being related to the original codomain element. That is the strategy used in the implementation below. This function adds the convenience of finding the codomain element without the caller having to supply it.

```

<<generalization matrix public functions>>=
/// The `reclassify` function modifies the generalization so as to unrelate
/// the `prev_partition_id` and `prev_domain_element` from its related codomain
/// element and establish a new generalization to the same codomain element
/// given by `new_partition_id` and `new_domain_element`.
pub fn reclassify(
    _: Self,
    comptime prev_partition_id: PartitionId,
    prev_domain_tuple: DomainTupleRef(prev_partition_id),
    comptime new_partition_id: PartitionId,
    new_domain_tuple: DomainTupleRef(new_partition_id),
) void {
    const prev_partition_adj_matrix = get_partition_adj_matrix(prev_partition_id);
    const codomain_set = @TypeOf(prev_partition_adj_matrix)
        .relation_matrix[prev_domain_tuple.tuple_id];
    const codomain_element = codomain_set.findFirstSet()?.?; ①
    const codomain_tuple = @TypeOf(codomain).TupleRef.initFromIndex(codomain_element);

    prev_partition_adj_matrix.unrelate(prev_domain_tuple, codomain_tuple);

    const new_partition_adj_matrix = get_partition_adj_matrix(new_partition_id);
    new_partition_adj_matrix.relate(new_domain_tuple, codomain_tuple);
}

```

① If the generalization is referentially consistent, there must be exactly one codomain element related to the `prev_domain_element`.

Tests

```
<<binary_relation module tests>>=
const classify_tests = struct {
    const Super = RelVar(
        struct {
            name: []const u8,
        },
        &.{{&{.name}}},
        10,
        null,
        &.{{
            .{ .a , .{ .name = "a" } },
            .{ .aa , .{ .name = "aa" } },
            .{ .b , .{ .name = "b" } },
            .{ .bb , .{ .name = "bb" } },
            .{ .c , .{ .name = "c" } },
            .{ .cc , .{ .name = "cc" } },
        }},
    );
    const super: Super = .{};

    const Sub_a = RelVar(
        struct {
            name: []const u8,
        },
        &.{{&{.name}}},
        10,
        null,
        &.{{
            .{ .a , .{ .name = "a" } },
            .{ .aa , .{ .name = "aa" } },
        }},
    );
    const sub_a: Sub_a = .{};

    const Sub_b = RelVar(
        struct {
            name: []const u8,
        },
        &.{{&{.name}}},
        10,
        null,
        &.{{
            .{ .b , .{ .name = "b" } },
            .{ .bb , .{ .name = "bb" } },
        }},
    );
    const sub_b: Sub_b = .{};

    const Sub_c = RelVar(
        struct {
```

```

        name: []const u8,
    },
    &.{&.{ .name }},
    10,
    null,
    &.{.
        .{ .c , .{ .name = "c" } },
        .{ .cc , .{ .name = "cc" } },
    },
),
const sub_c: Sub_c = .{};

const R2 = GeneralizationAdjMatrix(
    Super,
    &.{.
        .{ .sub_a, Sub_a },
        .{ .sub_b, Sub_b },
        .{ .sub_c, Sub_c },
    },
    &.{.
        .{ .sub_a, .a, .a },
        .{ .sub_a, .aa, .aa },
        .{ .sub_b, .b, .b },
        .{ .sub_b, .bb, .bb },
        .{ .sub_c, .c, .c },
        .{ .sub_c, .cc, .cc },
    },
);
const r2: R2 = .{};

pub fn init() void {
    _ = Super.init();
    _ = Sub_a.init();
    _ = Sub_b.init();
    _ = Sub_c.init();
}
};

```

```

<<binary_relation module tests>>=
test "generalization reclassification" {
    classify_tests.init();
    const Super = classify_tests.Super;
    const A = classify_tests.Sub_a;
    const C = classify_tests.Sub_c;
    const r2 = classify_tests.r2;

    const a_a = A.TupleRef.initFromInitialInstance(.a);
    const c_c = C.TupleRef.initFromInitialInstance(.c);

    r2.reclassify(.sub_a, a_a, .sub_c, c_c);
}

```

```

// a_a is now unrelated to super.a
const sub_a_assoc = r2.get_partition_assoc(.sub_a);
try testing.expect(!@TypeOf(sub_a_assoc)
    .relation_matrix[@intFromEnum(A.InitialInstanceId.a)]
    .isSet(@intFromEnum(Super.InitialInstanceId.a)));
try testing.expect(!@TypeOf(sub_a_assoc)
    .transpose_matrix[@intFromEnum(Super.InitialInstanceId.a)]
    .isSet(@intFromEnum(A.InitialInstanceId.a)));

// c_c is now related to super.a
const sub_c_assoc = r2.get_partition_assoc(.sub_c);
try testing.expect(@TypeOf(sub_c_assoc)
    .relation_matrix[@intFromEnum(C.InitialInstanceId.c)]
    .isSet(@intFromEnum(Super.InitialInstanceId.c)));
try testing.expect(@TypeOf(sub_c_assoc)
    .transpose_matrix[@intFromEnum(Super.InitialInstanceId.c)]
    .isSet(@intFromEnum(C.InitialInstanceId.c)));

// clean up the state of the generalization
r2.reclassify(.sub_c, c_c, .sub_a, a_a);
}

```

Applying the Generalization Relation

When computing the image of a generalization, we must specify the partition to which a domain element belongs. In technical words, the image (like the image of an association) is the result of applying the adjacency matrix for a domain, along a given partition, to obtain a codomain element. In modeling words, we traverse from a specific subclass to the superclass.

```

<<generalization matrix public functions>>=
/// The `image` function computes the set of codomain elements related to
/// `domain_set`. Note the cardinality of the resulting codomain set
/// is necessarily, by the properties of a generalization, the same as
/// the cardinality of `domain_set`.
pub fn image(
    : Self,
    comptime partition_id: PartitionId,
    domain_set: DomainTupleRefSet(partition_id),
) CodomainTupleRefSet {
    const partition_adj_matrix = get_partition_adj_matrix(partition_id);
    return partition_adj_matrix.image(domain_set);
}

```

When computing the preimage of a generalization, we must specify the partition with which we are concerned. In technical words, the preimage is the result of applying the adjacency matrix for a codomain along a given partition. In modeling words, the preimage is the result of the traversal of the generalization from the superclass to a specific subclass (as given by its partition identifier).

```

<<generalization matrix public functions>>=
/// The `preimage` function computes the set of domain elements contained in
/// the partition given by the `partition_id` argument
/// that are associated with the elements in `codomain_set`.
pub fn preimage(
    : Self,
    comptime partition_id: PartitionId,
    codomain_set: CodomainTupleRefSet,
) DomainTupleRefSet(partition_id) {
    const partition_adj_matrix = get_partition_adj_matrix(partition_id);
    return partition_adj_matrix.preimage(codomain_set);
}

```

Tests

```

<<binary_relation module tests>>=
test "generalization image and preimage" {
    const a = classify_tests.sub_a;
    const r2 = classify_tests.r2;

    const a_domain_ref = a.selectInitialInstance(.a);
    var a_domain_set = a_domain_ref.tupleRefSetFromTupleRef();
    var a_image = r2.image(.sub_a, a_domain_set);
    try testing.expectEqual(@as(usize, 1), a_image.count());
    try testing.expectEqualSlices(u8, a_image.readAttribute(.name), a_domain_ref
        .readAttribute(.name));

    // Computing the preimage across domain partition, "sub_c", using a codomain
    // set that contains only elements related to partition, "sub_a", yields
    // the empty set.
    const c_preimage = r2.preimage(.sub_c, a_image);
    try testing.expectEqual(@as(usize, 0), c_preimage.count());

    const aa = a.selectInitialInstance(.aa);
    a_domain_set.setUnion(aa.tupleRefSetFromTupleRef());
    a_image = r2.image(.sub_a, a_domain_set);
    try testing.expectEqual(@as(usize, 2), a_image.count());
}

```

Generalization Referential Consistency

The rules for referential consistency were described [previously](#). The implementation is a direct encoding of those rules.

The implementation strategy is to iterate over all the partitions of the domain checking for consistency of each partition. While iterating, the image of each partition is accumulated to compute the complete codomain. Each partition's image is checked against the accumulated image to ensure it is disjoint. Finally, the accumulated image is compared against the universe of the codomain.

```

<<generalization matrix public functions>>=
/// The `isGeneralizationConsistent` function determines if the given
/// generalization relationship follows the consistency rules.
/// The `partition_universes` argument is a slice of `DomainTupleRefSet`
/// which provide the universe of each domain partition against which
/// consistency is evaluated.
pub fn isGeneralizationConsistent(
    self: Self,
) bool {
    var total_image = emptyCodomainTupleRefSet;

    const partition_ids = comptime enums.values(PartitionId);
    inline for (partition_ids) |partition_id| {
        const partition_assoc = self.get_partition_assoc(partition_id);
        if (!partition_assoc.isAssociationConsistent()) {
            return false; ①
        }

        const domain_universe = @TypeOf(partition_assoc).domain.selectAll();
        const partition_image = self.image(partition_id, domain_universe);
        if (total_image.intersectWith(partition_image).count() != 0) { ②
            return false;
        }
        total_image.setUnion(partition_image);
    }

    const codomain_universe = codomain.selectAll();
    return if(total_image.eql(codomain_universe)) true else false; ③
}

```

- ① Check that the domain partition is injective.
- ② Check that the domain partition images are disjoint.
- ③ Check that the union of the domain partition images covers the codomain.

Tests

```

<<binary_relation module tests>>=
test "generalization consistency" {
    const A = classify_tests.Sub_a;
    const r2 = classify_tests.r2;

    try testing.expect(r2.isGeneralizationConsistent());

    const a_aa = A.TupleRef.initFromInitialInstance(.aa);
    _ = r2.unrelateDomainTuple(.sub_a, a_aa);
    try testing.expect(!r2.isGeneralizationConsistent());
}

```

Debugging Support

We provide a way to print a simple character based rendition of the partition generalization matrices. The output is a sequence of adjacency matrices for each partition in the generalization. An “x” character marks locations in the bit-set that are not in the domain or codomain sets and a “.” character for those that are included. If two elements are associated a “1” is printed in the corresponding cell, otherwise a “-” is printed. Rows and columns are labeled with the value of the domain or codomain element index modulo 10. The type name of the matrix is printed and an optional title are printed at the end. The following is a example output.

```
=====
 0 1 2 3 4
 . . x x x
0 . 1 - - - -
1 x - - - - -
2 x - - - - -
3 x - - - - -
4 x - - - - -
=====
partition 0
rship_adj_matrix.AssociationAdjMatrix(5,.@"?".@"1",5)

=====
 0 1 2 3 4
 . . x x x
0 . - 1 - - -
1 x - - - - -
2 x - - - - -
3 x - - - - -
4 x - - - - -
=====
partition 1
rship_adj_matrix.AssociationAdjMatrix(5,.@"?".@"1",5)
```

```
<<generalization matrix public functions>>=
/// The `printGenMatrices` function outputs an ASCII representation of
/// the adjacency matrices that represent the given generalization.
pub fn format(
    _: Self,
    writer: *std.Io.Writer,
) std.Io.Writer.Error!void {
    const partition_ids = comptime enums.values(PartitionId);
    inline for (partition_ids) |partition_id| {
        const partition_assoc = get_partition_adj_matrix(partition_id);
        const partition_name = @tagName(partition_id);
        try writer.print("{any}\npartition: '{s}'\n", .{ partition_assoc, partition_name });
    }
}
```

Tests

```
<<binary_relation module tests>>=
test "generalization printing" {
    std.log.debug("{any}\n", .{classify_tests.r2});
}
```

The next set of tests use the multiple generalization case presented [previously](#). Note the tests just manipulate the domain partitions and codomain elements, *i.e.* there is no application data attached to any of these manipulations.

```
<<binary_relation module tests>>=
const multi_gen_test = struct {
    const TailNumber = []const u8;
    const TakeOff = enum { tilt_wing, tilt_rotor, thrust };

    const RotaryWingAircraft = RelVar(
        struct { id: TailNumber, make: []const u8, model: []const u8 },
        &.{&.{.id}},
        10,
        null,
        &.{.
            {.heli_1, {.id = "N001", .make = "Bell", .model = "429" } },
            {.vtol_1, {.id = "N432", .make = "Airbus", .model = "Vahana" } },
        },
    );
    const rotary_wing_aircraft: RotaryWingAircraft = .{};

    const FixedWingAircraft = RelVar(
        struct { id: TailNumber, make: []const u8, model: []const u8 },
        &.{&.{.id}},
        5,
        null,
        &.{.
            {.vtol_1, {.id = "N432", .make = "Airbus", .model = "Vahana" } },
            {.air_1, {.id = "N501", .make = "Cessna", .model = "525" } },
        },
    );
    const fixed_wing_aircraft: FixedWingAircraft = .{};

    const Helicopter = RelVar(
        struct { id: TailNumber, rotor_count: u8 },
        &.{&.{.id}},
        10,
        null,
        &.{.
            {.heli_1, {.id = "N001", .rotor_count = 2 } },
        },
    );
}
```

```

);
const helicopter: Helicopter = .{};

const VtolAircraft = RelVar(
    struct { id: TailNumber, type: TakeOff },
    &.{&.{.id}},
    10,
    null,
    &.{.
        .{ .vtol_1, .{ .id = "N432", .type = .tilt_wing } },
    },
);
const vtol_aircraft: VtolAircraft = .{};

const Airplane = RelVar(
    struct { id: TailNumber, wing_count: u8 },
    &.{&.{.id}},
    5,
    null,
    &.{.
        .{ .air_1, .{ .id = "N501", .wing_count = 1 } },
    },
);
const airplane: Airplane = .{};

const R3 = GeneralizationAdjMatrix(
    RotaryWingAircraft,
    &.{.
        .{ .helicopter, Helicopter },
        .{ .vtol_aircraft, VtolAircraft },
    },
    &.{.
        .{ .helicopter, .heli_1, .heli_1 },
        .{ .vtol_aircraft, .vtol_1, .vtol_1 },
    },
);
const r3: R3 = .{};

const R4 = GeneralizationAdjMatrix(
    FixedWingAircraft,
    &.{.
        .{ .airplane, Airplane },
        .{ .vtol_aircraft, VtolAircraft },
    },
    &.{.
        .{ .airplane, .air_1, .air_1 },
        .{ .vtol_aircraft, .vtol_1, .vtol_1 },
    },
);
const r4: R4 = .{};

fn init() void {

```

```

        _ = RotaryWingAircraft.init();
        _ = FixedWingAircraft.init();
        _ = Helicopter.init();
        _ = VtolAircraft.init();
        _ = Airplane.init();
    }
};


```

```

<<binary_relation module tests>>=
test "multiple generalization tests" {
    multi_gen_test.init();
    const r3 = multi_gen_test.r3;
    const r4 = multi_gen_test.r4;
    const RotaryWingAircraft = multi_gen_test.RotaryWingAircraft;
    const rotary_wing_aircraft = multi_gen_test.rotary_wing_aircraft;
    const FixedWingAircraft = multi_gen_test.FixedWingAircraft;
    const fixed_wing_aircraft = multi_gen_test.fixed_wing_aircraft;
    const VtolAircraft = multi_gen_test.VtolAircraft;

    try testing.expect(r3.isGeneralizationConsistent());
    try testing.expect(r4.isGeneralizationConsistent());

    // Traverse R3 from Rotary Winged Aircraft N001 to Helicopter
    // to find all the helicopters (one in this case).
    const rwa_heli = try rotary_wing_aircraft.selectByIdentifier(0, .{ .id = "N001" });
    const helicopters = r3.preimage(.helicopter, rwa_heli.tupleRefSetFromTupleRef());
    try testing.expectEqual(@as(usize, 1), helicopters.count());

    // Traverse R4 from Fixed Winged Aircraft N501 to Airplane
    // to find all the airplanes (one in this case).
    const fwa_air = try fixed_wing_aircraft.selectByIdentifier(0, .{ .id = "N501" });
    const airplanes = r4.preimage(.airplane, fwa_air.tupleRefSetFromTupleRef());
    try testing.expectEqual(@as(usize, 1), airplanes.count());

    // Create a new VTOL aircraft and relate it to both fixed and rotary wing aircraft.
    const fwa_vtol = try FixedWingAircraft.TupleRef.createOrError(.{
        .id = "N465",
        .make = "Airbus",
        .model = "Vahana",
    });
    const rwa_vtol = try RotaryWingAircraft.TupleRef.createOrError(.{
        .id = "N465",
        .make = "Airbus",
        .model = "Vahana",
    });
    const vtol = try VtolAircraft.TupleRef.createOrError(.{
        .id = "N465",
        .type = .tilt_wing,
    });

    r3.relate(.vtol_aircraft, vtol, rwa_vtol);
    r4.relate(.vtol_aircraft, vtol, fwa_vtol);
}

```

```

try testing.expect(r3.isGeneralizationConsistent());
try testing.expect(r4.isGeneralizationConsistent());

// Traverse R3 from rotary winged aircraft to find all the related vtol aircraft.
// Then traverse R4 to from vtol aircraft to find all the fixed wing counterparts.
// This is just the preimage over R3 followed by the image over R4.
const all_rwa_vtol = rotary_wing_aircraft.selectAll();
const all_vtol = r3.preimage(.vtol_aircraft, all_rwa_vtol);
const vtol_as_fwa = r4.image(.vtol_aircraft, all_vtol);
try testing.expectEqual(@as(usize, 2), all_vtol.count());
try testing.expectEqual(@as(usize, 2), vtol_as_fwa.count());
}

```

Code Layout

```

<<binary_relation.zig>>=
//! The `binary_relation.zig` module implements the concepts of associations
//! and generalizations for relation variables. Associations are subsets of the
//! extended Cartesian product and generalizations based on the ideas of disjoint
//! sets. This module contains an implementation of binary relations that uses
//! an adjacency matrix to store the ordered pairs of a binary relation.
<<edit warning>>
<<copyright info>>

const std = @import("std");
const builtin = @import("builtin");
const testing = if (builtin.os.tag == .freestanding)
    @import("resee_testing")
else
    std.testing;
const fmt = std(fmt);
const enums = std.enums;
const meta = std.meta;

const assert = std.debug.assert;
const panic = std.debug.panic;

pub const version = "1.0.0-a1";

<<binary_relation module declarations>>

<<binary_relation module public functions>>

<<binary_relation module private functions>>

<<binary_relation module tests>>

```

[1] Not to be confused with a Zig tuple.

[2] Even if, for example floating point values, equality is within some epsilon margin.

[3] In the past, the Zig standard library had a `traits` module that contained this function. That module was removed. The implementation here was brought forward from the original in the Zig standard library

[4] This is sometimes called, the “allbut” operation

- [5] With the usual harangue that because tuple attributes have no order, a tabular display is not unique
- [6] A gronkolator is a meta-syntactic name for some entity that interacts with a program.
- [7] https://en.wikipedia.org/wiki/Transitive_closure
- [8] The expression results may be used to cause a side effect in the environment, but we consider that prospect later.
- [9] We specifically avoid terms like "key", ``primary key'', or "candidate key". These terms are used in RDMS. We also do not use the term "foreign key", preferring *referential attribute*, instead.
- [10] Identifiers whose attributes form a subset are not minimal.
- [11] We don't consider *n*-ary relationships.
- [12] As before, we take a casual attitude toward UML graphical notation.
- [13] These simple ideas can be easily conflated with SQL terminology of so called "foreign keys". We use the terms referential and identifying to describe the formalism of a relationship.
- [14] With the obligatory remark that relation values have *no* inherent order of either tuples or attributes. Despite what a tabular representation shows, there are no operations to *index* into a relation value or its heading.
- [15] You know when you have "touched ground" in an abstraction when the rules are not affected by problem specifics.
- [16] Note the encoding uses true for those instances which are part of the subset under consideration. The subscripts are used to identify the instances in the set.
- [17] Thanks, Paul, for prompting attention to this.
- [18] assuming you can find some common agreement among the various definitions of object orientation.

Execution Fundamentals

In the [chapter](#) discussing the execution model, state models were used to express the sequence of computations that were required to be performed in reaction to button pushes which controlled LED illumination. We were not specific about the rules of the state model and, for such a simple example, there was little chance of confusion. Now, we must get specific. If we are to enable executing the model in our heads, there must be precise rules for the logic of that execution. There are many variations of state models in use in the software world. We don't enumerate them here. Rather, we simply posit the particular variation of state model used here.

State Model Definitions

1. Execution sequencing and synchronization is specified by a [Moore type state model](#).
2. A Moore state model consists of a set of states, a set of events, a transition function mapping the current state and an event onto a new target state, and an optional *activity function* ^[1] associated with each state in the state model. The activity function corresponding to a state is executed each time a received event causes a *transition* into the state.
3. The activity function of a state may perform zero or more actions. Actions represent arbitrary computation, but there are four categories of actions that change the state of the domain or of the system as a whole:
 - Update the domain data model. This includes creating class instances, deleting class instances, updating class attributes or modifying association or generalization relationships between class instances.
 - Signal an event as an interaction with a different state machine.
 - Signal an event to cause the state machine to continue executing in another state.
 - Produce an external interaction for the domain. Domains produce external interactions with other domains and with the system environment.

The basic Moore state model is extended in three ways.

1. Not all events cause a transition. There are two non-transitioning target states, which we choose to call *ignore* and *never*. Any event which causes a transition into the target state, **ignore**, is discarded, no transition happens, and no activity function is invoked. Ignored events are a shorthand to reduce the number of states in a state diagram and are useful for situations where the event simply arrives at the wrong time. Any event which causes a transition into the target state, **never**, is forbidden and creates a panic condition. Never transitions indicate an analysis error and the implementation has no means of recovery. In this context, never means **absolutely never** and not *shouldn't happen* or *doesn't happen very often*. Transitions to **never** are undefined in much the same sense that division by zero is undefined.
2. An event may carry an *event parameter payload*. The event parameters are delivered as arguments to the activity function of the target state. A corollary to this rule is that the *event signature* of an event, i.e. the number, order, and types of the values of the event parameter payload, must match the *activity function signature* of the target state that receives the event.
3. The actions of a state activity can access the data model of the domain. The typical mathematical definition of a state machine uses the terms *input alphabet* to describe what drives the state transitions and *output alphabet* to describe any output of the transition. To enable an activity function of a state model to have access to the domain data model, a state model can only be used to describe the *life cycle* of a Relational Class or the synchronization of an Association. We describe Relation Classes and Associations in a later chapter. For now, you can consider a Relational Class as an enhanced Relation Variable and an Association as an

enhanced adjacency matrix. Both of these concepts were described in the last chapter. + By binding a state model to a Relational Class or Association, the activities of the model have access to the domain model data. Each class with a non-trivial life cycle or each association that synchronizes competitive resource usage must have a **state model** to specify its behavior. We'll refer to these classes as *active classes* and to these associations as *assigners*. All instances of an active class have the same behavior. Each instance of an active class has its own distinct notion of its current state. When referring to the state behavior of class instances, we use the term **state machine**.

Additional rules are:

1. Events are never lost.
2. Events signaled between any sending / receiving pair of instances are received in the order that they are signaled. Otherwise, there are no guarantees for the order of reception when multiple senders send events to the same receiving instance.
3. The state activity invoked upon a transition runs to completion before it considers any other events. State actions are not ``interrupted'' when an event is signaled.
4. When an active class instance is synchronously created, its state machine is placed in some particular state. It is allowed, although uncommon, for a state machine to be created in any state of the model. Creating an active class instance in some particular state *does not* cause the activity function of the state to be executed. For synchronously created instances, the initial state can be considered as just another data value that must be properly initialized.
5. An active class instance may be asynchronously created by signaling a *creation event*. When a creation event is dispatched, the instance is created in the *pseudo-initial state* and the event is dispatched to the newly created instance as part of a single operation.
6. When an event is received by a state machine, it computes the target state of the transition by computing the transition function of the state model. The transition function is computed from the machine's current state and the received event and returns the new target state. The machine's current state is set to the new state and the activity function associated with the new state is executed. This happens even if the target state is the same as the previous current state, *i.e.* transitioning a state machine back to the same state causes the activity function to be executed again.
7. An active class instance may be deleted asynchronous by designating at least one state as *terminal*. Upon transitioning to a terminal state and executing the activity function of the state, the dispatch mechanism then deletes the instance. Asynchronous deletion can happen even if the class instance was not created asynchronously and asynchronously created instance do not necessarily have to be deleted asynchronously.
8. An active class instance may signal an event to itself. These events are called, *continuation events*, and are a means to have transitory states where the activity of a given state is continued in another state. The dispatch mechanism insures that continuation events are received before any events from another state machine. The invocation of a state machine activity function may signal at most one continuation event. Continuation events may be signaled conditionally depending upon the conditional logic of the activity function. Thus, states may behave as transitory states either conditionally or unconditionally.
9. A state machine may signal a *delayed event*. Delayed events are interpreted as a request to the dispatch mechanism to signal the event at some time in the future. There may be at most one pending delayed event of a given type between any sending / receiving pair (even if they are the same instance). This design treats attempts to send duplicate delayed events as a request to cancel the original event and request sending the event at the newly requested delay interval^{footnote::[The other interpretation of this situation is to treat it as a panic condition and force the activity function to cancel any delayed event before sending a new one at a new time. This interpretation is deemed more in line with common use cases.]}. A delayed event may be cancelled and the amount of time remaining before it is dispatched can be queried. A delayed

event may be signaled with a zero delay time. This design treats delayed events with a zero time as a request to the dispatch mechanism to signal the event immediately. Since delayed events are signaled by the dispatch mechanism based on the passing of time in the environment, they are not considered a continuation event despite the possibility of both the source and target of the event being the same instance.

10. An association may also exhibit stateful behavior and have a state model to describe its behavior. These are called, *assigner state models*, as they are used for associations that are competitive in nature over some resource which must be assigned in a single action. Assigner state models are discussed in a later section.
11. A class that participates as the superclass (codomain) of a Generalization relationship and whose subclasses are active classes may designate a subset of the state machine events as *polymorphic*. When a polymorphic event is targeted at a superclass instance, the dispatch mechanism determines the subclass instance to which the superclass is currently related and delivers the event to that subclass. Polymorphic events are typically used to express modal behavior and are discussed in a later section.

Representing State Models

From the above rules, it is clear that specifying a state model requires substantial information. The required information is accumulated in the following specification structure.

```
<<active class module declarations>>=
/// The `StateModelSpec` structure specifies the properties of a state model
/// that can be attached to a Relational Class.
pub const StateModelSpec = struct {
    /// The `state_names` field is a slice of strings that give the names
    /// of all the regular states in the state model. The pseudo-initial state
    /// must not be included.
    state_names: []const [:0]const u8,
    /// The `event_names` field is a slice of strings that give the names
    /// of the events in the state model.
    event_names: []const [:0]const u8,
    /// The `transitions` field is a slice of transition specifications.
    transitions: []const TransitionSpec,
    /// The `default_initial_state` is the name of the state into which
    /// the state machine is placed when a class instance is created
    /// synchronously and no other initial state is specified.
    default_initial_state: [:0]const u8,
    /// The `StateActivities` field is a namespace struct that contains the action
    /// functions of the state model. All states that have a non-empty
    /// activity function must have a function declaration by the same name
    /// in the `StateActivities` namespace. If no activity function is found
    /// for a state, then the state activity is assumed to be empty and
    /// no computation is done upon transitioning into such a state. The
    /// `StateActivities` namespace may only contain function declarations.
    /// For functions intended to serve as activity functions, the
    /// first argument must be an `InstanceRef` type of a Relational
    /// Class. Other functions may appear in
    /// `StateActivities` and are assumed to be common code shared among
    /// the activity functions. Functions in `StateActivities` intended to be
    /// activity functions must be declared as `pub`. Any other function
```

```

    /// in `StateActivities` should be not be declared as `pub` and are
    /// intended as private, shared code for the activity functions.
    StateActivities: type,
    /// The `unspecified_transition` -- is a string value that gives the name
    /// of one of the two non-transitioning states. This state is used
    /// for any transition not found in the `transitions` argument.
    /// The entire transition function (as a `states` by `events`
    /// matrix) must be populated. The `transitions` argument allows
    /// for a minimal specification of the transitions and any missing
    /// transitions are supplied as the `unspecified_transition` value.
    /// In almost all cases, this should be specified as "never" and
    /// those events which are ignored are specified directly in the
    /// `transitions` arguments. However, there are cases where "ignore"
    /// is a better choice for default and this argument provides that
    /// flexibility.
    unspecified_transition: [:0]const u8, // "ignore" or "never"
    /// The `terminal_states` field is a slice of state names that are
    /// considered terminal states. Any class instance that
    /// transitions to a terminal state is deleted after executing the
    /// activity function of the state. A `terminal_states` value which is
    /// an empty slice indicates that the model has no terminal states.
    terminal_states: []const [:0]const u8,
};


```

Since a state model is a directed graph, the specification of transitions describes the arcs of the graph.

```

<<active class module declarations>>=
/// The `TransitionSpec` structure specifies the state model transitions
/// and consists of a source state name and a slice of outbound
/// transitions. The source state name must be one of the regular
/// state names of the model or it may be the pseudo-initial state name,
/// "@". The elements of the outbound transitions slice are the event
/// name which causes a transition and the target state name that is the
/// destination of the transition. A target state name may be an regular
/// state name or one of the non-transitioning state names, "ignore" or
/// "never". A target state name may not be the pseudo-initial state,
/// "@", since it has no inbound transitions by definition. The fields
/// of `TransitionSpec` specifies an adjacency list representation for
/// the implied directed graph of the state model.
pub const TransitionSpec = struct {
    [:0]const u8, // source state name -- either one of state_names or "@"
    []const struct { // outbound transitions
        [:0]const u8, // event name
        [:0]const u8, // target state name,
                      // either one of state_names or "ignore" or "never"
    },
};


```

A state model is created by the `StateModel` type function using the specification data just discussed.

```

<<active class module public functions>>=
/// The `StateModel` function is a type function which returns a data type
/// that is bound to the relational class given by the, `RelClassType`,
/// argument and has the state model properties given by the, `model_spec`,
/// argument.
pub fn StateModel(
    comptime RelClassType: type,
    comptime model_spec: *const StateModelSpec,
) type {
    return struct {
        const Self = @This();

        <<state model variables>>
        <<state model declarations>>
        <<state model public functions>>
        <<state model functions>>
    };
}

```

Representing States

States in state models are given regular string names. We use state names directly in the meta-programming so all state names must also be valid Zig identifiers.

It is also useful to categorize the roles played by states:

1. A *regular* state is simply a state with no additional specific properties. Most states are *regular*. All of the states given in the `state_names` field are *regular*.
2. A *final* state has no outbound transitions. With no outbound transitions, the state machine will remain in that state until it is deleted or until the program terminates. There is nothing inherently wrong with a *final* state and they are sometimes used as a place the instance can land before it is synchronously deleted. A *final* state is not present as a source state in the transition specification for the state model.
3. A *terminal* state is a *final* state where the class instance is deleted by the dispatch mechanism after its activity function (if any) is executed. Since a *terminal* state is also a *final* state, it can have no outbound transitions. It is meaningless to specify outbound transitions on a state where the instance is deleted and can therefore never receive another event.
4. The *default_initial* state is the state into which an instance is placed when it is synchronously created and no other initial state was specified. Every state model must specify exactly one *regular* state as the *default_initial* state.
5. A *pseudo-initial* state is a special transitory state into which an instance is placed when it is created asynchronously. There can be at most one *pseudo-initial* state in a state model. We use the at sign, `@`, to indicate the *pseudo-initial* state. A *pseudo-initial* state is added to the state model whenever an outbound transition from the `@` state is specified in the transition specification. Note that the *pseudo-initial* state has no inbound transitions and no action function. The *pseudo-initial* state may only appear as the source state of a transition and need only be specified for those state models where the associated Relational Class is asynchronously created. The rules for the *pseudo-initial* state are an adaptation to the specific semantics of asynchronous creation. Note that when an instance is asynchronously created, the creation of the instance and the dispatch of an event happen as a single operation.

```
<<active class module declarations>>=
pub const pseudo_initial_state_name = "@";
```

Every state model is augmented with the two non-transitioning states, `ignore` and `never`. These non-transitioning states may only be used as the target of a transition.

```
<<state model functions>>=
fn isNonTransitionTarget(
    comptime name: [:0]const u8,
) bool {
    return if (ascii.eqlIgnoreCase(name, "ignore") or ascii.eqlIgnoreCase(name, "never"))
        true
    else
        false;
}
```

To accommodate the non-transitioning states, we encode -1 for ignore and -2 for never. All other target states are encoded as non-negative integers.

```
<<active class module declarations>>=
const ignore_state_value = -1;
const never_state_value = -2;
```

Source and Target States

The transition function for a state model is encoded as a 2-D matrix. There are *source state count* number of rows and *event count* number of columns. Each cell contains a *target state*.

To clarify, a source state is any state mentioned in the `state_names` argument plus, optionally, the pseudo-initial state, `@`. A target state is any state mentioned in the `state_names` argument plus the two non-transitioning states, `ignore` and `never`. The intersection of the sets of source and target state is simply the states given in the `state_names` argument. This nuance will become important later and is used to compute the transition function.

State Machine Current States

Each instance of a state model stores its own value of its current states.

```
<<state model variables>>=
var current_states: [RelClassType.capacity]StateId = undefined;
```

Encoding States

As was done in many previous cases, we use Zig's ability to map strings to enumerations at compile time to create enumerations for state names that are used in throughout the code in preference to handling strings.

```
<<state model declarations>>=
pub const StateId = TagSourceStates();
```

```

<<state model functions>>=
/// The `TagSourceStates` type function creates an enumeration type
/// based on the `state_names` argument. One enum tag is created for
/// each name in `state_names`. If the state model includes outbound
/// transitions from the pseudo-initial state, then it is also included
/// in the generated enumeration type.
fn TagSourceStates() type {
    var fields: []const EnumField = &[_]EnumField{};

    inline for (model_spec.state_names, 0..) |state_name, state_index| {
        // The "ignore" and "never" names for the non-transitioning states are reserved.
        if (isNonTransitionTarget(state_name)) {
            @compileError("state, '" ++ state_name ++
                "', is not an allowed state name for the source of a transition");
        } else if (mem.eql(u8, state_name, pseudo_initial_state_name)) {
            @compileError(
                "the pseudo-initial state may not appear in the set of model source
states");
        } else {
            fields = fields ++
                &[_]EnumField{.name = state_name, .value = state_index};
        }
    }

    if (has_pseudo_initial) { ①
        fields = fields ++
            &[_]EnumField{.name = "@", .value = model_spec.state_names.len };
    }

    return @Type(.{
        .@enum" = .{
            .tag_type = u8, ②
            .fields = fields,
            .decls = &.{},
            .is_exhaustive = true,
        },
    });
}

```

- ① The pseudo-initial state is simply appended to the end of the enumeration fields, if necessary.
- ② The type of the enumeration tag is fixed at `u8` for convenience of alternating between the enumeration tag and its integer representation.

To create an enumeration type for target states, we must add the "ignore" and "never" non-transitioning states to the `state_names` specified for the model. Note that the result *does not* include the pseudo-initial state, as desired.

```

<<state model declarations>>=
pub const TargetStateId = TagTargetStates();

```

```
<<state model functions>>=
```

```

/// The `TagTargetStates` function returns an enumeration data type
/// that encodes target states. This includes all the regular states
/// and the two non-transitioning states, "ignore" and "never".
fn TagTargetStates() type {
    var fields: []const EnumField = &[_]EnumField{};
    inline for (model_spec.state_names, 0..) |state_name, state_index| {
        fields = fields ++ &[_]EnumField{.name = state_name, .value = state_index };
    }
    fields = fields ++ &[_]EnumField{
        .{ .name = "ignore", .value = ignore_state_value },
        .{ .name = "never", .value = never_state_value },
    };
}

return @Type(.{
    .@enum" = .{
        .tag_type = i8,
        .fields = fields,
        .decls = &.{},
        .is_exhaustive = true,
    },
});
}

```

Because of the careful encoding of the enumerations that represent source and target state, the conversion between them is essentially a no-op. The conversion is necessary after computing the transition function since the current state of a machine does not include non-transitioning states.

```

<<state model functions>>=
/// Returns a source state id from a target state id.
fn stateFromTargetState(
    target_id: TargetStateId,
) StateId {
    return @enumFromInt(@intFromEnum(target_id));
}

```

It is convenient for the dispatch mechanism to have a way to determine if a given state is a terminal state.

```

<<state model declarations>>=
/// The set of states that are terminal states.
const terminals = terminalStatesSet();

```

```

<<state model functions>>=
/// The `terminalStatesSet` returns an `EnumSet` value that contains all the
/// states that are designated as terminal.
fn terminalStatesSet() enums.EnumSet(StateId) {
    var term_state_set: enums.EnumSet(StateId) = .initEmpty();

    inline for (model_spec.terminal_states) |term_state_name| {
        if (isNonTransitionTarget(term_state_name)) {

```

```

    @compileError(
        "non-transitioning states cannot be terminal: found '" ++
        term_state_name ++
        "'"
    );
} else if (mem.eql(u8, term_state_name, pseudo_initial_state_name)) {
    @compileError("the pseudo-initial state cannot be terminal");
} else if (hasOutboundTransitions(@field(StateId, term_state_name))) {
    @compileError(
        "terminal state '" ++
        term_state_name ++
        "' may not have outbound transitions"
    );
} else {
    term_state_set.insert(@field(StateId, term_state_name));
}
}
return term_state_set;
}

```

The set of terminal states is used by the dispatch mechanism to determine if a transition has caused a state machine to reach a terminal state. In that case, it must be reported so the class instance may be deleted.

```

<<state model functions>>=
/// Returns whether the given `state_id` is contained in the set of terminal state
/// for the model.
fn isTerminalState(
    state_id: StateId,
) bool {
    return terminals.contains(state_id);
}

```

```

<<state model functions>>=
/// Returns the non-transitioning state id that is designated as the as the
/// default transition for the model.
fn validateDefaultTransition() TargetStateId {
    if (!(mem.eql(u8, model_spec.unspecified_transition, "ignore") or
          mem.eql(u8, model_spec.unspecified_transition, "never")))
    {
        @compileError("the default transition must be either 'ignore' or 'never'");
    }
    return @field(TargetStateId, model_spec.unspecified_transition);
}

```

```

<<state model declarations>>=
const default_transition = validateDefaultTransition();

```

Similarly, it is useful to have the default initial state handy. Note here we validate that "ignore" or "never" do not appear as the default initial state. If the name given for the default initial state is not

in the regular states, then obtaining its value via `@field()` will fail.

```
<<state model functions>>
/// Returns a `StateId` value into which an instance is placed upon
/// creation if no other state was specified.
fn validateDefaultInitialState() StateId {
    if (mem.eql(u8, model_spec.default_initial_state, pseudo_initial_state_name)) {
        @compileError("the pseudo-initial state may not be specified as the default initial
state");
    }
    return @field(StateId, model_spec.default_initial_state);
}
```

```
<<state model declarations>>
const initial_state = validateDefaultInitialState();
```

Representing Events

The definition of a state model includes a set of events. As with states, the enumerated form is used in the interface for functions. For events, there are no special rules. The slice of `event_names` is simply turned into an enumerated type.

```
<<state model functions>>
fn TagEvents() type {
    var fields: []const EnumField = &[_]EnumField{};
    inline for (model_spec.event_names, 0..) |event_name, event_index| {
        fields = fields ++ &[_]EnumField{.name = event_name, .value = event_index};
    }
    return @Type(.{
        .@enum" = .{
            .tag_type = u8,
            .fields = fields,
            .decls = &.{},
            .is_exhaustive = true,
        },
    });
}
```

```
<<state model declarations>>
pub const EventId = TagEvents();
```

A state machine signaling an event to another state machine is the primary means by which execution is sequenced. When the signaled event is received by the target state machine, any transition the event causes results, generally, in the execution of an activity function of the state model. These types of events are called, *interactive events*. The data type used for signaling is specific to the state model receiving the event because of the variations in the event encoding and parameter payload.

The Event type contains three essential pieces of information:

1. a means to identify the Relational Class instance to which the event is directed,
2. a means to identify of the event identifier, and
3. the event payload values that are passed to the activity function when the event is received.

This information is encoded in a single tagged union data structure.

1. The union is tagged with `EventId`.
2. The data is a union of the tuples that are the event payload signature.
3. The first argument to all state activity functions is an `InstanceRef` value that refers to the class instance receiving the event.

We build the `Event` type for each active class and it is specific to the characteristics of the class state model.

```
<<state model declarations>>=
pub const Event = MakeEvent();
```

```
<<state model functions>>=
/// The `MakeEvent` function constructs a type which is suitable for use
/// to signal events to instances of the given state model.
fn MakeEvent() type {
    var arg_tuple_fields: []const UnionField = &[_]UnionField{};

    inline for (enums.values(EventId)) |event_id| {
        const inbound_states = inboundTransitions.getPtrConst(event_id);
        const inbound_count = inbound_states.count();

        if (inbound_count == 0) {
            // If for a given event, we can find no regular (i.e. transitioning)
            // state to which it causes an inbound transition, then either the
            // event is unused or is used only to target non-transitioning states
            // (i.e. ignore and never). This is a specification error.
            @compileError("event '" ++ @tagName(event_id) ++
                "' causes no transitions to regular states: the event is unproductive");
        } else {
            var iter = inbound_states.iterator();
            const inbound_state = iter.next().?; // inbound_count is non-zero
            const inbound_action =
                @field(activity_map.get(inbound_state), @tagName(inbound_state));
            const InboundActionType = @TypeOf(inbound_action);
            const InboundActionArgsType = if (InboundActionType != void) iblk: {
                // In this case, inbound action type is a pointer to a function.
                // Fish out the function type itself to feed to ArgsTuple
                const InboundFunctionType = @typeInfo(InboundActionType).pointer.child;
                break :iblk meta.ArgsTuple(InboundFunctionType);
            } else eblk: {
                // Void action types arise when no activity function is specified
                // for a state in the transitions argument. Even if there
                // is no action for a state, we require the event to carry
                // the instance reference, i.e. an InstanceRef of the class.
        }
    }
}
```

```

// In this case, the activity function args are a 1-tuple with
// an InstanceRef field.
break :eblk @Type(.{
    .@"struct" = .{
        .layout = .auto,
        .backing_integer = null,
        .fields = &[_]StructField{
            .{
                .name = "0",
                .type = InstanceRef,
                .default_value_ptr = null,
                .is_comptime = false,
                .alignment = if (@sizeOf(InstanceRef) == 0) 1 else
@alignOf(InstanceRef),
            },
            .,
        },
        .decls = &.{},
        .is_tuple = true,
    },
});
};

arg_tuple_fields = arg_tuple_fields ++
&[_]UnionField{.{
    .name = @tagName(event_id),
    .type = InboundActionArgsType,
    .alignment = if (@sizeOf(InboundActionArgsType) == 0) 1 else @alignOf
(InboundActionArgsType),
}};

// Continue examining the other states that have an inbound transition caused
// by this event to verify that the activity function arguments match.
inline while (iter.next()) |other_state| {
    const other_action =
        @field(activity_map.get(other_state), @tagName(other_state));
    const OtherActionType = @TypeOf(other_action);
    if (OtherActionType != void) {
        const OtherFunctionType = @typeInfo(OtherActionType).pointer.child;
        const OtherActionArgsType = meta.ArgsTuple(OtherFunctionType);

        if (OtherActionArgsType != InboundActionArgsType) {
            @compileError("event '" ++ @tagName(event_id) ++
                "' causes transitions to '" ++ @tagName(inbound_state) ++
                "' and '" ++ @tagName(other_state) ++
                "' which have different signatures");
        }
    }
}

// In the end, an Event type is just a tagged union of event payload tuples.

```

```

return @Type(.{
    .@"union" = .{
        .layout = .auto,
        .tag_type = EventId,
        .fields = arg_tuple_fields,
        .decls = &.{},
    },
});
}

```

Since `comptime` created types cannot have function declarations, we initialize an event using a function separate from the `Event` type.

```

<<instance reference declarations>>=
/// The `makeEvent` function returns an `Event` value which can be
/// signaled to `inst_ref` and carries event parameter payload given
/// by, `event_payload`. The `event_payload` parameter is a Zig tuple
/// containing those values for a state activity function as supplemental
/// parameters beyond the required first argument `InstanceRef`. The
/// order and types of the `event_payload` must match those of the
/// state activity function for which the `event_id` causes an inbound
/// transition.
pub fn makeEvent(
    inst_ref: InstanceRef,
    comptime event_id: EventId,
    event_payload: anytype, // tuple of additional activity arguments
) Event {
    // Build the event by combining the `inst_ref` with the `event_payload`
    // to yield the argument tuple for the activity function.
    const ActivityArgs = @FieldType(Event, @tagName(event_id));
    var activity_args: ActivityArgs = undefined;
    activity_args[0] = inst_ref;
    inline for (event_payload, 1..) |payload_arg, activity_index| {
        activity_args[activity_index] = payload_arg;
    }

    return @unionInit(Event, @tagName(event_id), activity_args);
}

```

Representing the Transition Function

The `transitions` field in the `StateModelSpec` is a slice of `TransitionSpec` values. This structure is essentially an adjacency list specifying, for each source state with outbound transitions, the set of event / target state pairs that cause the machine's transitions. The events can be viewed as labeled, directed arcs of a directed graph. The arc is directed from a source state to a target state.

The adjacency list specification is converted into a transition matrix, filling in each cell, and mapping states and events to target states. This makes the computation of the transition function simpler and the matrix can be generated at from the `TransitionSpec` values at `comptime`.

```

<<state model declarations>>=

```

```
const transition_map = makeTransitionMap();
```

Define the size of the transition matrix.

```
<<state model declarations>>=
const state_count = model_spec.state_names.len + @intFromBool(has_pseudo_initial);
const event_count = model_spec.event_names.len;
```

```
<<state model functions>>=
/// The `makeTransitionMap` function creates a Moore type state transition
/// matrix composed of _source state_ rows by _event_ columns. Each cell
/// in the matrix contains the _target state_ value which becomes the new
/// state of the machine when a given event is received by the machine
/// in a given state.
fn makeTransitionMap() [state_count][event_count]TargetStateId {
    // The strategy is to initialize the matrix to the default initial state
    // and then overwrite the cell values using the information in the transition
    // specification.
    var trans_matrix: [state_count][event_count]TargetStateId =
        @splat(@splat(default_transition));

    // Make sure any outbound transitions from the pseudo-initial
    // state that are not specified are marked as `never` even if the
    // default transition is `ignore`. Outbound transitions from the
    // pseudo-initial state cannot be ignored.
    if (has_pseudo_initial and default_transition == .ignore) {
        inline for (trans_matrix[@intFromEnum(StateId.@["@"])[0...]]) |*psis_trans| {
            psis_trans.* = .never;
        }
    }

    // Insist that transition specs have at most one mention of any given state,
    // i.e. the set of source states in the transition specification must be a set.
    var states_encountered: enums.EnumSet(StateId) = .{};

    // Since the transition specification is in the form of an adjacency list,
    // iterating over it gives the information for filling in the cell values.
    inline for (model_spec.transitions) |state_trans| {
        const state = @field(StateId, state_trans@"0");
        if (states_encountered.contains(state)) {
            @compileError(
                "duplicate source state specification for '" ++
                @tagName(state) ++
                "'");
        } else {
            states_encountered.insert(state);
        }

        // Insist that the transitions for a state mention a given event
        // at most one time, i.e. the outbound transitions for a state
        // also form a set.
        var events_encountered: enums.EnumSet(EventId) = .{};

    }
}
```

```

    const outbound = state_trans.@"1";
    if (outbound.len == 0)
        @compileError("there must be at least one outbound transition " ++
                      "in a transition specification");
    inline for (outbound) |trans| {
        const event = @field(EventId, trans.@"0");
        if (events_encountered.contains(event)) {
            @compileError(
                "duplicate event transition for '" ++
                @tagName(event) ++
                "'");
        } else {
            events_encountered.insert(event);
        }
        const target_state = @field(TargetStateId, trans.@"1");
        if (has_pseudo_initial and state == .@ "@" and target_state == .ignore) {
            @compileError(
                "transitions from the pseudo initial state cannot be ignored");
        }
        // Note we have chosen to index into the transition matrix using integer
        // values. Alternatively, you could consider a nested set of `EnumArray`-
        // structures. This was chosen for familiarity and readability.
        // Also note, the cell values are enumerated values of `TargetStateId`-
        // which include the "ignore" and "never" non-transitioning states.
        trans_matrix[@intFromEnum(state)][@intFromEnum(event)] = target_state;
    }
}

return trans_matrix;
}

```

Ultimately, the state model transition function reduces down to a single array indexing operation.

```

<<state model functions>>=
/// The `transition` function computes the new target state into which
/// a state machine is placed when `event` is received and the machine's
/// state is `current_state`.
fn transition(
    _: Self,
    current_state: StateId,
    event: EventId,
) TargetStateId {
    return transition_map[@intFromEnum(current_state)][@intFromEnum(event)];
}

```

There are a few other useful views of a `TransitionSpec` and we present some `comptime` functions that summarize information about the values in the `transitions` field.

Since the pseudo-initial state may only appear in the transitions, we provide a function to detect its presence. The check for whether the pseudo-initial state is used simply iterates across the transition specification to check if there is a source of a transition from the `@` state.

```

<<state model functions>>=
/// The `containsPseudoInitialState` function return true if the pseudo initial
/// state name, `@`, appears as a source state in the transitions.
fn containsPseudoInitialState() bool {
    inline for (model_spec.transitions) |trans_spec| {
        const source_state = trans_spec.@"0";
        if (mem.eql(u8, source_state, pseudo_initial_state_name)) return true;
    }
    return false;
}

```

Since the pseudo-initial state determination is used in several places, It is convenient to hold the result.

```

<<state model declarations>>=
pub const has_pseudo_initial = containsPseudoInitialState();

```

It is also necessary, in code presented later, to determine if a given event causes an inbound transition to a given state. This is equivalent to determining if an event is one of the labeled, directed arcs into a given state.

```

<<state model declarations>>=
// Scan an event adjacency list to discover the transitions inbound
// to the various states. This allows us to discover the event
// parameters that must be carried along with the event.
const inboundTransitions = makeInboundEventMap();

```

The design is the use an `EnumArray` that is indexed by an `EventId` value. The elements of the array are an `EnumSet` that contains a set of `StateId` values for which the event causes an inbound transition.

```

<<state model declarations>>=
const InboundStateSet = enums.EnumSet(StateId);
const InboundEventMap = enums.EnumArray(EventId, InboundStateSet);

```

```

<<state model functions>>=
/// The `makeInboundEventMap` function returns an `InboundEventMap` value whose
/// elements are `InboundStateSet` values. The array is indexed by `EventId`
/// values and the elements of the array form a set of those `StateId`
/// values where the corresponding event causes a transition _into_
/// the state.
fn makeInboundEventMap() InboundEventMap {
    // Scan the transition table for each event and find the inbound states
    // placing them in an EnumSet.
    var inboundMap: InboundEventMap = .initFill(.initEmpty());
    inline for (enums.values(EventId)) |event_id| {
        const inboundStates = inboundMap.getPtr(event_id);
        inline for (enums.values(StateId)) |state_id| {
            const target_state_id =

```

```

        transition_map[@intFromEnum(state_id)][@intFromEnum(event_id)];
    if (isTransitioningTarget(target_state_id)) {
        inboundStates.insert(stateFromTargetState(target_state_id));
    }
}
}

return inboundMap;
}

```

```

<<state model functions>>=
/// The `isTransitioningTarget` function returns `true` if `target_id` is
/// a `TargetStateId` value that _does not_ include the non-transitioning
/// state values and `false` otherwise.
fn isTransitioningTarget(
    target_id: TargetStateId,
) bool {
    return !(target_id == .ignore or target_id == .never);
}

```

```

<<state model functions>>=
/// The `isInboundEvent` function returns `true` if `state_id` is the target
/// of a transition causes by `event_id` and `false` otherwise.
fn isInboundEvent(
    state_id: StateId,
    event_id: EventId,
) bool {
    return inboundTransitions.getPtrConst(event_id).contains(state_id);
}

```

```

<<state model functions>>=
/// The `hasInboundTransitions` function returns `true` if `state_id` is the target
/// of a transition from another state.
fn hasInboundTransitions(
    state_id: StateId,
) bool {
    inline for (enums.values(EventId)) |event_id| {
        if (isInboundEvent(state_id, event_id)) return true;
    }

    return false;
}

```

We are also interested in determining if a given state has any outbound transitions. A state with no outbound transitions is a final state and any state marked as a terminal state must be a final state.

The transition function for the state model is computed by indexing into a two dimensional matrix, returning the value of the cell. The transition matrix contains one row for each source state and one column for each event. Each cell in the matrix holds the value of a target state to which the transition is directed.

```

<<state model declarations>>=
// Scan an event adjacency list to discover the transitions inbound
// to the various states. This allows us to discover the event
// parameters that must be carried along with the event.
const outboundTransitions = makeOutboundTransMap();

```

Following the example of the inbound transitions, we use an `EnumArray` that is indexed by a `StateId` value. The elements of the array are an `EnumSet` that contains the set of `EventId` values that cause outbound transitions for the state.

```

<<state model declarations>>=
const OutboundEventSet = enums.EnumSet(EventId);
const OutboundTransMap = enums.EnumArray(StateId, OutboundEventSet);

```

```

<<state model functions>>=
/// The `makeOutboundTransMap` function returns an `OutboundTransMap` value whose
/// elements are `OutboundEventSet` values. The array is indexed by `StateId`
/// values and the elements of the array form a set of those `EventId`
/// values where the corresponding event causes a transition _out of_
/// the state.
fn makeOutboundTransMap() OutboundTransMap {
    // Scan the transition table for each event and find the inbound states
    // placing them in an EnumSet.
    var outboundMap: OutboundTransMap = .initFill(.initEmpty());
    inline for (enums.values(StateId)) |state_id| {
        const outboundTrans = outboundMap.getPtr(state_id);
        inline for (enums.values(EventId)) |event_id| {
            const target_state_id =
                transition_map[@intFromEnum(state_id)][@intFromEnum(event_id)];
            if (isTransitioningTarget(target_state_id)) {
                outboundTrans.insert(event_id);
            }
        }
    }

    return outboundMap;
}

```

```

<<state model functions>>=
/// The `hasOutboundTransitions` function returns `true` if `state_id` is
/// has at least one outbound transition.
fn hasOutboundTransitions(
    state_id: StateId,
) bool {
    return outboundTransitions.getPtrConst(state_id).count() != 0;
}

```

Representing Activity Functions

The activity functions for a state model have some specific constraints on them. Before building the representation of those activity function, it is necessary to insure the specification of the activity functions is correct. A set of argument validation functions are presented to enable better error messages when specifying StateAction namespaces.

1. All state activity functions are defined in the `StateActivities` namespace.
2. The `StateActivities` namespace must be a `struct` and may not be a Zig tuple.
3. The `StateActivities` namespace `struct` may not contain any fields. All application data is stored in a Relational Class and may not be hidden in the `StateActivities`.
4. The activity function for a state has the same name as the state.
5. The activity function for a state must be declared as public.
6. The `StateActivities` namespace `struct` may contain non-public functions that has common code shared among the activity functions.
7. An activity function may not be generic or have variable arguments, and must have a return type of `void`.
8. All activity functions must have at least one argument, that being an `InstanceSet` of the Relational Class.

```
<<state model declarations>>=
const InstanceSet = RelClassType.InstanceState;
```

The rule that the state activity signature must match the event parameter payload is *not* enforced here. That validation is done later when we have more information about the state model's directed graph.

```
<<state model functions>>=
/// The `validateStateActivities` function examines the functions defined in
/// the `StateActivities` part of a state model specification to validate
/// they match rules for activity functions. Any non-conforming activity
/// functions result in a compile error.
fn validateStateActivities() void {
    const activities_info = @typeInfo(model_spec.StateActivities);
    if (activities_info != .@("struct"))
        @compileError("state activities must be contained in a structure");
    if (activities_info.@("struct").is_tuple) @compileError("state activities cannot be
tuples");
    if (activities_info.@("struct").fields.len != 0)
        @compileError("state activities namespace structures may not have fields");

    inline for (model_spec.state_names) |state_name| {
        if (comptime hasStateActivityFunction(state_name)) {
            const activity_fn_info =
                @typeInfo(@TypeOf(@field(model_spec.StateActivities, state_name))).@("fn");
            if (activity_fn_info.is_generic) @compileError(
                "a state activity function cannot be generic: '" ++
                state_name ++
                "'");
```

```

    if (activity_fn_info.is_var_args) @compileError(
        "a state activity cannot have variable arguments: '" ++
        state_name ++
        "'");
    if (activity_fn_info.return_type) |activity_return| {
        if (activity_return != void) @compileError(
            "a state activity must have a void return: '" ++
            state_name ++
            "' returns '" ++
            @typeName(activity_return) ++
            "'";
        );
    } else {
        @compileError(
            "a state activity must have a return type: '" ++
            state_name ++
            "'");
    }
}

if (!comptime hasInboundTransitions(@field(StateId, state_name)))
    @compileError(
        "states with activity functions must have inbound transitions");

const activity_params = activity_fn_info.params;
if (activity_params.len == 0) @compileError(
    "a state activity must have at least one parameter: '" ++
    state_name ++
    "'");
comptime validateFirstActivityParam(activity_params[0]);
}
}
}

```

There are several places where we need to know if an activity function was declared for a given state.

```

<<state model functions>>=
/// Return true if an activity function was declared for the given `state_name`.
fn hasStateActivityFunction(
    comptime state_name: [:0]const u8,
) bool {
    return meta.hasFn(model_spec.StateActivities, state_name);
}

```

```

<<state model functions>>=
/// The `validateFirstActivityParam` function ensures that the first
/// parameter of an activity function is an `InstanceSet` for the Relational
/// Class containing the state model.
fn validateFirstActivityParam(
    first_param: std.builtin.Type.Fn.Param,
) void {

```

```

if (first_param.is_generic) {
    @compileError("the first parameter of a state activity cannot be generic");
}
if (first_param.type) |param_type| {
    if (param_type != InstanceRef) {
        @compileError("the first parameter of a state activity must be an
InstanceRef");
    }
} else {
    @compileError("the first parameter of a state activity cannot be anytype");
}
}

```

A state machine transition is a two part process. First, as we have seen in the last section, the transition function is executed. It maps the current state and the received event to a new target state. Assuming the new state is a transitioning state, then the activity function associated with the state is executed.

In this section we describe how the activity functions are associated to their corresponding states and how they are invoked upon a state machine transition.

The simplest means of associating an activity function to a state is simply to place the activity function pointers into an array that is indexed by state number. This is rather more complicated than might be assumed at first glance. Since all the elements of an array must be the same type, we have to handle the fact that all activity function argument signatures are *not* the same. Since events carry parameters and state activity functions accept those parameters as arguments, the worst case is that every activity function has a different argument signature. This is compounded by the desire that, for those states that do not have any activity to perform, we don't want to invoke an empty function just to have it return without doing anything useful.

To solve this problem, we create a tagged union to hold pointers to the activity function. The union is tagged by `StateId` and the union fields are pointers to an activity function (appropriately typed for its argument signature) or `void` if there is no activity for the state.

```

<<state model declarations>>=
const ActionElement = MakeActionElement();

```

```

<<state model functions>>=
/// The `MakeActionElement` activity function creates a tagged union type that is suitable
/// for holding a pointer to an activity function associated with a state machine state.
fn MakeActionElement() type {
    comptime validateStateActivities();

    const state_id_values = enums.values(StateId);
    var activities_found: usize = 0;

    var element_fields: []const UnionField = &[_]UnionField{};
    inline for (state_id_values) |state_id| {
        const state_name = @tagName(state_id);
        if (hasStateActivityFunction(state_name)) {
            // This computes a type that is a pointer to an activity function found
            // in the `StateActivities` namespace.
    }
}

```

```

const ActionType = @TypeOf(&@field(model_spec.StateActivities, state_name));
element_fields = element_fields ++
&[_]UnionField{.{
    .name = state_name,
    .type = ActionType,
    .alignment = if (@sizeOf(ActionType) == 0) 1 else @alignOf(ActionType),
}};
activities_found += 1;
} else {
    element_fields = element_fields ++
&[_]UnionField{.{
    .name = state_name,
    .type = void,
    .alignment = 1,
}};
}
}

// It's easy to forget to declare the activity functions public.
if (activities_found == 0) @compileError(
    "no activity functions found: check that activity functions are declared public");

return @Type(.{
    .@"union" = .{
        .layout = .auto,
        .tag_type = StateId,
        .fields = element_fields,
        .decls = &.{},
    },
});
}
}

```

Similar to the transition map, the action map is a constant computed at `comptime`.

```

<<state model declarations>>=
const activity_map = makeActivityMap();

```

Having an appropriate type for the action map, we use an `EnumArray` to hold the mapping between states and their corresponding activity functions.

```

<<state model functions>>=
/// The `makeActivityMap` function returns an `EnumArray` value that
/// contains the tagged union values for the activity functions of the
// state model.
fn makeActivityMap() enums.EnumArray(StateId, ActionElement) {
    var act_table: enums.EnumArray(StateId, ActionElement) = .initUndefined();
    inline for (enums.values(StateId)) |state_id| {
        const state_name = @tagName(state_id);
        if (hasStateActivityFunction(state_name)) {
            act_table.set(
                state_id,
                @unionInit(

```

```

        ActionElement,
        state_name,
        &@field(model_spec.StateActivities, state_name),
    ),
);
} else {
    act_table.set(
        state_id,
        @unionInit(ActionElement, state_name, {}),
    );
}
}

return act_table;
}

```

The final step of transition into a new state is to execute the activity function. This is a complicated interaction between runtime and `comptime`. The strategy is to use `inline else` to compute a switch statement to determine the state for which we want to execute its activity function. Another `inline else` is used to compute a switch statement based on the event which carries the parameter payload.

```

<<state model functions>>=
/// The `action` activity function invokes the state activity function for
/// `state` using the event parameter payload contained in `event`.
fn activity(
    state: StateId,
    event: Event,
) void {
    // `activity_spec` is a tagged union of all the activity function pointers.
    const activity_spec = activity_map.get(state); // Run time known.
    switch (activity_spec) {
        // The inline else creates tests for all the values for the union value
        // and its tag.
        inline else => |activity_fn_ptr, state_tag| {
            // Return early if there is no activity function.
            if (@TypeOf(activity_fn_ptr) == void) return;

            // Now we have to pair the activity function with the event arguments.
            // The `Event` type is a tagged union of tuples that are suitable
            // as argument tuples to the activity function.
            // We must compare the tag of `event` to the all the tags of Event type.
            switch (event) {
                // Again, the compiler generates a series of values for args and event_tag
                inline else => |args, event_tag| {
                    // We are only interested in those events which cause an inbound
                    // transition to `state`. If the event does not cause an inbound
                    // transition to the state the `@call` is discarded at comptime.
                    // This test is necessary because the nested inline else prongs
                    // compute the Cartesian product of the state and event ids. Not all
                    // combinations of state and event product produce an inbound
transition.

```

```

        // Note this is comptime evaluated since both `state_tag` and
        // `event_tag` are both comptime known.
        if (comptime isInboundEvent(state_tag, event_tag)) {
            @call(.auto, activity_fn_ptr, args);
        }
    },
},
},
}
}

```

Transitioning State Machines

We now have the ability to compute the transition function of the state model and to invoke the activity function of the target state of a transition. The dispatch mechanism, which we have not seen yet, uses a queue to hold signaled events. Dispatching involves taking one event from the queue and handing it to the target state model to dispatch to the target state machine indicated by the Event value. The ``reception'' of the event by the state machine pulls together the computation of the transition and the invocation of the corresponding activity function (if any).

As mentioned earlier, a state machine may signal an event to itself, *i.e.* a *continuation* event. Continuation events are handled in preference to receiving another interactive event from another state machine. So, receiving an event by a state machine involves processing the event and any continuation events that a state activity may signal. A state activity may signal at most one continuation event. Of course, the continuation events must be stored somewhere.

```

<<state model variables>>=
pub var continuation_events: [RelClassType.capacity]?Event = @splat(null);

```

```

<<state model public functions>>=
/// The `receiveEvent` function processes the event given by, `event`,
/// computing the transition function and invoking any activity function
/// associated with the target state of the transition.
/// Processing continues to consume any continuation events signaled
/// by the state machine to itself.
/// The function returns `true` if the receiving state machine reaches
/// a terminal state and `false` otherwise.
pub fn receiveEvent(
    self: Self,
    event: Event,
) bool {
    const received_inst_ref = switch (event) {
        inline else => |e| e.@"0",
    };
    const tuple_index = received_inst_ref.tuple_ref.tuple_id;
    assert(continuation_events[tuple_index] == null);

    var reached_terminal = self.receiveOneEvent(event);

    // Put some limit on the number of sequential transitory states to
    // prevent an infinite loop. The most common case is one continuation

```

```

// event. Two or three arise less frequently, usually in the case
// of association assigner state models. In this context, 50 is
// tantamount to infinity.
var continuation_limit: usize = 50;
while (continuation_limit != 0) : (continuation_limit -= 1)
{
    if (continuation_events[tuple_index]) |continuation_event| {
        // Signaling a continuation event from a terminal state is not allowed.
        // The state machine instance is deleted after a terminal state and
        // so there is no instance that can continue the processing.
        // This is an analysis error detected at runtime.
        if (reached_terminal) {
            panic("{s}: signaled a continuation event from a terminal state activity",
.{
                @tagName(RelClassType.class_id),
            });
        }
        continuation_events[tuple_index] = null;
        RelClassType.trace_operations.continue_event(
            received_inst_ref,
            @intFromEnum(meta.activeTag(continuation_event)),
        );
        reached_terminal = self.receiveOneEvent(continuation_event);
    } else break;
} else {
    panic("{s}: exceeded limit for sequential continuation events", .{
        @tagName(RelClassType.class_id),
    });
}

return reached_terminal;
}

```

So, receiving an event is divided into processing one event and then looping to continue processing any continuation events that might have been generated.

Receiving a single event must diagnose several errors that ultimately are analysis errors, but which can only be detected at runtime. All of these errors create a panic condition. The following cases must be handled:

1. It is possible for an state machine to be signaled and between the time the signal occurs and the time the event is received, the state machine may have been deleted. This is called an *event in flight* error. It is fundamentally an analysis error and indicates that the deletion protocol defined in the analytical model is faulty.
2. The target state of the transition is the *never* non-transitioning state. Transitioning to the **never** state causes a panic condition.

```

<<state model public functions>>=
/// The `receiveOneEvent` function processes the event given by, `event`,
/// causing a transition and executing any association activity function.
/// The function returns `true` if the transition causes the state machine
/// to enter a `terminal` state.

```

```

pub fn receiveOneEvent(
    self: Self,
    event: Event,
) bool {
    const event_id = meta.activeTag(event);
    const received_inst_ref = switch (event) {
        inline else => |e| e.@"0",
    };
    // Event in flight error check.
    received_inst_ref.tuple_ref.verifyAsNot(.free);

    const tuple_index = received_inst_ref.tuple_ref.tuple_id;
    var current_state = current_states[tuple_index];

    // Creation events are those events that are signaled to an instance state machine
    // where the instance was created in the reserved state. Here we active the
    // class instance before receiving the event.
    if (received_inst_ref.tuple_ref.isReserved()) {
        received_inst_ref.activateReserved();
        RelClassType.trace_operations.created_inst_async(received_inst_ref);
    }

    const target_state = self.transition(current_state, event_id);

    switch (target_state) {
        .never => {
            // Rather than panic here, we allow the domain to catch the trace
            // and handle the consequences elsewhere. This is useful for
            // testing purposes when you want to force an undefined transition,
            // but do not necessarily want to create a panic condition.
            RelClassType.trace_operations.undefined_transition(
                received_inst_ref,
                @intFromEnum(current_state),
                @intFromEnum(event_id),
            );
            return false;
        },
        .ignore => {
            RelClassType.trace_operations.transitioned(
                received_inst_ref,
                @intFromEnum(current_state),
                @intFromEnum(event_id),
                @intFromEnum(target_state),
            );
            return false;
        },
        else => {
            const prev_state = current_state;
            current_state = stateFromTargetState(target_state);
            current_states[tuple_index] = current_state;

            // The scope of a relation value expression is the activity function for a

```

```

state.

    relExprBegin();
    activity(current_state, event);
    relExprEnd();

    // Trace only after the activity is executed.
    RelClassType.trace_operations.transitioned(
        received_inst_ref,
        @intFromEnum(prev_state),
        @intFromEnum(event_id),
        @intFromEnum(target_state),
    );
    RelClassType.trace_operations.exited_state(received_inst_ref, @intFromEnum
(prev_state));

        return if (isTerminalState(current_state)) true else false;
    },
}
}

```

The tracing performed by the `receiveOneEvent` is critical to understanding the dynamics of the state model as well as for testing the path coverage of the state model graph.

```

<<state model declarations>>=
const InstanceRef = RelClassType.InstanceRef;

```

```

<<instance reference declarations>>=
/// The `continueWith` function signals a continuation event where the
/// event type is given by `event_id` and the event parameters are given
/// by `event_payload`. The continuation event is received immediately
/// after the activity function which invokes this function completes.
/// It is not allowed to signal more than one continuation event in any
/// given activity function.
pub fn continueWith(
    inst_ref: InstanceRef,
    comptime event_id: EventId,
    event_payload: anytype, // tuple of additional activity arguments
) void {
    const event = inst_ref.makeEvent(event_id, event_payload);
    const cont_event_ptr =
        &@TypeOf(inst_ref.classvar).ClassModel.continuation_events[inst_ref.tuple_ref
.tuple_id];
    assert (cont_event_ptr.* == null);
    cont_event_ptr.* = event;
}

```

State Model Code Layout

```

<<state_model.zig>>=

```

```

///! The `state_model.zig` file implements a Moore type state model that can be
///! applied to Relational Classes to impart asynchronous behavior to the class.
///! The state model consists of a set of states, a set of events, a set of transitions,
///! and other minor controls. See the `StateModel` type function for the
///! complete interface to create a StateModel.
<<edit warning>>
<<copyright info>>

const std = @import("std");
const builtin = @import("builtin");
const EnumField = std.builtin.Type.EnumField;
const StructField = std.builtin.Type.StructField;
const UnionField = std.builtin.Type.UnionField;

const testing = if (builtin.os.tag == .freestanding)
    @import("resee_testing")
else
    std.testing;
const meta = std.meta;
const mem = std.mem;
const enums = std.enums;
const fmt = std.fmt;
const math = std.math;
const ascii = std.ascii;

const assert = std.debug.assert;
const panic = std.debug.panic;

const class = @import("class.zig");
const RelClass = class.RelClass;
const ClassGeneratedId = class.ClassGeneratedId;
const generated_id_default = class.generated_id_default;

const relvalue = @import("relvalue.zig");
const relExprBegin = relvalue.relExprBegin;
const relExprEnd = relvalue.relExprEnd;

pub const version = "1.0.0-a1";

<<active class module declarations>>

<<active class module public functions>>

<<active class module private functions>>

<<active class module tests>>

```

[1] We use UML terminology here. The computation performed by a state machine are called *activities* which consist of a set of *actions*.

Managing Application Data

To recap, the previous two chapters showed the fundamental constructs that are used in ReSEE to manage data and execution. Data management is based on relational algebra and execution sequencing is based on Moore-type finite state models. These foundation are sparse in their operations and, if used directly, require a more verbose solution to common application models.

In this chapter and the one that follows, we expand upon the basic management foundation to obtain a set of capabilities that make it possible for the implementation of ReSEE to match the needs of a domain model.

Buckets of Moonbeams in my Hand

In a previous [chapter](#), we discussed the usage of the term, *class*, as a concept used to represent the logic of application subject matter in data. Despite the many definitions applied to the word, *class*, we continue to use it. But in recognition of the many meanings the term carries, we use the term, *relational class*, to refer to our particular refinement of a class when it is used for application data. A further refinement to our notion of class will be made when we discuss managing the execution sequencing of the application. For now, Relational classes are based on Relation Variables that include additional capabilities and integrate closely with concepts of Associations and Generalizations. In this section, the additional requirements on a relational class that make them useful in encoding application logic in data are discussed. Later sections continue to augment the capabilities of relational classes.

Additional Requirements on Relational Classes

The concept of a *class*, by our definition, is based on and implemented by a Relation Variable. But relational classes bring along a few additional requirements imposed upon them by the execution model.

- Relational classes can provide values for an identifying attribute that are unique within a given class. There can be at most one such identifier for a class and it consists of a single attribute. This is helpful for those situations where class instances do not have a natural identifier and an arbitrary identifier can be used to ensure identity constraints. It is also useful for those situations where a natural identifier consists of several attributes, a single identifying attribute is more convenient, and the natural identifier is needed for descriptive purposes.
- Class attributes may be designated *mathematically dependent*. For dependent attributes, a formula is executed to compute its value whenever the attribute is read. This is helpful for those occasions when the dependent quantity is calculated from other descriptive attributes. For example, the area of a rectangle is the product of its width and length. It is sometimes more convenient to have a “dependent” area attribute that is simply computed each time it is needed. A dependent attribute may *not* be used as an identifying attribute. Dependent attributes may not be updated. Note that calculation of dependent attributes is intended to be local and only the value of a Relational Tuple is provided to the formula.
- Relational class instances may be created synchronously or asynchronously. This is discussed further in a later [section](#).

Representing Relational Classes

The foundation of a relational class is a RelVar and its components, RelValue and RelTuple.

```

<<class module declarations>>=
const relvar = @import("relvar.zig");
const RelVar = relvar.RelVar;
const RelVarError = relvar.RelVarError;
const Loading = relvar.Loading;
const GenerationNumber = relvar.GenerationNumber;
const undefined_generation = relvar.undefined_generation;

const relvalue = @import("relvalue.zig");
const RelValue = relvalue.RelValue;
pub const RelationCapacity = relvalue.RelationCapacity;

const reltuple = @import("reltuple.zig");
const RelTuple = reltuple.RelTuple;

const state_model = @import("state_model.zig");
pub const StateModelSpec = state_model.StateModelSpec;
pub const TransitionSpec = state_model.TransitionSpec;
const StateModel = state_model.StateModel;

```

Generated Identifying Attributes

Every instance of every class has a class-generated, intrinsic, architecturally-defined identifier. The generated identifier is unique only within the class. It is *not* unique within the subject matter domain, the entire system, or globally unique^[1]. These identifiers are used internally for data management. By request, an identifier whose value is equal to the intrinsic identifier may appear as an attribute of the class. Class generated identifiers are attributes that use a specially provided type that is given a unique value when an instance is created for the class.

A Relation Variable already defines a TupleRef type and the information carried by that type is suitable for use as the type of a generated identifier.

```

<<class module declarations>>=
/// The `ClassGeneratedId` type is used as an attribute type when
/// the value represents a generated instance id.
pub const ClassGeneratedId = RelTuple(struct {
    tuple_id: RelationCapacity = 0,
    gen_id: GenerationNumber = undefined_generation,
});
/// A convenient initializer for generated identifying attributes that highlights
/// the attribute as being generated.
pub const generated_id_default = ClassGeneratedId.create(.{});

```

Dependent Attributes

For mathematically dependent attributes, the attribute appears naturally in the heading of the class, but is designated as dependent by supplying the formula for computing the attribute value when the class is created.

```

<<class declarations>>=
/// The type signature for a dependent attribute formula. The first
/// argument is the tuple value which contains the dependent attribute
/// being read. The second argument is the `AttributeId` of the dependent
/// attribute. It is this attribute whose value is to be computed. The
/// type of the function return value is a tuple. The usual expectation
/// is that this function computes the value of `attr_id` and perform a
/// `tuple update` on the input tuple returning the result of the update.
pub const DependentAttrFormula =
    *const fn(tuple: Tuple, comptime attr_id: AttributeId) Tuple;

```

Some syntactic sugar can make plain the dependent attribute during create operations.

```

<<class module declarations>>=
/// A convenient initializer for dependent attributes that highlights
/// the attribute's usage.
pub const dependent_attr_default = undefined;

```

Defining a Relational Class Data Type

Having decided upon the implementation of the additional requirements, we show a type function that creates a type suitable for use as a Relational Class.

```

<<class module declarations>>=
/// The `RelClass` function is a type function which returns a namespace
/// `struct` type for a relational class with the properties given by the
/// arguments. The `class_tag` argument is an enum literal that is used
/// to identify the class. The `Heading` argument must be an acceptable
/// `RelVar` heading, which is further validated for dependent attributes.
/// The identifiers of the class are given by, `identifiers`, which
/// is a slice of `AttributeId` slices. The maximum tuple storage
/// capacity for the class is given by the `max_capacity` argument.
/// The `dep_attr_specs` argument is a slice of dependent attribute
/// definitions. Each definition is an attribute id / attribute formula
/// 2-tuple. The `class_options` can be supplied to control several
/// aspects of the class. See `RelClassOptions` for definitions of
/// what can be controlled. The value of the `max_capacity` argument
/// set the upper bound on the number of instances of the class.
/// The `initial_population` argument is a slice of `Heading` values
/// that are installed in the `RelClass` storage when the type is created.
pub fn RelClass(
    comptime class_tag: @Type(.enum_literal),
    comptime Heading: type,
    comptime identifiers: []const []const RelTuple(Heading).AttributeId,
    comptime dep_attr_specs: []const DependentAttrSpec(RelTuple(Heading)),
    comptime class_options: RelClassOptions,

```

```

    comptime max_capacity: RelationCapacity,
    comptime initial_population: []const struct { @Type(.enum_literal), Heading },
) type {
    validateDependentAttrs(Heading, dep_attr_specs, identifiers);
    return struct {
        const Self = @This();

        <<class variables>>
        <<class declarations>>
        <<class public functions>>
        <<class private functions>>
    };
}

```

```

<<class module declarations>>=
pub const RelClassOptions = struct {
    opt_state_model_spec: ?*const StateModelSpec = null,
    polymorphic_events: []const []const u8 = &.{},
    load_factor: ?Loading = null,
};

```

Dependent attributes are not allowed to be part of an identifier. That validation is done first to catch such errors at compile time.

```

<<class module declarations>>=
pub fn DependentAttrSpec(
    comptime DepTuple: type,
) type {
    return struct {
        DepTuple.AttributeId,
        *const fn(
            tuple: DepTuple,
            comptime attr_id: DepTuple.AttributeId,
        ) DepTuple,
    };
}

```

```

<<class module private functions>>=
/// The `validateDependentAttrs` function validates the dependent
/// attributes with respect to the class identifiers. All dependent
/// attributes must be contained by the heading and _not_ contained in
/// any identifier.
fn validateDependentAttrs(
    comptime Heading: type,
    comptime dep_attr_specs: anytype,
    comptime identifiers: []const []const RelTuple(Heading).AttributeId,
) void {
    inline for (dep_attr_specs) |dep_attr_definition| {

```

```

    inline for (identifiers) |identifier| {
        var match_id_attr = false;
        inline for (identifier) |id_attr| {
            match_id_attr = match_id_attr or (id_attr == dep_attr_definition.@"0");
        }
        if (match_id_attr)
            @compileError("dependent attribute '" ++
                @tagName(dep_attr_definition.@"0") ++
                "' cannot be used as an identifying attribute");
    }
}
}

```

```

<<class variables>>=
pub var class_number: u8 = undefined;
pub const class_id = class_tag;

```

The foundation of tuple storage for a relational class is a RelVar

```

<<class variables>>=
/// The namespace struct that contains the relation variable for the class.
pub const ClassVar = RelVar(
    TupleBase,
    identifiers,
    max_capacity,
    class_options.load_factor,
    &resolveGeneratedIds(initial_population),
);
pub const classvar: ClassVar = .{};

```

Because we have introduced the notion of a class generated identifiers, we must “patch” any initial instance population to fill in the class generated identifier. Note there is an implicit out-of-band agreement between this function and the relvar.reserve function as to how the generation of class identifiers works numerically.

```

<<class declarations>>=
pub const InitialInstance = struct {
    @Type(.enum_literal),
    TupleBase,
};

```

```

<<class private functions>>=
fn resolveGeneratedIds(
    comptime pop: []const InitialInstance,
) [pop.len] InitialInstance {
    var gen_pop: [pop.len]InitialInstance = undefined;
    @memcpy(&gen_pop, pop);

    if (generated_attr) |gen_attr| {
        inline for (&gen_pop, 0..) [*init_inst, inst_index] {

```

```

        @field(init_inst@"1", @tagName(gen_attr)) = ClassGeneratedId.create(.{
            .tuple_id = inst_index,
            .gen_id = 1,
        });
    }

    return gen_pop;
}

```

As before, we find it convenient to include a `void` field to help identify a relational class type.

```

<<class declarations>>=
pub const is_relclass: void = {};

```

```

<<class module public functions>>=
/// The `isRelClassType` function returns `true` if the value of `RelClassType`
/// argument was generated by the `RelClass` type function and `false` otherwise.
pub fn isRelClassType(
    comptime RelClassType: type,
) bool {
    return @typeInfo(RelClassType) == .@"struct" and @hasDecl(RelClassType, "is_relclass");
}

```

Following our previous pattern, we hold some constants around for symbolic convenience.

```

<<class declarations>>=
/// The base structure of a tuple for this relation variable.
pub const TupleBase = Heading;
/// The type of a relation tuple for this class heading.
pub const Tuple = RelTuple(TupleBase);
/// The type of a relation value for this class heading.
pub const RelValueType = RelValue(TupleBase);
/// The enumerated type representing tuple attributes.
pub const AttributeId = Tuple.AttributeId;
/// Function to return the type of an attribute.
pub const AttributeType = Tuple.AttributeType;
/// Function to return the tuple type of an identifier.
pub const IdentifierAttributes = ClassVar.IdentifierAttributes;
/// Function to return the attribute values from a tuple for a given identifier.
pub const IdentifierValues = ClassVar.IdentifierValues;
/// The type of sets of relvar tuples.
pub const TupleRefSet = ClassVar.TupleRefSet;
/// The value of an empty TupleRefSet.
pub const emptyTupleRefSet: TupleRefSet = .initEmpty();
/// Initial instances enumeration.
pub const InitialInstanceId = ClassVar.InitialInstanceId;
/// The maximum number of tuples that can be stored in a `RelClass` variable.
pub const capacity = ClassVar.relvar_capacity;

```

We also hold the definitions of the dependent attributes as a constant of the `RelClass`. The dependent attribute information is stored in the following structure.

```
<<class declarations>>=
/// A structure used to convey dependent attribute information to the `RelClass` function.
pub const DependentAttrProperties = struct {
    attr_id: AttributeId,
    formula: DependentAttrFormula,
};
```

We create a type constant to hold the dependent attribute information for later use.

```
<<class declarations>>=
const dependentAttrs = ablk: {
    var dattr_properties: []const DependentAttrProperties = &[_]DependentAttrProperties{};
    // dep_attr is a pair -- attribute id, formula
    for (dep_attr_specs) |dep_attr_definition| {
        assert(dep_attr_definition.len == 2);
        dattr_properties = dattr_properties ++ &[_]DependentAttrProperties.{.
            .attr_id = dep_attr_definition[@"0"],
            .formula = dep_attr_definition[@"1"],
        });
    }
    break :ablk dattr_properties;
};
```

A convenience function is provided to check if any given attribute is dependent.

```
<<class private functions>>=
/// The `isDependentAttr` function returns a `DependentAttrFormula` if the
/// attribute indicated by the `attr_id` argument was declared as
/// mathematically dependent and `null` otherwise.
fn isDependentAttr(
    comptime attr_id: AttributeId,
) ?DependentAttrFormula {
    inline for (dependentAttrs) |dependent_attr| {
        if (attr_id == dependent_attr.attr_id) {
            return dependent_attr.formula;
        }
    }
    return null;
}
```

We also want to store away any identifying attribute which was requested to have its value generated and to validate that there is at most one.

```

<<class declarations>>=
/// An optional `AttributeId` containing the value of any class
/// generated identifying attribute, if any.
const generated_attr: ?AttributeId = gblk: {
    var gen_attr: ?AttributeId = null;

    for (identifiers) |identifier| {
        for (identifier) |id_attr| {
            if (AttributeType(id_attr) == ClassGeneratedId) {
                if (gen_attr == null) {
                    gen_attr = id_attr;
                } else {
                    @compileError("at most one attribute may be class generated");
                }
            }
        }
    }

    break :gblk gen_attr;
};

```

```

<<class public functions>>=
pub fn init(
    number: u8,
) Self {
    class_number = number;
    _ = ClassVar.init();
    return .{};

}

```

Tests

```

<<class module tests>>=
test "class initialization" {
    const Heading1 = struct {
        name: []const u8,
        length: usize,
        weight: f32,
    };
    const Class1 = RelClass(.class1, Heading1, &.{ &.{ .name } }, &.{}, .{}, 10, &.{});
    _ = Class1.init(1);
    try testing.expectEqual(@as(usize, 0), Class1.dependentAttrs.len);
    try testing.expectEqual(null, Class1.generated_attr);
    try testing.expectEqual(@as(usize, 10), Class1.capacity);

    const Heading2 = struct {
        id: ClassGeneratedId = generated_id_default,
        length: usize,
    };
}

```

```

        width: usize,
        weight: f32,
    };
    const Class2 = RelClass(.class2, Heading2, &{ &{ .id } }, &{}, .{}, 20, &{});
    _ = Class2.init(2);
    try testing.expectEqual(@as(usize, 0), Class2.dependent_attrs.len);
    try testing.expect(null != Class2.generated_attr);
    try testing.expectEqual(@as(usize, 20), Class2.capacity);
}

```

Relational Class Creation

A relational class instance may be created in one of two time frames:

1. Synchronously, *i.e.* the class instance is available immediately after the create function call.
2. Asynchronously, *i.e.* the class instance must be “activated” before it is available for general use. Activation occurs as part of dispatching a state machine event and is discussed later when the [execution management](#) is discussed.

In both creation cases, the tuple value must be specified to the creation function. For the synchronous case, the instance is immediately activated. For the asynchronous case, other code in the execution model activates the instance at the appropriate time. In the asynchronous case, we are “stealing” a relvar slot to hold the tuple value until such time as the execution sequencing code decides to activate it.

There are also two possible errors when a relational class is created:

1. There is insufficient space. Recall, that the maximum number of tuples that may be stored in a class is specified at compile time.
2. The creation would violate the identity constraints for the class. Recall, that all values of identifiers for a class must be unique.

The most common use case for relational class creation is to assume a “perfect” storage mechanism and the errors don’t happen. For the targeted class of applications, an upper bound on tuple storage is set and the system is exercised ensure those bounds are not exceeded. For this case, the errors are considered a *panic* situation and the system is in a state where it cannot make any recovery.

But there are cases, particularly at the boundary of a domain, where class instances are created in response to meeting requirements allocated to the domain. For example, communications messages arriving at a domain may cause class instance creation. We do not want an overly active client domain to cause the system to panic because it had an error and generated too many spurious requests. In this case, we want to know if the allocation causes an error and take any appropriate recovery. The usual recovery is to ignore the request that caused the storage to be depleted. For communications, that strategy can be effective if the protocol can handle the loss of information.

```

<<class declarations>>=
/// The `InstanceRef` type is used to hold the components of a
/// direct reference to a class instance.
pub const InstanceRef = struct {

```

```

    classvar: Self = .{},
    tuple_ref: ClassVar.TupleRef,

    <<instance reference declarations>>
};


```

```

<<instance reference declarations>>=
pub fn initFromIndex(
    inst_index: usize,
) InstanceRef {
    return .{
        .tuple_ref = ClassVar.TupleRef.initFromIndex(inst_index),
    };
}

pub fn instanceSetFromInstanceRef(
    inst_ref: InstanceRef,
) InstanceSet {
    return .initFromInstanceRef(inst_ref);
}

pub fn dereference(
    inst_ref: InstanceRef,
) Tuple {
    return inst_ref.tuple_ref.dereference();
}

pub fn reserve() RelVarError!InstanceRef {
    return .{
        .tuple_ref = classvar.reserveOrError(),
    };
}

pub fn destroy(
    inst_ref: InstanceRef,
) void {
    trace_operations.deleted_inst(inst_ref); // Report before the actual destruction.
    inst_ref.tuple_ref.destroy();
}

pub fn format(
    inst_ref: InstanceRef,
    writer: *std.io.Writer,
) std.io.Writer.Error!void {
    try writer.print("{any}", .{inst_ref.tuple_ref});
}

pub fn traverse(
    inst_ref: InstanceRef,
    comptime relationship: anytype,
    comptime traversal_phrase: anytype,
    comptime dest_class: anytype,

```

```

) @TypeOf(relationship).InferDestRelClass(traversal_phrase, dest_class).InstanceSet {
    return if (comptime association.isAssociationType(@TypeOf(relationship)))
        inst_ref.traverseAssociation(relationship, traversal_phrase, dest_class)
    else if (comptime generalization.isGeneralizationType(@TypeOf(relationship)))
        inst_ref.traverseGeneralization(relationship, traversal_phrase, dest_class)
    else
        @compileError("relationship argument is neither an association nor
generalization");
}

pub fn signal(
    src_inst: InstanceRef,
    target: anytype,
    comptime event_id: @Type(.enum_literal),
    event_payload: anytype, // tuple of additional activity arguments
) void {
    const TargetClass = @TypeOf(target.classvar);
    const target_event_id = @as(TargetClass.EventId, event_id);
    const src_class_ref = src_inst.classInstanceRef();

    if (@hasField(@TypeOf(target), "tuple_ref_set")) {
        var set_iter = target.iterator();
        while (set_iter.next() |target_inst| {
            const event = target_inst.makeEvent(target_event_id, event_payload);
            executor.signal(
                src_class_ref,
                target_inst.classInstanceRef(),
                @ptrCast(&event),
            );
            trace_operations.signaled_event(
                src_inst,
                target_inst,
                @intFromEnum(target_event_id),
            );
        }
    } else if (@hasField(@TypeOf(target), "tuple_ref")) {
        const event = target.makeEvent(target_event_id, event_payload);
        executor.signal(
            src_class_ref,
            target.classInstanceRef(),
            @ptrCast(&event),
        );
        trace_operations.signaled_event(
            src_inst,
            target,
            @intFromEnum(target_event_id),
        );
    } else {
        @compileError("expected InstanceRef or InstanceSet");
    }
}

```

```

<<class module declarations>>=
pub const ClassInstanceRef = RelTuple(struct {
    class_index: u8,
    tuple_id: RelationCapacity,
    gen_id: GenerationNumber,
});

```

```

<<instance reference declarations>>=
fn classInstanceRef(
    inst_ref: InstanceRef,
) ClassInstanceRef {
    return ClassInstanceRef.create(.{
        .class_index = class_number,
        .tuple_id = inst_ref.tuple_ref.tuple_id,
        .gen_id = inst_ref.tuple_ref.gen_id,
    });
}

```

For now, we need to build some foundation that allows this behavior.

```

<<class public functions>>=
/// The `create` function creates a new class instance whose
/// attribute values are given by the `tuple` argument. The returned
/// value is an `InstanceRef`, identifying the newly created instance.
/// Any error detected during the creation is considered a panic
/// condition.
pub fn create(
    self: Self,
    tuple: TupleBase,
) InstanceRef {
    return self.createOrError(tuple) catch |err|
        panic("failed to create class instance: '{s}'\n", {@errorName(err)});
}

```

```

<<class public functions>>=
/// The `createOrError` function creates a new class instance
/// whose attribute values are given by the `tuple` argument.
/// The returned value is an `InstanceRef`. Any error detected during
/// the creation is also returned.
pub fn createOrError(
    self: Self,
    tuple: TupleBase,
) RelVarError!InstanceRef {
    const inst_ref = try self.createReservedOrError(tuple);
    try inst_ref.activateReservedOrError();
    if (has_state_model) classmodel.initInDefaultState(inst_ref);
}

```

```

    trace_operations.created_inst(inst_ref);
    return inst_ref;
}

```

All instance creations start by creating a tuple in a reserved relation variable slot. This is the core function upon which all relational class tuple allocation is based.

```

<<class public functions>>=
pub fn createReserved(
    self: Self,
    tuple: TupleBase,
) InstanceRef {
    return self.createReservedOrError(tuple) catch |err|
        panic("failed to reserve instance storage: '{s}'\n", .{@errorName(err)});
}

```

```

<<class public functions>>=
/// The `createReservedOrError` function creates a new tuple in the given class,
/// setting its allocation status to reserved. Failure to create the tuple
/// returns an error. Upon success, an `InstanceRef` value is returned.
/// The `InstanceRef` value uniquely identifies the new tuple within the
/// given class.
pub fn createReservedOrError(
    _: Self,
    tuple: TupleBase,
) RelVarError!InstanceRef {
    var insert_tuple = tuple;
    const tuple_ref = try ClassVar.TupleRef.reserve();
    if (generated_attr) |gen_attr| { ①
        const inst_id_tuple = ClassGeneratedId.create(.{
            .tuple_id = tuple_ref.tuple_id,
            .gen_id = tuple_ref.gen_id,
        });
        @field(insert_tuple, @tagName(gen_attr)) = inst_id_tuple;
    }
    try tuple_ref.updateReserved(insert_tuple);

    const inst_ref: InstanceRef = .{ .tuple_ref = tuple_ref };
    if (has_state_model and ClassModel.has_pseudo_initial) {
        classmodel.initInPseudoInitialState(inst_ref);
    }

    return inst_ref;
}

```

- ① If there is to be a generated identifier, that value is constructed here.

Once created, we must activate the storage slot. Again two flavors of this function are provided. For those creation attempts which can result in a panic, the `activateReserved` function is provided.

```
<<instance reference declarations>>=
pub fn activateReserved(
    inst_ref: InstanceRef,
) void {
    inst_ref.activateReservedOrError() catch |err|
        panic("failed to activate tuple: '{s}'\n", {.@errorName(err)});
}
```

If it is necessary to capture any activation errors, the `activateReservedOrError` function is provided.

```
<<instance reference declarations>>=
pub fn activateReservedOrError(
    inst_ref: InstanceRef,
) RelVarError!void {
    try inst_ref.tuple_ref.activate(); ①
}
```

① On failure, `activate` deletes the tuple.

```
<<class public functions>>=
pub fn createSet(
    self: Self,
    tuples: []const TupleBase,
) InstanceSet {
    return self.createSetOrError(tuples) catch |err| {
        panic("failed to populate instance set: '{s}'\n", {.@errorName(err)});
    };
}
```

```
<<class public functions>>=
/// The `createSetOrError` function inserts zero or more class instances
/// into the given class. Population is an all-or-nothing operation.
/// Either all the tuples requested are inserted correctly, none of them
/// are inserted.
pub fn createSetOrError(
    self: Self,
    tuples: []const TupleBase,
) RelVarError!InstanceSet {
    var population = emptyTupleRefSet;
    errdefer population.destroy();

    for (tuples) |tuple| {
        const inserted_ref = try self.createOrError(tuple);
```

```

        population.set(inserted_ref.tuple_ref.tuple_id);
    }

    return .initFromTupleRefSet(population);
}

```

```

<<instance reference declarations>>=
/// The `createInState` function creates an active instance of the
/// Relational Class and places it in the state given by `state_id`.
/// It returns an `InstanceRef` of cardinality one referencing the newly
/// created instance.
pub fn createInState(
    self: Self,
    tuple: TupleBase,
    state_id: StateId,
) InstanceRef {
    return self.createInStateOrError(tuple, state_id) catch |err| {
        panic("failed to create class instance: '{s}'\n", .{@errorName(err)});
    };
}

```

```

<<instance reference declarations>>=
/// The `createInStateOrError` function creates an active instance of the
/// Relational Class and places it in the state given by `state_id`.
/// It returns an `InstanceRef` of cardinality one referencing the newly
/// created instance.
pub fn createInStateOrError(
    self: Self,
    tuple: TupleBase,
    state_id: StateId,
) RelVarError!InstanceRef {
    if (has_state_model) {
        const inst_ref = try self.createOrError(tuple);
        classmodel.initState(inst_ref, state_id);
        return inst_ref;
    } else
        @compileError("the '" ++ tagName(class_id) ++ "' class has no state model");
}

```

```

<<class public functions>>=
/// The `createAsync` function reserves an active instance of the
/// Relational Class and places it in the pseudo-initial state, `@` .
/// It returns an `InstanceRef` referencing to the newly created instance.
pub fn createAsync(
    self: Self,
    tuple: TupleBase,
) InstanceRef {
    return self.createAsyncOrError(tuple) catch |err| {
        panic("failed to create class instance: '{s}'\n", .{@errorName(err)});
    };
}

```

```
}
```

```
<<class public functions>>=
/// The `createAsyncOrError` function reserves an active instance of the
/// Relational Class and places it in the pseudo-initial state, `@`.
/// It returns an `InstanceRef` of cardinality one referencing the newly
/// created instance or an error if the creation fails.
pub fn createAsyncOrError(
    self: Self,
    tuple: TupleBase,
) RelVarError!InstanceRef {
    if (has_state_model) {
        return try self.createReservedOrError(tuple);
    } else
        @compileError("the '" ++ tagName(class_id) ++ "' class has no state model");
}
```

```
<<class public functions>>=
/// The `updateFromRelValue` function updates all the tuples in the
/// given relational class using values from the `updates` argument.
/// The identifying attributes of each tuple in `updates` are used to
/// locate the tuple in the relation variable which is then overwritten
/// with the tuple from `updates`. Note, it is an error if a tuple from
/// `updates` is not found in the relation variable.
pub fn updateFromRelValue(
    _: Self,
    value: RelValueType,
) void {
    classvar.updateFromRelValue(value);
}
```

```
<<class public functions>>=
/// The `insertFromRelValue` inserts all the tuples from the relation variable
/// `value` into the given relation variable. The return value is an
/// `InstanceSet` of the inserted tuples. It is an error to attempt
/// to insert a duplicate tuple.
pub fn insertFromRelValue(
    _: Self,
    value: RelValueType,
) InstanceSet {
    return .initFromTupleRefSet(classvar.insertFromRelValue(value));
}
```

```

<<class public functions>>=
/// The `updateOrInsertFromRelValue` either updates existing tuples in
/// the given relation variable or, if the tuple does not exist in the
/// relation variable, it inserts the tuple from the relation value as
/// given by the `value` argument. In either case, upon successful
/// return, the relation variable holds all the tuples contained in
/// `value`. The return value of the function is an `InstanceSet`
/// indicating which tuples were inserted.
pub fn updateOrInsertFromRelValue(
    _: Self,
    value: RelValueType,
) InstanceSet {
    return .initFromTupleRefSet(classvar.updateOrInsertFromRelValue(value));
}

```

```

<<class public functions>>=
pub fn tupleFromInitialInstance(
    _: Self,
    initial_inst_id: InitialInstanceId,
) Tuple {
    return ClassVar.initFromInitialInstance(initial_inst_id).dereference();
}

```

Tests

```

<<class module tests>>=
test "class creation" {
    // Simple relational class -- only normal attributes.
    const Class_1 = RelClass(
        .class_1, // class_tag
        struct {
            name: []const u8,
            length: usize,
            weight: f32,
        }, // Heading
        &.{ // identifiers
            &{.name}, // identifier 0
        }, // identifiers
        &{}, // dependent_attr_props -- none
        .{}, // options -- all default
        10, // max_capacity
        &{}, // initial population
    );

    const class_1 = Class_1.init(2);
    try testing.expectEqual(@as(usize, 0), Class_1.dependentAttrs.len);
    try testing.expectEqual(null, Class_1.generatedAttr);
    try testing.expectEqual(@as(usize, 10), Class_1.ClassVar.relvar_capacity);
}

```

```

const created1 = class_1.create(.{
    .name = "foo",
    .length = 20,
    .weight = 17.5,
});
try testing.expectEqual(@as(usize, 1), class_1.cardinality());
try testing.expectEqualSlices(u8, "foo", created1.readAttribute(.name));

// attempt to create a duplicate
const dup1 = class_1.createOrError(.{
    .name = "foo",
    .length = 20,
    .weight = 17.5,
});
try testing.expectError(RelVarError.DuplicateTuple, dup1);

// create a tuple that is reserved.
const bar_inst = try class_1.createReservedOrError(.{
    .name = "bar",
    .length = 20,
    .weight = 17.5,
});
try bar_inst.activateReservedOrError();
try testing.expectEqualSlices(u8, "bar", bar_inst.readAttribute(.name));

// identity constraints are not evaluated until a tuple is activated.
// the following has a duplicate identifier value, but the length attribute is
different.
const dup_bar_inst = try class_1.createReservedOrError(.{
    .name = "bar",
    .length = 40,
    .weight = 17.5,
});
try testing.expectError(RelVarError.DuplicateTuple, dup_bar_inst
.activateReservedOrError());
// the duplicate tuple was discarded when the activation failed,
// i.e. the length attribute is 20, the same as the originally inserted tuple.
const selected_bar_inst = try class_1.selectByIdentifier(0, .{ .name = "bar" });
try testing.expectEqual(@as(usize, 20), selected_bar_inst.readAttribute(.length));

// Duplicate tuples are not allowed.
// For the populate function, either all tuples are inserted or none are.
// This test results in leaving the contents of `class1` undisturbed.
const dup_population = class_1.createSetOrError(&.{

    .{
        .name = "baz", // new
        .length = 20,
        .weight = 17.5,
    },
    .{
        .name = "bar", // duplicate
    }
});

```

```

        .length = 20,
        .weight = 17.5,
    },
    .{
        .name = "foo", // duplicate
        .length = 20,
        .weight = 17.5,
    },
});
try testing.expectError(RelVarError.DuplicateTuple, dup_population);
try testing.expectEqual(@as(usize, 2), class_1.cardinality());
try testing.expectEqualSlices(u8, "foo", created1.readAttribute(.name));

// Test a class-generated id.
const Class_2 = RelClass(
    .class_2, // class tag
    struct {
        id: ClassGeneratedId = generated_id_default,
        length: usize,
        width: usize,
        weight: f32,
    }, // Heading
    &.{ // identifiers
        &{.id},
    },
    &{}, // dependent_attr_props
    {}, // options
    20, // max capacity
    &{}, // initial population
);
const class_2 = Class_2.init(3);
try testing.expectEqual(@as(usize, 0), Class_2.dependentAttrs.len);
try testing.expectEqual(.id, Class_2.generatedAttr);
try testing.expectEqual(@as(usize, 20), Class_2.ClassVar.relvar_capacity);
const created2 = class_2.create(.{
    .length = 20,
    .width = 40,
    .weight = 17.5,
});
try testing.expectEqual(@as(usize, 40), created2.readAttribute(.width));

// Class-generated id and mathematically dependent attribute.
const Heading3 = struct {
    id: ClassGeneratedId = generated_id_default,
    length: usize,
    width: usize,
    weight: f32,
    area: usize = dependentAttrDefault, // default values help show dependent
attributes
};

const calcArea = struct {

```

```

fn calcArea(
    tuple: RelTuple(Heading3),
    comptime attr_id: RelTuple(Heading3).AttributeId,
) RelTuple(Heading3) {
    const area = tuple.extract(.length) * tuple.extract(.width);
    return tuple.update(attr_id, area);
}
}.calcArea;

const Class_3 = RelClass(
    .class_3, // class tag
    Heading3, // heading
    &.{ // identifier
        &{.id},
    },
    &{ // dependent attribute properties
        .{ .area, calcArea },
    },
    &{}, // class options
    20, // max capacity
    &{}, // initial population
);
const class_3 = Class_3.init(3);
try testing.expectEqual(@as(usize, 1), Class_3.dependentAttrs.len);
try testing.expectEqual(.id, Class_3.generatedAttr);
try testing.expectEqual(@as(usize, 20), Class_3.ClassVar.relvar_capacity);
const inst_1 = class_3.create(.{
    .length = 20,
    .width = 40,
    .weight = 17.5,
});
const inst_2 = class_3.create(.{
    .length = 30,
    .width = 50,
    .weight = 27.5,
    .area = dependentAttrDefault,
});
try testing.expectEqual(@as(usize, 2), class_3.cardinality());
try testing.expectEqual(@as(usize, 800), inst_1.readAttribute(.area));
try testing.expectEqual(@as(usize, 1500), inst_2.readAttribute(.area));

// Create new instances by modifying the relational class instances.
relvalue.relExprBegin(); defer relvalue.relExprEnd();

// Update the width attribute using a relation variable.
const class3_value = class_3.selectAll().relValueOf();
const new_value = try class3_value.updateAllAttributes(.width, 60);
_ = class_3.updateOrInsertFromRelValue(new_value);
const width_60 = class_3.selectEqual(.width, 60);
try testing.expectEqual(@as(usize, 2), width_60.cardinality());

// Update the width attribute directly.

```

```

class_3.selectAll().updateAttribute(.width, 80);
const width_80 = class_3.selectEqual(.width, 80);
try testing.expectEqual(@as(usize, 2), width_80.cardinality());
}

```

Class Instance Introspection

The usual introspection functions are provided.

```

<<class public functions>>=
pub fn cardinality(
    _: Self,
) usize {
    return classvar.cardinality();
}

```

```

<<class public functions>>=
pub fn degree(
    _: Self,
) usize {
    return classvar.degree();
}

```

Class Instance Sets

In this design, Relation Variables are held entirely in memory. We would like to take some implementation advantage of that arrangement. Most frequently, we are interested only in a subset of the tuples held in a relation variable. One way to accomplish that is simply to hand around a Relation Value that contains the tuples of interest. But that representation is not convenient when we deal with relationships between relation variables. Relationships are discussed in the next section. Because all relation variables are in memory, we need not be bound to strictly value-based schemes. What is convenient is to have some form of “reference” to the tuples in a relation variable. Since the tuples of a relation variable are contained in an array, then the array index (along with knowing which relation variable we are dealing with) serves as an identifier for the tuple. It is convenient to keep relation variable tuple references as a pointer to a relation variable and a bit set. You can think of this as a “multi-pointer” where a raw pointer refers to the relation variable and the bit set indicates the subset of tuples being referenced. The fundamental operations on a relation variable return a subset of the tuples as a `TupleRefSet`. By associating a `TupleRefSet` bit-set with a pointer to a relation variable, we have a mechanism to manipulate class instances by reference.

In this section, we show the details of instance subsets.

```

<<class declarations>>=
/// A reference to a set of Relation Variable tuples.

```

```

pub const InstanceSet = struct {
    classvar: Self = {},
    tuple_ref_set: TupleRefSet,

    <<instance set public functions>>
};


```

Initialization of Instance Sets

There are many ways to initialize an instance set.

```

<<instance set public functions>>=
pub fn initEmpty() InstanceSet {
    return .{
        .tuple_ref_set = .initEmpty(),
    };
}


```

```

<<instance set public functions>>=
pub fn initFromTupleRefSet(
    tuple_set: TupleRefSet,
) InstanceSet {
    return .{
        .tuple_ref_set = tuple_set,
    };
}


```

```

<<instance set public functions>>=
pub fn initFromInstanceRef(
    inst_ref: InstanceRef,
) InstanceSet {
    var tuple_set = emptyTupleRefSet;
    tuple_set.set(inst_ref.tuple_ref.tuple_id);
    return .{
        .tuple_ref_set = tuple_set,
    };
}


```

```

<<instance set public functions>>=
pub fn instanceRefFromInstanceSet(
    self: InstanceSet,
) InstanceRef {
    assert(self.cardinality() == 1);

    const inst_index = self.tuple_ref_set.findFirstSet()?.?;
    return InstanceRef.initFromIndex(inst_index);
}


```

```

<<instance set public functions>>=
pub fn tupleFromInstanceSet(
    self: InstanceSet,
) Tuple {
    assert(self.cardinality() == 1);

    return self.instanceRefFromInstanceSet().dereference();
}

```

Cardinality

```

<<instance set public functions>>=
/// The `cardinality` function returns the number of tuple_ref_set referenced
/// by the given Instance Set.
pub fn cardinality(
    self: InstanceSet,
) usize {
    return self.tuple_ref_set.count();
}

```

Empty

```

<<instance set public functions>>=
/// The `empty` function returns the true if the given Instance Set references
/// no tuple_ref_set.
pub fn empty(
    self: InstanceSet,
) bool {
    return self.cardinality() == 0;
}

```

Not Empty

```

<<instance set public functions>>=
/// The `notEmpty` function returns the true if the given Instance Set references
/// at least one tuple.
pub fn notEmpty(
    self: InstanceSet,
) bool {
    return !self.empty();
}

```

Complement

```
<<instance set public functions>>=
/// The `complement` function returns the instance set that is the complement
/// of the given one.
pub fn complement(
    self: InstanceSet,
) InstanceSet {
    var compl = self.tuple_ref_set.complement();
    compl.setIntersection(@TypeOf(classvar).universe);
    return .initFromTupleRefSet(compl);
}
```

Limit

```
<<instance set public functions>>=
/// The `limit` function returns the instance set that contains at most
/// `limit` number of tuple_ref_set.
pub fn limit(
    self: InstanceSet,
    count: RelationCapacity,
) InstanceSet {
    var result: InstanceSet = .initEmpty();
    var inst_iter = self.iterator();
    var tuple_count = count;
    while (tuple_count != 0) : (tuple_count -= 1) {
        if (inst_iter.next()) |inst_ref| {
            result.tuple_ref_set.set(inst_ref.tuple_ref.tuple_id);
        } else break;
    }

    return result;
}
```

Destroy

Tuples may be deleted from a Relation Variable. The storage management design is such that the empty slot can be reused.

```
<<instance set public functions>>=
/// The `destroy` function deletes all the tuple_ref_set in the given instance set.
pub fn destroy(
    self: InstanceSet,
) void {
    var inst_iter = self.iterator();
```

```

while (inst_iter.next()) |inst_ref| {
    inst_ref.destroy();
}
}

```

Union

```

<<instance set public functions>>=
/// The `unionInstanceSet` function performs the set union of the given instance
/// with the instance set given by the `other` argument.
pub fn unionInstanceSet(
    self: InstanceSet,
    other: InstanceSet,
) InstanceSet {
    return .initFromTupleRefSet(self.tuple_ref_set.unionWith(other.tuple_ref_set));
}

pub inline fn @"union"(

    self: InstanceSet,
    other: InstanceSet,
) InstanceSet {
    return self.unionInstanceSet(other);
}

```

Intersect

```

<<instance set public functions>>=
/// The `intersect` function performs the set intersection of the given instance
/// with the instance set given by the `other` argument.
pub fn intersect(
    self: InstanceSet,
    other: InstanceSet,
) InstanceSet {
    return .initFromTupleRefSet(self.tuple_ref_set.intersectWith(other.tuple_ref_set));
}

```

Minus

```

<<instance set public functions>>=
/// The `minus` function performs the set subtraction of the `subtrahend`
/// instances set from the `minuend` instance set.
pub fn minus(
    minuend: InstanceSet,
    subtrahend: InstanceSet,
)

```

```
) InstanceSet {
    return .initFromTupleRefSet(minuend.tuple_ref_set.differenceWith(subtrahend
    .tuple_ref_set));
}
```

Eql

```
<<instance set public functions>>=
/// The `eql` function returns `true` if the given instance set is equal to
/// the `other` instance set and `false` otherwise.
pub fn eql(
    one: InstanceSet,
    other: InstanceSet,
) bool {
    return one.tuple_ref_set.eql(other.tuple_ref_set);
}
```

Iterator

```
<<instance set public functions>>=
pub fn iterator(
    self: InstanceSet,
) Iterator() {
    return .{
        .bits = self.tuple_ref_set.tset,
    };
}

pub fn Iterator() type {
    return struct {
        const IterSelf = @This();

        bits: @FieldType(@FieldType(InstanceSet, "tuple_ref_set"), "tset"),

        pub fn next(
            self: *IterSelf,
        ) ?InstanceRef {
            if (self.bits.findFirstSet() |index| {
                self.bits.unset(index);
                return Self.InstanceRef.initFromIndex(index);
            } else {
                return null;
            }
        }
    };
}
```

Traverse

```
<<instance set public functions>>=
pub fn traverse(
    inst_set: InstanceSet,
    comptime relationship: anytype,
    comptime traversal_phrase: anytype,
    comptime dest_class: anytype,
) @TypeOf(relationship).InferDestRelClass(traversal_phrase, dest_class).InstanceSet {
    return if (comptime association.isAssociationType(@TypeOf(relationship)))
        inst_set.traverseAssociation(relationship, traversal_phrase, dest_class)
    else if (comptime generalization.isGeneralizationType(@TypeOf(relationship)))
        inst_set.traverseGeneralization(relationship, traversal_phrase, dest_class)
    else
        @compileError("relationship argument is neither an association nor
generalization");
}
```

Signal

```
<<instance set public functions>>=
pub fn signal(
    inst_set: InstanceSet,
    target: anytype,
    comptime event_id: @Type(.enum_literal),
    event_payload: anytype, // tuple of additional activity arguments
) void {
    var iter = inst_set.iterator();
    while (iter.next() |inst_ref| {
        inst_ref.signal(target, event_id, event_payload);
    }
}
```

Contains

```
<<instance set public functions>>=
/// The `contains` function returns `true` if `one` instance set is
/// a superset of the `other` instance set and `false` otherwise.
pub fn contains(
    one_set: InstanceSet,
    other_ref: InstanceRef,
) bool {
    return one_set.tuple_ref_set.isSet(other_ref.tuple_ref.tuple_id);
}
```

Select

```
<<instance set public functions>>=
pub fn restrict(
    self: InstanceSet,
    selector: *const fn(tuple: Tuple, args: anytype) bool,
    selector_args: anytype,
) InstanceSet {
    var tuple_set = emptyTupleRefSet;

    var set_iter = self.iterator();
    while (set_iter.next()) |inst_ref| {
        const tuple = inst_ref.dereference();
        if (selector(tuple, selector_args)) {
            tuple_set.set(inst_ref.tuple_ref.tuple_id);
        }
    }

    return .initFromTupleRefSet(tuple_set);
}
```

Select Equal

```
<<instance set public functions>>=
pub fn restrictEqual(
    self: InstanceSet,
    comptime attr_id: AttributeId,
    attr_value: AttributeType(attr_id),
) InstanceSet {
    return self.restrict(relvalue.equalitySelector, .{ attr_id, attr_value });
}
```

Tests

```
<<class module tests>>=
test "destroy tests" {
    const Heading = struct {
        name: []const u8,
        length: usize,
        weight: f32,
    };

    const Class_1 = RelClass(.class_1, Heading, &.{ &{.name} }, &{}, .{}, 10, &{});
    const class_1 = Class_1.init(1);

    const inst_1 = class_1.create(.{
        .name = "foo",
        .length = 42,
```

```

        .weight = 13.7,
    });
    const inst_2 = class_1.create({
        .name = "bar",
        .length = 48,
        .weight = 17.7,
    });

    try testing.expectEqual(@as(usize, 2), class_1.cardinality());

    inst_1.destroy();
    try testing.expectEqual(@as(usize, 1), class_1.cardinality());

    inst_2.destroy();
    try testing.expectEqual(@as(usize, 0), class_1.cardinality());
}

```

Selecting Class Instances

There are two primary ways to obtain a set of class instances:

1. Selection based on the value of tuple attributes in the class relvar.
2. Selection based on traversing a relationship.

In later sections on associations and generalizations, we show how a traversing relationship results in selecting instances of a class. Here, functions to select class instance subsets based on attribute values are shown.

There are two categories of selection based upon whether the attribute values supplied constitute an identifier. Recall that each identifier has a hash table associated with it, so selections based on identifying attribute values is $O(1)$ in computation. Other forms of selection based on non-identifying attributes is accomplished by sequential comparison and is $O(N)$ in the worst case.

```

<<class public functions>>
/// The `selectAll` function returns an `InstanceSet` that references all the
/// tuples currently in the given class.
pub fn selectAll(
    _: Self,
) InstanceSet {
    return .initFromTupleRefSet(ClassVar.universe);
}

```

```

<<class public functions>>
/// The `selectByIdentifier` function returns an `InstanceSet`
/// whose cardinality is at most one. If the cardinality is one, the
/// `InstanceSet` references the tuple in the relation variable whose
/// attributes and values, as given by the `av_pairs` argument match

```

```

/// the identifier given by the `ident_num`. The `ident_num` is a
/// small integer value that gives the identifier on which the search
/// is based. The `id_attr_values` argument is a anonymous `struct`
/// literal containing the attributes and values that make up the
/// identifiers specified by `ident_num`. If no tuple in the relation
/// variable has attribute values that match the identifying attributes,
/// then the cardinality of the returned `InstanceSet` value is zero.
pub fn selectByIdentifier(
    _: Self,
    comptime ident_num: usize,
    id_attr_values: IdentifierAttributes(ident_num),
) RelVarError!InstanceRef {
    return .{
        .tuple_ref = try classvar.selectByIdentifier(ident_num, id_attr_values),
    };
}

```

```

<<class public functions>>=
pub fn selectInitialInstance(
    _: Self,
    inst_tag: InitialInstanceId,
) InstanceRef {
    return .{
        .tuple_ref = classvar.selectInitialInstance(inst_tag),
    };
}

```

```

<<class public functions>>=
/// The `select` function returns an `InstanceSet` value that contains a
/// subset of the tuples from the given class. The tuples included are
/// determined by the function pointed to by `selector`. The `selector`
/// function is invoked for each tuple in the class and is passed the
/// `selector_args` argument. If the return value from an invocation
/// of `selector` is `true`, then the argument tuple to `selector` is
/// included in the returned `InstanceSet`.
pub fn select(
    _: Self,
    selector: *const fn(tuple: Tuple, args: anytype) bool,
    selector_args: anytype,
) InstanceSet {
    return .initFromTupleRefSet(classvar.select(selector, selector_args));
}

```

The `selector` functions for Relation Variables have the same interface as those for classes. A set of function are provided to supply the syntactic sugar for reusing the selector functions.

```

<<class public functions>>=
/// The `selectEqual` function returns a `InstanceSet` value that
/// references those tuples where the value of the `attr_id` attribute
/// is equal to the value of the `attr_value` argument.
pub fn selectEqual(
    _: Self,
    comptime attr_id: AttributeId,
    attr_value: AttributeType(attr_id),
) InstanceSet {
    return .initFromTupleRefSet(classvar.selectEqual(attr_id, attr_value));
}

```

```

<<class public functions>>=
/// The `selectAllEqual` function returns an `InstanceSet` value that
/// references those tuples where all of the attribute id / attribute
/// value pairs of the `av_pairs` argument are true. The `av_pairs`
/// argument is assumed to be a list of attribute id / value pairs.
pub fn selectAllEqual(
    _: Self,
    av_pairs: anytype,
) InstanceSet {
    return .initFromTupleRefSet(
        classvar.select(relvalue.conjunctiveEqualitySelector, av_pairs),
    );
}

```

```

<<class public functions>>=
/// The `selectAnyEqual` function returns a `InstanceSet` value that
/// references those tuples where at least one of the attribute id /
/// attribute value pairs of the `av_pairs` argument is true. The
/// `av_pairs` argument is assumed to be a list of attribute id /
/// value pairs.
pub fn selectAnyEqual(
    _: Self,
    av_pairs: anytype,
) InstanceSet {
    return .initFromTupleRefSet(
        classvar.select(relvalue.disjunctiveEqualitySelector, av_pairs),
    );
}

```

Following the same pattern as with equality selector functions, we provide simpler interfaces which make using the comparative selector function convenient.

```

<<class public functions>>=
/// The `selectComparison` function returns a `InstanceSet` value that
/// references those tuples where the value of the attribute given by
/// `attr_id` when compared to the `attr_value` by the `op` operation
/// is true.
pub fn selectComparison(
    _: Self,
    comptime attr_id: classvar.AttributeId,
    op: math.CompareOperator,
    attr_value: AttributeType(attr_id),
) InstanceSet {
    return .initFromTupleRefSet(
        classvar.select(
            relvalue.comparativeSelector,
            .{ attr_id, op, attr_value },
        ),
    );
}

```

```

<<class public functions>>=
/// The `selectAllComparison` function returns a `InstanceSet` value
/// that references those tuples where the logical *AND* of all the
/// comparisons given by `exprs` are true. The `exprs` argument is
/// taken as a list of triples. Each triple is as described for the
/// `comparativeSelector` function.
pub fn selectAllComparison(
    _: Self,
    exprs: anytype,
) InstanceSet {
    return .initFromTupleRefSet(
        classvar.select(relvalue.conjunctiveComparativeSelector, exprs),
    );
}

```

```

<<class public functions>>=
/// The `selectAnyComparison` function returns a `InstanceSet` value
/// that references those tuples where the logical *OR* of all the
/// comparisons given by `exprs` are true. The `exprs` argument is
/// taken as a list of triples. Each triple is as described for the
/// `comparativeSelector` function.
pub fn selectAnyComparison(
    _: Self,
    exprs: anytype,
) InstanceSet {
    return .initFromTupleRefSet(
        classvar.select(relvalue.disjunctiveComparativeSelector, exprs),
    );
}

```

```
);  
}
```

Tests

```
<<class module tests>>=  
test "selection tests" {  
    testing.log_level = .info;  
  
    const Heading = struct {  
        id: ClassGeneratedId = generated_id_default,  
        name: []const u8,  
        length: usize,  
        weight: f32,  
    };  
  
    const Rvar_1 = RelClass(.rvar_1, Heading, &{ &{.id} }, &{}, .{}, 10, &{});  
    const rvar_1 = Rvar_1.init(1);  
  
    const pop = rvar_1.createSet(&.{  
        .{ .name = "foo", .length = 42, .weight = 13.7 },  
        .{ .name = "bar", .length = 43, .weight = 13.8 },  
        .{ .name = "baz", .length = 44, .weight = 13.9 },  
        .{ .name = "fuzz", .length = 45, .weight = 14.0 },  
    });  
    defer pop.destroy();  
  
    const bar_inst = rvar_1.selectEqual(.name, "bar");  
    try testing.expectEqual(@as(usize, 1), bar_inst.cardinality());  
  
    const foo_baz_insts = rvar_1.selectAnyEqual(&.{  
        .{ .name, "foo" },  
        .{ .name, "baz" },  
    });  
    try testing.expectEqual(@as(usize, 2), foo_baz_insts.cardinality());  
  
    const lengths = rvar_1.selectAllComparison(&.{  
        .{ .length, .gt, 42 },  
        .{ .length, .lt, 45 },  
    });  
    try testing.expectEqual(@as(usize, 2), lengths.cardinality());  
}
```

Dereferencing Instance Sets

Computation on tuple subsets of a Relation Variable is done by extracting the subset of class instances into a Relation Value and then applying the relation variable algebra to compute a result. The complication here arises from dependent attributes. Converting an InstanceSet into a relation variable implies that the dependent attribute formulas must be evaluated also.

```

<<instance set public functions>>=
/// The `relValueOf` returns a `RelValue` value containing those tuples
/// included in the `inst_set` argument.
pub fn relValueOf(
    self: InstanceSet,
) RelValueType {
    var builder = RelValueType.Builder.init(self.tuple_ref_set.count());

    var set_iter = self.tuple_ref_set.iterator({});

    while (set_iter.next() |tuple_ref| {
        var tuple = tuple_ref.dereference();
        inline for (dependentAttrs) |*dep_attr_prop| {
            const attr_id = dep_attr_prop.attr_id;
            const formula = dep_attr_prop.formula;
            tuple = formula(tuple, attr_id);
        }
        const inserted = builder.insert(tuple);
        assert(inserted);
    }

    return builder.finalize();
}

```

Tests

```

<<class module tests>>=
test "relValueOf tests" {
    relvalue.relExprBegin(); defer relvalue.relExprEnd();

    const Heading = struct {
        id: ClassGeneratedId = generated_id_default,
        name: []const u8,
        length: usize,
        weight: f32,
    };

    const Rvar_1 = RelClass(.rvar_1, Heading, &{ &{.id} }, &{}, {}, 10, &{});
    const rvar_1 = Rvar_1.init(1);

    const inst_1 = rvar_1.createSet(&{
        .{ .name = "foo", .length = 42, .weight = 13.7 },
        .{ .name = "bar", .length = 43, .weight = 13.8 },
        .{ .name = "baz", .length = 44, .weight = 13.9 },
        .{ .name = "fuzz", .length = 45, .weight = 14.0 },
    });
    try testing.expectEqual(@as(usize, 4), inst_1.cardinality());
    try testing.expectEqual(@as(usize, 4), rvar_1.cardinality());
}

```

```

const inst_value = inst_1.relValueOf();
try testing.expectEqual(@as(usize, 4), inst_value.cardinality());
try testing.expectEqual(@as(usize, 42 + 43 + 44 + 45), inst_value.sum(.length));

// Obtain the relation value when there are dependent attributes
const Heading2 = struct {
    id: ClassGeneratedId = generated_id_default,
    length: usize,
    width: usize,
    area: usize = dependent_attr_default,
};

const calcArea = struct {
    fn calcArea(
        tuple: RelTuple(Heading2),
        comptime attr_id: RelTuple(Heading2).AttributeId,
    ) RelTuple(Heading2) {
        const area = tuple.extract(.length) * tuple.extract(.width);
        return tuple.update(attr_id, area);
    }
}.calcArea;

const Rvar_2 = RelClass(
    .rvar_2, // class tag
    Heading2, // Heading
    &.{ // identifier
        &{ .id }
    },
    &.{ // dependent attribute specs
        .{.area, calcArea}
    },
    .{},
    20,
    &.{ // attribute values
        .{ .l10, .{ .length = 10, .width = 10 } },
        .{ .l20, .{ .length = 20, .width = 20 } },
        .{ .l30, .{ .length = 30, .width = 30 } },
    },
);
const rvar_2 = Rvar_2.init(2);
const inst_2 = rvar_2.selectAll();
const inst_2_value = inst_2.relValueOf();
try testing.expectEqual(@as(usize, 3), inst_2_value.cardinality());
try testing.expectEqual(@as(usize, 100 + 400 + 900), inst_2_value.sum(.area));
}

```

Attribute Access

There are several common use cases where it is convenient to have access to read and update attributes of a relation variable directly.

```

<<instance reference declarations>>=
/// The `readAttribute` attribute function returns the value of the
/// attribute given by `attr_id` from the tuple identified by `inst_set`.
/// The cardinality of the tuple set given by `inst_set` must be one.
pub fn readAttribute(
    inst_ref: InstanceRef,
    comptime attr_id: AttributeId,
) AttributeType(attr_id) {
    const attr_value = if (isDependentAttr(attr_id)) |attr_formula| afblk: {
        const tuple = inst_ref.dereference();
        const dependent = attr_formula(tuple, attr_id);
        break :afblk dependent.extract(attr_id);
    } else inst_ref.dereference().extract(attr_id);
    trace_operations.read_attr(
        inst_ref,
        @intFromEnum(attr_id),
        @ptrCast(&attr_value),
    );
    return attr_value;
}

```

```

<<instance set public functions>>=
/// The `readAttribute` attribute function returns the value of the
/// attribute given by `attr_id` from the tuple identified by `inst_set`.
/// The cardinality of the tuple set given by `inst_set` must be one.
pub fn readAttribute(
    self: InstanceSet,
    comptime attr_id: AttributeId,
) AttributeType(attr_id) {
    const attr_value = if (isDependentAttr(attr_id)) |attr_formula| afblk: {
        if (self.tuple_ref_set.findFirstSet()) |element_index| {
            const tuple = InstanceRef.initFromIndex(element_index).dereference();
            const dependent = attr_formula(tuple, attr_id);
            break :afblk dependent.extract(attr_id);
        } else {
            panic("attempt to read attribute of an empty instance set", .{});
        }
    } else self.tuple_ref_set.readAttribute(attr_id);
    trace_operations.read_attr(
        self.instanceRefFromInstanceSet(),
        @intFromEnum(attr_id),
        @ptrCast(&attr_value),
    );
    return attr_value;
}

```

Likewise, updating a single attribute is a common use case.

```

<<instance reference declarations>>=
/// The `updateAttribute` function updates the attribute given by `attr_id`
/// to the value given by `value` for the tuple referenced by `inst_ref`.
/// The update changes the attribute value in the relation variable _in situ_.
pub fn updateAttribute(
    inst_ref: InstanceRef,
    comptime attr_id: AttributeId,
    value: AttributeType(attr_id),
) void {
    if (isDependentAttr(attr_id) != null) {
        @compileError("attribute, '" ++ @tagName(attr_id) ++
            "' is a dependent attribute and cannot be updated");
    }
    inst_ref.tuple_ref.updateAttribute(attr_id, value);
}

```

```

<<instance set public functions>>=
/// The `updateAttributeSet` function sets the attribute given by `attr_id`
/// to the value given by `value` for all tuples in `inst_set`.
/// The update changes the attribute value of the all the tuples that
/// are in the relation variable _in situ_.
pub fn updateAttribute(
    self: InstanceSet,
    comptime attr_id: AttributeId,
    value: AttributeType(attr_id),
) void {
    if (isDependentAttr(attr_id) != null) {
        @compileError("attribute, '" ++ @tagName(attr_id) ++
            "' is a dependent attribute and cannot be updated");
    }
    self.tuple_ref_set.updateAttribute(attr_id, value);
    var set_iter = self.iterator();
    while (set_iter.next() |inst_ref| {
        trace_operations.updated_attr(
            inst_ref,
            @intFromEnum(attr_id),
            @ptrCast(&value),
        );
    }
}

```

Tests

```

<<class module tests>>=
test "attribute access tests" {
    const Heading = struct {

```

```

        name: []const u8,
        length: usize,
        weight: f32,
    };

    const Rvar_1 = RelClass(.rvar_1, Heading, &{
        &{ .name, .length },
    }, &{}, {}, 10, &{});
    const rvar_1 = Rvar_1.init(1);

    const pop = rvar_1.createSet(&{
        .{ .name = "foo", .length = 42, .weight = 13.7 },
        .{ .name = "bar", .length = 43, .weight = 13.8 },
        .{ .name = "baz", .length = 44, .weight = 13.9 },
        .{ .name = "fuzz", .length = 45, .weight = 14.0 },
    });
    defer pop.destroy();

    const bar_inst = try rvar_1.selectByIdentifier(0, .{
        .name = "bar",
        .length = 43,
    });
    try testing.expectEqual(@as(f32, 13.8), bar_inst.readAttribute(.weight));

    const baz_inst = try rvar_1.selectByIdentifier(0, .{
        .name = "baz",
        .length = 44,
    });
    baz_inst.updateAttribute(.weight, 15.0);
    try testing.expectEqual(@as(f32, 15.0), baz_inst.readAttribute(.weight));
}

```

Domain Interactions

```

<<class module declarations>>=
const domain_mod = @import("domain.zig");
const class_trace_point = domain_mod.class_trace_point;
const DomainExecution = domain_mod.DomainExecution;

```

Code layout

```

<<class.zig>>=
///! The `class.zig` module implements the concept of a *Relational Class*.
///! A Relational Class is based on a Relational Variable and is the
///! primary means for managing application data. A Relational Class includes
///! the ability to have class generated identifying attribute values and
///! to define attributes as _mathematically dependent_ by supplying a
///! formula used to calculate the attribute value.
<<edit warning>>
<<copyright info>>

```

```

const std = @import("std");
const builtin = @import("builtin");
const testing = if (builtin.os.tag == .freestanding)
    @import("resee_testing")
else
    std.testing;
const meta = std.meta;
const mem = std.mem;
const enums = std.enums;
const fmt = std.format;
const math = std.math;

const association = @import("association.zig");
const generalization = @import("generalization.zig");

const assert = std.debug.assert;
const panic = std.debug.panic;

pub const version = "1.0.0-a1";

<<class module declarations>>
<<class module public functions>>
<<class module private functions>>
<<class module tests>>

```

I Want to Hold Your Hand

In the last section, we showed the application of Relation Variables store and manage application data by creating a Relational Class directly based a Relation Variable. In this section, we show the application of binary relations, as implemented by an adjacency matrix, to manage Association relationships between Relational Classes.

In this context, an Association represents a semantic rule between two Relational Classes. They model real-world associations that are found in the subject matter domain. The semantics of the Association is carried by the verb phrases used to annotate the class diagram of a domain.

Association Reification

Formally, an association is a binary relation between two sets. It is fundamentally based on the concept of a subset of the extended Cartesian product of the participating sets. In terms of our execution model, all Associations are reified by using a Relational Class of a specific form. Such classes are called *Associative Classes* since their role is to carry the data required to map between instances of the participating Relational Classes. The Associative Class holds the binary relation mapping as *referential attributes* whose *values* are equal to the corresponding *identifying attributes* of the related classes. Our model of an Association is value based in the same manner as described for the relational model of data. This was shown graphically in [Figure 12.3](#) when we discussed associations as part of the execution model that is implemented by ReSEE. That figure is reproduced here for convenience.

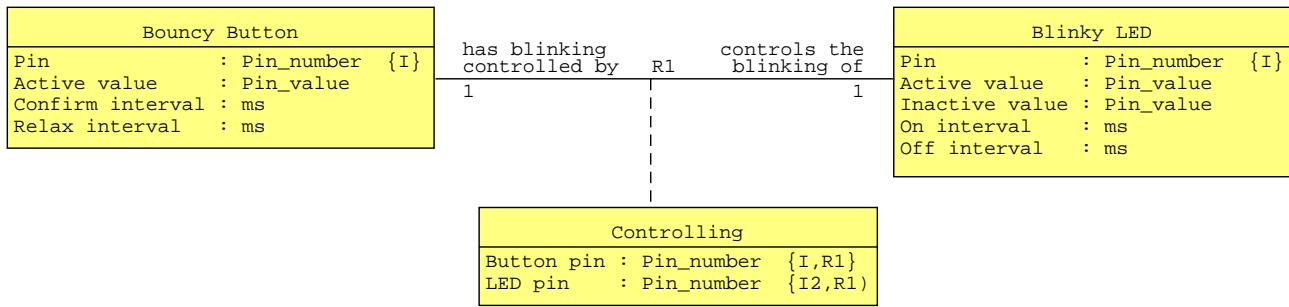


Figure 36. Relationship for LED Blinking Control

This usage gives the model its implementation independence since we are only relying on the fundamental idea of attribute value equality.

Model vs. Implementation

As described in the chapter on Data Management Fundamentals, the implementation in the ReSEE environment of the **Controlling** class from the previous figure is by means of an association adjacency matrix. The association adjacency matrix implementation is capable of holding the mapping of participating instances for any association regardless of the multiplicity and conditionality^[2].

For simple Associative Classes whose attributes are used only to reify the mapping implied by the association, the associative class is not present in the ReSEE implementation. The Associative Class found in the model is simply discarded since its role is fulfilled by the adjacency matrix implementation.

A Special Case

Association relationships frequently have the following particular properties:

1. The association is a **total function**.
2. There are no attributes that are descriptive of the association itself.
3. The associative class does not have any relationship to classes other than the participants in the association.
4. The associative class exhibits no life cycle behavior.

Despite this being a “special” case, it is quite common. Again, we showed that case in the previous discussion and reproduce the figure below.

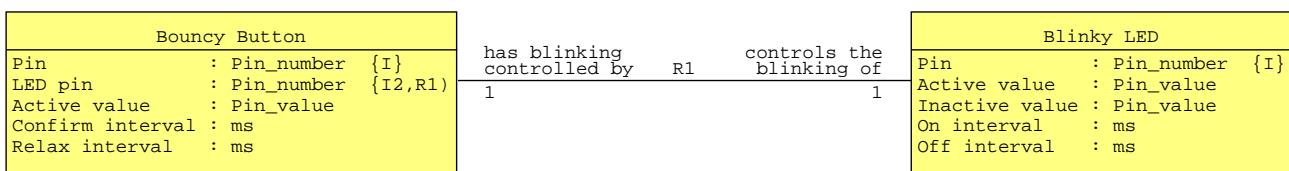


Figure 37. Simplified Relationship for LED Blinking Control

Since for a function, the number of instances of an Associative Class is the same as the number of instances of the participating class with the greater multiplicity, it is possible, in the modeling notation, to simply include the necessary referential attribute(s) in the participating class. In the

previous figure, the **Controlling** class has been elided and the LED pin attribute was moved to the **Bouncy Button** class^[3]. Models take advantage of this special case since it occurs frequently, reduces the number classes that must tracked, and simplifies the class diagram graphic.

This case is also easily conflated with the RDMS notion of “foreign keys”. A RDMS foreign key that is not NULL^[4] can manage the referential consistency of associations that are total functions.

Even in this special case, when the model of the Association is mapped onto ReSEE, the adjacency matrix handles the binary relation implementation properly and the referential attribute (LED pin in this case) is elided in the ReSEE implementation.

Associative Class Properties

Outside of the special case just discussed, our model of an Association requires an Associative Class for its reification. This is a direct consequence of the Association requiring one of the above properties that would otherwise allow us to use the special case. These properties are further explained as follows.

1. Two types of associations arise where the number of instances of the association mapping does not equal the number of instances of one of its participants. + The first type is based on multiplicity. If a relationship is many-to-many, either conditional or not, there can be more instances of the association instance mapping than could be held by referential attributes in either association participant class. Such binary relations are *set-valued functions* or *set-valued partial functions* and the associative class is required to track the set membership. + The second type is based on conditionality. If a relationship has conditionality such that it is not a total function, e.g. A *--? B, then the binary relation is a partial function and the number of instances of the association mapping is, in general, less than that of either participant. Since we do not allow NULL attribute values, there is no way to represent the partial function aspect of the association using the attributes of one of the participants.
2. It is not uncommon for domain requirements to require information about an instance of an association itself be recorded. For example, the time at which an aircraft lands on a runway is neither an attribute of the aircraft nor an attribute of the runway. Instead, it is a property of an instance of the association between the aircraft and the runway that is formed when the aircraft has touched ground.
3. Since an Associative Class is a “real” Relational Class it can participate in other associations. Having associative class instances related to other class instances can provide a powerful means to encode logical constraints about the domain’s subject matter.
4. Some Associative Classes represent ephemeral circumstances and have a life cycle of their own. As we say in the chapter on Execution Fundamentals, a state model may be attached to a Relational Class to model its life cycle. This is also true of an Associative Class. Consider a communications connection between two end points. The connection itself usually has a life cycle. Connections must formed, exist for some time, and are then taken down and discarded. This describes the life cycle of the association between end points and the end points must cooperate with the connection association when it is in the various phases of its life cycle.

An Associative Class is a first-class Relational Class. It is easy to slip into thought patterns that treat Associative Classes as some type of lesser construct, used mainly for bookkeeping purposes. But the set of propositions held in an Associative Class are valid truths just as are those held in an what might be considered an ordinary class. However, one difference is how the identifiers for the association class are determined. Since an associative class must have referential attributes to handle the binary relation mapping it models, those referential attributes can also serve as identifying attributes, i.e. the attribute plays two roles in the Associative Class. The Associative Class essentially “borrows” the set identification scheme of its participant classes. The use of referential attributes as identifying attributes is a happy circumstance for Associative Classes. At least one and sometimes two identifiers are strictly determined by the multiplicity and

conditionality of the association and using referential attributes in the role of identifying attributes. Of course, additional identifiers may be needed if the requirements, as captured by the model, dictate.

Those associations which satisfy the special case, *i.e.* *do not* exhibit any of the properties just discussed are, from the ReSEE implementation perspective, *simple associations*. For such simple associations, an adjacency matrix is used to implement all Associative Class requirements that manage the mapping between the participating class instance sets and any associative class or referential attributes used to model the simple association are elided when implemented in ReSEE. For all other circumstances, the Association requires an Associative Class which is implemented as an ordinary RelClass. This implies that the Associative Class may have an optional state model if it has a life cycle.

Association Traversal

We use the term *traversal* for the operation that maps a subset of the instances of a class participating in an Association to a subset of the instances of the other participant in the Association. Traversal is based on the concept of *applying* the underlying binary relation to compute the image or preimage of the binary relation.

For simple associations, as we have just defined them, with two participants, the ReSEE implementation using a single adjacency matrix satisfies the needs for traversal. However, when an Associative Class is involved, the arrangement is no longer binary and the third element of the Associative Class must be taken into account. In the case of an Associative Class, traversal requires the ability to traverse not only between the two Association participants but also between either participant and the Associative Class. To meet these requirements, the Association is *decomposed* into two separate binary relations, one between one participant and the Associative Class and one between the other participant and the Associative Class.

The following figure helps to demonstrate how this happens. The figure shows the [C.J.Date Supplier / Parts database](#) expressed in the UML graphical notation we have been using.

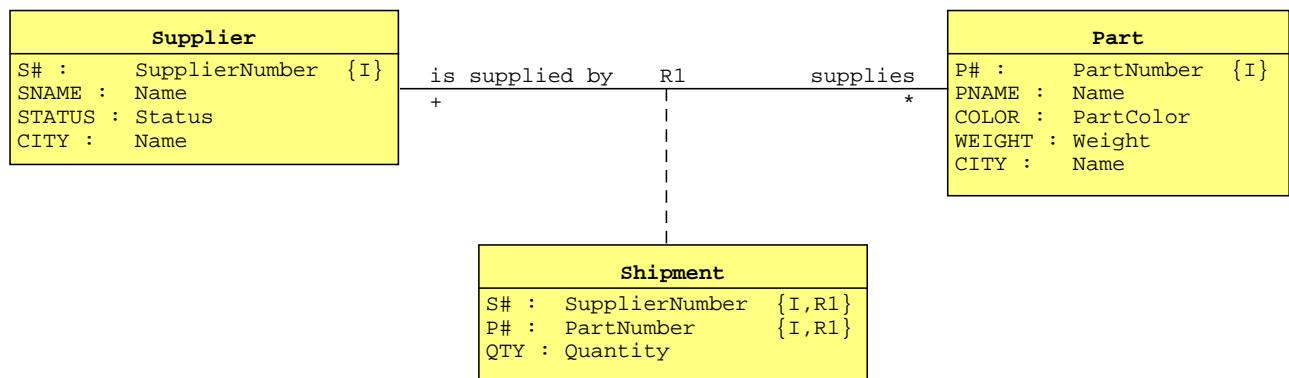


Figure 38. C.J. Date Supplier / Parts Database

The **R1** association captures the rules:

1. A **Supplier** supplies *any number of Parts*.
2. A **Part** is supplied by *at least one Supplier*.
3. The **Shipment** of each **Part** by a **Supplier** must record the quantity of **Parts** shipped.

Because the quantity of shipped parts is an attribute of the **Shipment** (and not an attribute of

either **Supplier** or **Part**), the **Shipment** Associative Class must be kept as part of the ReSEE implementation. To be able to traverse from either **Supplier** or **Part** to the Associative Class, **Shipment**, the association is decomposed into two simple associations.

The following figure shows the decomposition.

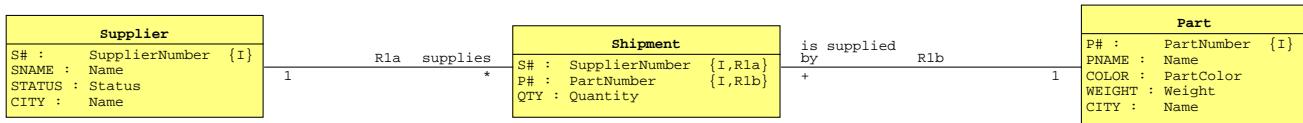


Figure 39. Supplier / Parts Association Decomposition

The result of the decomposition is two simple associations, **R1a** and **R1b**. Each of these decomposed associations is a total function since **Shipment** has an unconditional, exactly one association to both **Supplier** and **Part**. Note also that multiplicity and conditionality of **R1** traversing from **Supplier** to **Part** is now the multiplicity and conditionality of **R1a** when traversing from **Supplier** to **Shipment**. This is because the traversal of **R1b** from **Shipment** to **Part** is singular and unconditional. The conditionality of **R1** with respect to **Part** has been captured as the conditionality of **R1a** with respect to **Shipment** because any instance of **Shipment** is guaranteed to have a single related instance of **Part**. Essentially, the multiplicity and conditionality of **R1** have been “swapped” when viewed from the **Shipment** perspective. The same can be said about the multiplicity and conditionality of **R1b** between **Shipment** and **Part** for the same reasons. Readers are encouraged to fill out a few instances of each class in tabular form to convince themselves that the traversal outcome of **R1** is the same as the composition of traversing **R1a** and **R1b**. The significant difference is that the decomposition allows us to use only one “leg” of the association to traverse from either participant to the Associative Class.

Traversal, in our example from **Supplier** to **Part**, is achieved by applying the binary relation between **Supplier** and **Shipment** and then applying, to the first result, the binary relation between **Shipment** and **Part**. Note that the decomposition treats the Associative Class as the domain of both decomposed simple associations and the participants in the association take on the role of a codomain.

The traversal of the decomposed association from one participant to the other participant proceeds by first traversing from one participant to the Associative Class followed by a traversal from the Associative Class to the other participant. Since the Associative Class plays the role of domain for both adjacency matrices of the decomposition, traversing end-to-end follows the pattern of codomain to domain to codomain.

Summarizing, when an association is modeled using an associative class and that associative class *cannot* be elided because it has one of the properties discussed above (recording the shipped parts quantity in this example), it is then *not* a simple association from the ReSEE perspective and must be implemented in ReSEE by decomposing the association into two associations. In ReSEE terms, we call such associations, *compound*. The decomposition of a compound association yields two simple associations each of which is implemented by an association adjacency matrix.

Note that this difference between what is stated in the model and what is required by the implementation is due solely to the means by which the implementation is accomplished. The model view of Associations is consistent, treating all Associations the same with respect to the traversal between participants. For the model, traversal is defined in terms of equivalent values for referential attributes and identifying attributes. It is the mapping of that view onto the chosen implementation mechanism, namely adjacency matrices, that yields the additional complexity that distinguishes between simple and compound associations and the need to account for cases where it is necessary to decompose the modeled association to achieve the logic of the model. This is an archetypical example of Brooks' distinction between [essential and accidental complexity](#).

Reflexive Associations

An Association may be *reflexive*^[5] if the domain and codomain sets are the same. Reflexive associations require some practiced mental gymnastics to keep track in your head. There are two particular properties of reflexive associations to consider.

1. The conditionality of each side of the association must be the same. For example, there is no consistent population of instances for the association, **A ?--1 A**. Because of reflexivity, any conditionality will “poison” all the class instances. If the **A ?--1 A** association was modeled using a referential attribute in **A**, the it is not possible to specify an instance as *not* participating in the association, i.e. referential attribute would always have to have a value (remember there are no NULL values) which makes the association, **A 1—1 A**. The conditional version of this reflexive association is **A ?--? A**, which requires specifying an association class in the model.
2. The usual notion of using the class name for determining whether the image of the association or the preimage of the association is desired has an ambiguity since both participants in the association are the same sets. On a class diagram, the modeling convention of using distinct verb phrases at each end of the association graphic is used to disambiguate the intent of traversing the association.

The second point is where some mental discordance can arise. It is more convenient to talk about traversal when the classes of the association are distinct. The class names provide a clear directional path that the traversal follows. But it is the model annotation of verb phrases that is truly distinguishing. This gives rise to the notion that an association has an inherent *direction*. Interface ergonomics strongly suggest that multiple means of describing the traversal be available. For example, reflexive associations are forced to use the phrasing from the model annotation, but non-reflexive ones may use class names to specify the traversal. As we see below, both means are provided when specifying class traversal across associations.

Association Synchronization

Some associations represent competitive relationships and associations can manage the competitive nature related classes. Often competitive relationships are used to manage resource allocation. Usually there are fewer number of resources than there are clients requesting to use the resource. The resource is used temporarily and then reused by subsequent clients when the resource is returned. Further, the clients may makes requests at any time and resources may be returned at any time and in the face of any reasonable interpretation of concurrency, the assignment of a resource to a client must be done in a single operation. This implies that the Association must *synchronize* between the requests and the available resources to insure that the assignment is done in a predictable manner among the competing class instances and that the referential integrity constraints of the association are not violated. This role of an association is primarily based on the notion of synchronization of different branches of computation. We discuss this type of association behavior in a [later chapter](#), but, for now, it is sufficient to say that a state model, is attached to the Association itself, performs the required synchronization in a competitive association.

Representing an Association

We start with importing modules for a Relational Class and for an association adjacency matrix.

```
<<association module declarations>>=
const class = @import("class.zig");
const RelClass = class.RelClass;
const isRelClassType = class.isRelClassType;
const ClassGeneratedId = class.ClassGeneratedId;
```

```

const generated_id_default = class.generated_id_default;
const RelationCapacity = class.RelationCapacity;

const binary_relation = @import("binary_relation.zig");
const RefConstraint = binary_relation.RefConstraint;
const AssociationAdjMatrix = binary_relation.AssociationAdjMatrix;

const RelVarError = @import("relvar.zig").RelVarError;

```

From the previous discussion, there is substantial information that must be supplied to specify an Association between two participants. We collect that specification as a value of `AssociationSpec` type. To make the correspondence between the association and the information that must be supplied as the `AssociationSpec`, we adopt a mnemonic shorthand to write the association definition inline. Continuing with the C.J. Date example, we can state the definition of **R1** as:

Example 5. Mnemonic Shorthand for Specifying an Association

Supplier +-"is supplied by" -R1- "supplies"-* Part as Shipment

```

<<association module declarations>>=
/// An `AssociationSpec` is a `struct` that specifies the properties
/// of an Association. The order of the fields given below is
/// intended to be mnemonic of an inline description of the association.
/// For example,
/// Supplier +-"is supplied by" -R1- "supplies"-* Part as Shipment
/// is a short hand notation of the graphical representation of
/// the association. The ordering of the fields in `AssociationSpec`
/// follows the components of the inline notation.
///
/// * DomainRelClass -- the type of the Relational Class that serves as
///       the domain for the association. This must be a type as returned
///       by the `RelClass` type function.
/// * codomain_ref -- a `RefConstraint` value that determines the
///       multiplicity and conditionality of the associations from the
///       codomain perspective.
/// * codomain_phrase -- an `enum_literal` value that gives a distinct
///       phrase for the traversal of the association from the perspective
///       of traversing from the codomain to the domain.
/// * assoc_tag -- an `enum_literal` that gives a name to the association.
///       Each association should be given a unique value for assoc_tag.
/// * domain_phrase -- an `enum_literal` value that gives a distinct
///       phrase for the traversal of the association from the perspective
///       of traversing from the domain to the codomain.
/// * domain_ref -- a `RefConstraint` value that determines the
///       multiplicity and conditionality of the associations from the
///       domain perspective.
/// * CodomainRelClass -- the type of the Relational Class that serves as
///       the codomain for the association. This must be a type as returned
///       by the `RelClass` type function.
/// * AssociativeRelClass -- the type of the Relational Class that serves
///       the role of an Associative Class in the association. This type is

```

```

/// optional as not all associations have an Associative Class.
/// * assigner_state_model -- a optional pointer to a `StateModelSpec` value that designates a state model for the association which is used to synchronize competition between participants.
/// * PartitionClass -- the type of the Relational Class that serves to partition the competition between participant classes which is used to synchronize competition between subsets of participants.
/// The Partition Class is optional and must not be supplied if the `assigner_state_model` is not also supplied.
pub const AssociationSpec = struct {
    DomainRelClass: type,
    codomain_ref: RefConstraint,
    codomain_phrase: @Type(.enum_literal),
    assoc_tag: @Type(.enum_literal),
    domain_phrase: @Type(.enum_literal),
    domain_ref: RefConstraint,
    CodomainRelClass: type,
    AssociativeRelClass: ?type = null,
    assigner_state_model: ?*const StateModelSpec = null,
    PartitionClass: ?type = null,
};

```

The `assigner_state_model` and `PartitionClass` arguments pertain to optional behavioral aspects of an association. This is discussed in the [next chapter](#).

We follow our usual pattern of using a `comptime` type function to define the data type for a particular association.

```

<<association module public functions>>=
/// The `Association` function is a type function that returns a data type
/// suitable for use to declare a variable that is an association
/// relationship. The properties of the resulting `Association` are given
/// by the `association_spec` argument. An initial set of associations
/// may be given the `initial_assocs` argument. This argument is
/// assumed to be one of two configurations.
///
/// For simple associations, i.e. those who do not supply a value for
/// the `AssociativeRelClass` field in the `association_spec`,
/// the type is taken to be an slice of two-tuples of initial instance id's:
/// []struct {
///     DomainRelClass.InitialInstanceId,
///     CodomainRelClass.InitialInstanceId }
///
/// For compound associations, i.e. those that supply a value for
/// the `AssociativeRelClass` field, the type is slice of three-tuples of initial
/// instance id's:
/// []struct {
///     DomainRelClass.InitialInstanceId,
///     CodomainRelClass.InitialInstanceId,
///     AssociativeRelClass.InitialInstanceId }

```

```

/// The return type is a namespace struct that holds the implementation
/// of the association configured with the `initial_assoc` associations
/// (if any).
pub fn Association(
    comptime association_spec: *const AssociationSpec,
    comptime initial_assocs: anytype,
) type {
    if (!isRelClassType(association_spec.DomainRelClass))
        @compileError("DomainRelClass argument is not a RelClass");
    if (!isRelClassType(association_spec.CodomainRelClass))
        @compileError("CodomainRelClass argument is not a RelClass");
    if (association_spec.AssociativeRelClass) |AssocClass| {
        if (!isRelClassType(AssocClass))
            @compileError("AssociativeRelClass argument is not a RelClass");
    }
    if (association_spec.PartitionClass) |Partition| {
        if (!isRelClassType(Partition))
            @compileError("PartitionClass argument is not a RelClass");
    }
    if (association_spec.DomainRelClass == association_spec.CodomainRelClass)
        validateReflexiveConstraints(association_spec.domain_ref, association_spec.codomain_ref);

    return struct {
        const Self = @This();

        <<association variables>>
        <<association declarations>>
        <<association public functions>>
        <<association private functions>>
    };
}

```

When the relational classes of the association are the same, we must ensure that the conditionality of the referential constraints are the same. This may seem like an unusual rule, but careful consideration shows that it is not possible to populate a reflexive association if one side of the association has a different conditionality than the other.

```

<<association module private functions>>=
/// The `validateReflexiveConstraints` function validates that the
/// referential constrains given by the `domain_ref` and `codomain_ref`
/// arguments are suitable for reflexive associations.
fn validateReflexiveConstraints(
    domain_ref: RefConstraint,
    codomain_ref: RefConstraint,
) void {
    if (isConstraintConditional(domain_ref) != isConstraintConditional(codomain_ref))
        @compileError(
            "reflexive referential constraints must have the same conditionality");
}

```

```
}
```

```
<<association module private functions>>=
/// The `isConstraintConditional` function returns `true` if the
/// referential constraint given by the `constraint` argument allows for
/// conditional association and `false` otherwise.
fn isConstraintConditional(
    constraint: RefConstraint,
) bool {
    return switch (constraint) {
        .@ "?" => true,
        .@ "1" => false,
        .@ "+" => false,
        .@ "*" => true,
    };
}
```

Following our previous practice, we include some meta-information in the result type that enables checking if a given type is indeed an Association.

```
<<association variables>>=
pub const is_association: void = {};
```

```
<<association module public functions>>=
/// The `isAssociationType` function returns `true` if the value of `AssociationType`
/// argument was generated by the `Association` type function and `false` otherwise.
pub fn isAssociationType(
    comptime AssociationType: type,
) bool {
    return @typeInfo(AssociationType) == .@ "struct" and
        @hasDecl(AssociationType, "is_association");
}
```

Two other constants are used to classify the particular type of association. An association may have an `AssociativeRelClass` or it may be *reflexive*. These properties are orthogonal so we intend the inclusive or of associativity and reflexivity.

```
<<association variables>>=
pub const is_associative = association_spec.AssociativeRelClass != null;
pub const is_reflexive: bool = association_spec.DomainRelClass == association_spec
    .CodomainRelClass;
```

It is convenient to record the core information about the association. Note that we create variables for each `RelClass` type, despite there being no fields in the `comptime` created `RelClass` namespaces. These variable occupy no space, but give us a convenient handle to place methods on the Association. Like `RelClass` types, `Association` types are intended as distinct singleton entities.

```

<<association variables>>=
pub const DomainClass = association_spec.DomainRelClass;
pub const domain: DomainClass = .{};
pub const CodomainClass = association_spec.CodomainRelClass;
pub const codomain: CodomainClass = .{};
pub const AssociativeClass = if (is_associative)
    association_spec.AssociativeRelClass.?
else
    void;
pub const associative = if (is_associative) AssociativeClass{} else void;

```

When an Association is compound, we must provide the two adjacency matrices necessary to hold the decomposition of the Association. As much as we might like to put the adjacency matrices in an array, they are not of the same data type. We keep track of the simple vs. compound requirements directly in the code. There are, after all, only two cases and will ever only be two cases.

```

<<association variables>>=
/// `Associations` that have an `AssociativeClass` require two
/// adjacency matrices.
pub const AssocMatrix1 = if (is_associative)
    AssociationAdjMatrix( // compound case, AssociativeClass to DomainClass
        AssociativeClass.ClassVar,
        association_spec.codomain_ref,
        @"1",
        DomainClass.ClassVar,
        &compound_initial_domain_assocs(),
    )
else
    AssociationAdjMatrix(
        DomainClass.ClassVar, // simple case, DomainClass to CodomainClass
        association_spec.codomain_ref,
        association_spec.domain_ref,
        CodomainClass.ClassVar,
        &simple_initial_assocs(),
    );
pub const AssocMatrix2 = if (is_associative)
    AssociationAdjMatrix( // compound case, AssociativeClass to CodomainClass
        AssociativeClass.ClassVar,
        association_spec.domain_ref,
        @"1",
        CodomainClass.ClassVar,
        &compound_initial_codomain_assocs(),
    )
else
    void; // simple case, no second adjacency matrix required
pub const assoc_matrix1: AssocMatrix1 = .{};
pub const assoc_matrix2 = if (is_associative) AssocMatrix2{} else void;

```

Because we can have two or three initial instances specified at comptime when the Association is

created, different functions are used to format the initial association instances as needed by the adjacency matrix functions, which are all binary.

```
<<association private functions>>=
fn simple_initial_assocs() [initial_assocs.len]struct {
    DomainClass.InitialInstanceId,
    CodomainClass.InitialInstanceId,
} {
    var initials: [initial_assocs.len]struct {
        DomainClass.InitialInstanceId,
        CodomainClass.InitialInstanceId,
    } = undefined;
    for (initial_assocs, 0...) /*initial_assoc, initial_index| {
        if (initial_assoc.len != 2)
            @compileError("simple associations require two initial instance ids");
        initials[initial_index] = .{
            initial_assoc@"0",
            initial_assoc@"1",
        };
    }

    return initials;
}
```

For the compound association case, we have to split domain from codomain to include the associative class instances. There are, after all, two adjacency matrices being initialized.

```
<<association private functions>>=
fn compound_initial_domain_assocs() [initial_assocs.len]struct {
    AssociativeClass.InitialInstanceId,
    DomainClass.InitialInstanceId,
} {
    var initials: [initial_assocs.len]struct {
        AssociativeClass.InitialInstanceId,
        DomainClass.InitialInstanceId,
    } = undefined;
    for (initial_assocs, 0...) /*initial_assoc, initial_index| {
        if (initial_assoc.len != 3)
            @compileError("associations with an associative class require three initial
instance ids");
        initials[initial_index] = .{
            initial_assoc@"2",
            initial_assoc@"0",
        };
    }

    return initials;
}
```

```
<<association private functions>>=
fn compound_initial_codomain_assocs() [initial_assocs.len]struct {
```

```

        AssociativeClass.InitialInstanceId,
        CodomainClass.InitialInstanceId,
    } {
        var initials: [initial_assocs.len]struct {
            AssociativeClass.InitialInstanceId,
            DomainClass.InitialInstanceId,
        } = undefined;
        for (initial_assocs, 0...) /*initial_assoc, initial_index| {
            if (initial_assoc.len != 3)
                @compileError("associations with an associative class require three initial
instance ids");
            initials[initial_index] = .{
                initial_assoc@"2",
                initial_assoc@"1",
            };
        }

        return initials;
    }
}

```

The tag given the association is important to have around when we discuss collecting association together into a domain.

```

<<association declarations>>=
pub const assoc_id = association_spec.assoc_tag;
pub var assoc_number: u8 = undefined;

```

```

<<association public functions>>=
pub fn init(
    number: u8,
) Self {
    assoc_number = number;
    initAssigner();
    return {};
}

```

Forming Associations Between Instances

Since associations are between set elements of relational classes, the `InstanceRef` type, introduced during the `RelClass` discussion, is used as the primary interface for association functions. The provided association functions that create and delete an association between instances mirror the capability of the adjacency matrix functions but giving them an `InstanceRef` interface.

```

<<association declarations>>=
pub const DomainInstanceRef = DomainClass.InstanceRef;
pub const CodomainInstanceRef = CodomainClass.InstanceRef;
pub const AssociativeInstanceRef = if (is_associative)
    AssociativeClass.InstanceRef
else

```

```
void;
```

The design of the function interfaces for relating and unrelating class instances uses the verb phrasing that indicates the traversal direction to determine which argument to the function is the domain and which is the codomain. The traversal direction is also used in the next section when the functions for association traversal are presented. Here, we want to capture the phrases from the `association_spec` as an enumeration. The phrases ultimately indicate the direction of the association. To preserve the direction of the phrasing, the enumeration values are set to the values of an enumeration that directly indicates direction. We use an enumeration indicating traversal direction as a means to transform the semantic intent of the association phrases and create a mapping from the phrases to direction.

```
<<association declarations>>
pub const TraversalPhrase = rblk: {
    const fields = [_]EnumField{
        .{
            .name = @tagName(association_spec.domain_phrase),
            .value = @intFromEnum(TraversalDirection.domain_to_codomain),
        },
        .{
            .name = @tagName(association_spec.codomain_phrase),
            .value = @intFromEnum(TraversalDirection.codomain_to_domain),
        },
    };
    break :rblk @Type(.{
        .@enum" = .{
            .tag_type = u1,
            .fields = &fields,
            .decls = &.{},
            .is_exhaustive = true,
        },
    });
};
```

```
<<association declarations>>
const TraversalDirection = enum(u1) {
    domain_to_codomain = 0,
    codomain_to_domain = 1,
}

pub fn fromPhrase(
    comptime phrase: TraversalPhrase,
) TraversalDirection {
    // Using the same numbers for the values in each enumeration allows
    // for this trivial mapping between them.
    return @enumFromInt(@intFromEnum(phrase));
}

pub fn toPhrase(
    comptime dir: TraversalDirection,
) TraversalPhrase {
    return @enumFromInt(@intFromEnum(dir));
}
```

```
    }  
};
```

Relate

```
<<association public functions>>=  
/// The `relate` function creates an instance of the given Association  
/// between two `RelClass` instances. The `one_inst` and `other_inst`  
/// arguments must be `InstanceRef` values for a class that participates  
/// in the association. If the association has an Associative  
/// Class, then `assoc_inst` must be an `InstanceRef` value of the  
/// Associative Class. Otherwise `assoc_inst` may given as `null`. The  
/// `traversal_phrase` argument is one of the two verb phrases specified  
/// for the association. If the `traversal_phrase` indicates a domain to  
/// codomain direction, then `one_inst` must be an `InstanceRef` of the  
/// class that serves as the domain of the association and `other_inst`  
/// must refer to a codomain instance. Conversely, if `traversal_phrase`  
/// indicates a codomain to domain direction, `one_inst` must refer to  
/// a codomain instance and `other_inst` must refer to a domain instance.  
pub fn relate(  
    _: Self,  
    one_inst: anytype,  
    comptime traversal_phrase: TraversalPhrase,  
    other_inst: anytype,  
    assoc_inst: anytype,  
) void {  
    const OneType = @TypeOf(one_inst);  
    const OtherType = @TypeOf(other_inst);  
  
    if (is_reflexive and OneType != OtherType) {  
        @panic("reflexive associations must relate instances of the same type");  
    } else if (!is_reflexive and OneType == OtherType) {  
        @panic("non-reflexive associations cannot relate instances of the same type");  
    }  
  
    if (is_associative and assoc_inst == null)  
        @panic("associations with associative classes must give an 'assoc_inst' argument");  
    if (is_associative and @TypeOf(assoc_inst) != AssociativeClass.InstanceRef)  
        @panic("bad associative class instance reference");  
  
    switch (comptime TraversalDirection.fromPhrase(traversal_phrase)) {  
        .domain_to_codomain => {  
            if (OneType != DomainClass.InstanceRef)  
                @panic("expected domain class instance reference: got '" ++  
                    @typeName(OneType) ++ "'");  
            if (OtherType != CodomainClass.InstanceRef)  
                @panic("expected codomain class instance reference: got '" ++  
                    @typeName(OtherType) ++ "'");  
  
            if (is_associative) {
```

```

        assoc_matrix1.relate(assoc_inst.tuple_ref, one_inst.tuple_ref);
        assoc_matrix2.relate(assoc_inst.tuple_ref, other_inst.tuple_ref);
    } else {
        assoc_matrix1.relate(one_inst.tuple_ref, other_inst.tuple_ref);
    }
},
.codomain_to_domain => {
    if (OneType != CodomainClass.InstanceRef)
        @panic("expected codomain class instance reference: got '" ++
               @typeName(OneType) ++ "'");
    if (OtherType != DomainClass.InstanceRef)
        @panic("expected domain class instance reference: got '" ++
               @typeName(OtherType) ++ "'");
    if (is_associative) {
        assoc_matrix1.relate(assoc_inst.tuple_ref, other_inst.tuple_ref);
        assoc_matrix2.relate(assoc_inst.tuple_ref, one_inst.tuple_ref);
    } else {
        assoc_matrix1.relate(other_inst.tuple_ref, one_inst.tuple_ref);
    }
},
}
}

```

Unrelate

```

<<association public functions>>=
/// The `unrelate` function deletes an instance of the given Association
/// between two `RelClass` instances. The `one_inst` and `other_inst`
/// arguments must be `InstanceRef` values for a class that participates
/// in the association. If the association has an Associative
/// Class, then `assoc_inst` must be an `InstanceRef` value of the
/// Associative Class. Otherwise `assoc_inst` may given as `null`. The
/// `traversal_phrase` argument is one of the two verb phrases specified
/// for the association. If the `traversal_phrase` indicates a domain to
/// codomain direction, then `one_inst` must be an `InstanceRef` of the
/// class that serves as the domain of the association and `other_inst`
/// must refer to a codomain instance. Conversely, if `traversal_phrase`
/// indicates a codomain to domain direction, `one_inst` must refer to
/// a codomain instance and `other_inst` must refer to a domain instance.
pub fn unrelated(
    _: Self,
    one_inst: anytype,
    comptime traversal_phrase: TraversalPhrase,
    other_inst: anytype,
    assoc_inst: anytype,
) void {
    const OneType = @TypeOf(one_inst);
    const OtherType = @TypeOf(other_inst);

    if (is reflexive) {

```

```

        assert(OneType == OtherType);
    }
    if (is_associative and assoc_inst == null)
        @panic("associations with associative classes must give an 'assoc_inst' argument");
    if (is_associative and @TypeOf(assoc_inst) != AssociativeClass.InstanceRef)
        @panic("bad associative class instance reference");

    switch (comptime TraversalDirection.fromPhrase(traversal_phrase)) {
        .domain_to_codomain => {
            if (OneType != DomainClass.InstanceRef)
                @panic("expected domain class instance reference: got '" ++
                    @typeName(OneType) ++ "'");
            if (OtherType != CodomainClass.InstanceRef)
                @panic("expected codomain class instance reference: got '" ++
                    @typeName(OtherType) ++ "'");
            if (is_associative) {
                assoc_matrix1.unrelate(assoc_inst.tuple_ref, one_inst.tuple_ref);
                assoc_matrix2.unrelate(assoc_inst.tuple_ref, other_inst.tuple_ref);
            } else {
                assoc_matrix1.unrelate(one_inst.tuple_ref, other_inst.tuple_ref);
            }
        },
        .codomain_to_domain => {
            if (OneType != CodomainClass.InstanceRef)
                @panic("expected codomain class instance reference: got '" ++
                    @typeName(OneType) ++ "'");
            if (OtherType != DomainClass.InstanceRef)
                @panic("expected domain class instance reference: got '" ++
                    @typeName(OtherType) ++ "'");
            if (is_associative) {
                assoc_matrix1.unrelate(assoc_inst.tuple_ref, other_inst.tuple_ref);
                assoc_matrix2.unrelate(assoc_inst.tuple_ref, one_inst.tuple_ref);
            } else {
                assoc_matrix1.unrelate(other_inst.tuple_ref, one_inst.tuple_ref);
            }
        },
    }
}

```

Tests

```

<<association module tests>>=
test "simple association" {
    const A = RelClass(
        .a,
        struct { id: u8, which: u8 = 'A' },
        &. {.id},
        &{},
        .{},
    );
}

```

```

4,
&.{  

    .{ .id0, .{ .id = 0 } },  

    .{ .id1, .{ .id = 1 } },  

    .{ .id2, .{ .id = 2 } },  

    .{ .id3, .{ .id = 3 } },  

},
);  

const a = A.init(0);

const B = RelClass(  

    .b,  

    struct { id: u8, which: u8 = 'B' },  

    &.{&.{.id}}},  

&.{},  

.&{},  

2,  

&.{  

    .{ .id0, .{ .id = 0 } },  

    .{ .id1, .{ .id = 1 } },  

},
);  

const b = B.init(1);

// A +- "passive" -r1- "active"-1 B
const r1_spec = AssociationSpec{
    .DomainRelClass = A,
    .codomain_ref = .@"+",
    .codomain_phrase = .passive,
    .assoc_tag = .r1,
    .domain_phrase = .active,
    .domain_ref = .@"1",
    .CodomainRelClass = B,
};

const R1 = Association(
    &r1_spec,
    &.{},
);
const r1: R1 = .{};

const a_0 = a.selectInitialInstance(.id0);
const a_1 = a.selectInitialInstance(.id1);
const a_2 = a.selectInitialInstance(.id2);
const a_3 = a.selectInitialInstance(.id3);
const b_0 = b.selectInitialInstance(.id0);
const b_1 = b.selectInitialInstance(.id1);

// A.0 -> B.0
r1.relate(a_0, .active, b_0, null);
// A.1 -> B.1
r1.relate(a_1, .active, b_1, null);

```

```

// A.2 -> B.0
r1.relate(a_2, .active, b_0, null);
// A.3 -> B.0
r1.relate(a_3, .active, b_0, null);
try testing.expect(r1.isConsistent());

// Unrelated A.1 and B.1 causes the association to be inconsistent
// because it results in A.1 not being related to any instance of B.
// The referential constraint says that all B instances must be related
// to at least one A instance, i.e. A ++1 B.
r1.unrelate(a_1, .active, b_1, null);
try testing.expect(!r1.isConsistent());

// Relating A.1 to B.0 does not restore consistency since B.1 is
// still unrelated to any A instance.
r1.relate(a_1, .active, b_0, null);
try testing.expect(!r1.isConsistent());

// Consistency is restored by relating A.3 to B.1 after unrelating
// it from B.0.
r1.unrelate(b_0, .passive, a_3, null);
r1.relate(b_1, .passive, a_3, null);
try testing.expect(r1.isConsistent());
}

```

There are use cases where we have a domain (codomain) instance set and wish to unrelated any codomain (domain) instances by having the system look up the related elements rather than specifying them through the function interface. This case can arise when relational class instances are deleted.

Since a given class instance may be related to multiple other instances and since we have to look up the related instances anyway, an `InstanceSet` of the related instance is returned. To be able to return an `InstanceSet` means we must determine at `comptime` the precise type of the `InstanceSet`.

```

<<association public functions>>=
pub fn InferRelatedClass(
    comptime traversal_phrase: TraversalPhrase,
) type {
    const dir = comptime TraversalDirection.fromPhrase(traversal_phrase);
    return switch (dir) {
        .domain_to_codomain => CodomainClass,
        .codomain_to_domain => DomainClass,
    };
}

```

```

<<association public functions>>=
/// The `unrelateInstance` function unrelates the given instance by the
/// `inst_ref` argument from any instances it is related to for the
/// given association.
pub fn unrelateInstance(

```

```

_: Self,
inst_ref: anytype,
comptime traversal_phrase: TraversalPhrase,
) InferRelatedClass(traversal_phrase).InstanceSet {
    if (is_associative) @compileError("must not be an association with an associative
class");

    const ReferencedClass = @TypeOf(inst_ref.classvar);
    const traversal_dir = comptime TraversalDirection.fromPhrase(traversal_phrase);
    switch (traversal_dir) {
        .domain_to_codomain => {
            if (ReferencedClass == DomainClass) {
                const unrelated = assoc_matrix1.unrelateDomainTuple(inst_ref.tuple_ref);
                return CodomainInstanceSet.initFromTupleRefSet(unrelated);
            } else
                @panic("instance reference type mismatches traversal phrase");
        },
        .codomain_to_domain => {
            if (ReferencedClass == CodomainClass) {
                const unrelated = assoc_matrix1.unrelateCodomainTuple(inst_ref.tuple_ref);
                return DomainInstanceSet.initFromTupleRefSet(unrelated);
            } else
                @panic("instance reference type mismatches traversal phrase");
        },
    }
}

```

The case of an association with an associative class is separated out since there are two returned instance sets.

```

<<association public functions>>=
pub fn unrelateInstanceAssociative(
    _: Self,
    inst_ref: anytype,
) struct { DomainClass.InstanceSet, CodomainClass.InstanceSet} {
    if (!is_associative) @compileError("must be an association with an associative class");

    return .{
        assoc_matrix1.unrelateDomainTuple(inst_ref.tuple_ref),
        assoc_matrix2.unrelateDomainTuple(inst_ref.tuple_ref),
    };
}

```

Rerelate

Sometimes the execution situation makes it more convenient to update an association to change the related instances than have to unrelate and then relate to different instances. A trivial convenience function is provided.

```

<<association public functions>>=
/// The `rerelate` function updates the association to unrelated

```

```

/// the `inst_ref` instance from the instances given by the
/// `curr_related_inst` argument and then relate `inst_ref`
/// to the instance given by the `new_related_inst` argument.
/// The `traversal_phrase` argument indicates the direction of the
/// association update.
pub fn rerelease(
    self: Self,
    inst_ref: anyhow::anytype,
    comptime traversal_phrase: TraversalPhrase,
    curr_related_inst: anyhow::anytype,
    new_related_inst: anyhow::anytype,
) void {
    if (is_associative)
        @panic("only simple associations supported");

    self.unrelate(inst_ref, traversal_phrase, curr_related_inst, null);
    self.relate(inst_ref, traversal_phrase, new_related_inst, null);
}

```

Relating by Selection

```

<<association public functions>>=
pub fn relateBySelection(
    self: Self,
    comptime traversal_phrase: TraversalPhrase,
    instance_selectors: []const struct {
        DomainClass.IdentifierAttributes(0),
        CodomainClass.IdentifierAttributes(0),
    },
) RelVarError!void {
    for (instance_selectors) |instance_selector| {
        const domain_selector = instance_selector.@"0";
        const domain_inst_ref = try domain.selectByIdentifier(0, domain_selector);

        const codomain_selector = instance_selector.@"1";
        const codomain_inst_ref = try codomain.selectByIdentifier(0, codomain_selector);

        self.relate(domain_inst_ref, traversal_phrase, codomain_inst_ref, null);
    }
}

```

```

<<association public functions>>=
pub fn relateBySelectionAssociative(
    self: Self,
    comptime traversal_phrase: TraversalPhrase,
    instance_selectors: []const struct {
        DomainClass.IdentifierAttributes(0),
        CodomainClass.IdentifierAttributes(0),
        AssociativeClass.IdentifierAttributes(0),
    },
) RelVarError!void {

```

```

for (instance_selectors) |instance_selector| {
    const domain_selector = instance_selector.@"0";
    const domain_inst_ref = try domain.selectByIdentifier(0, domain_selector);

    const codomain_selector = instance_selector.@"1";
    const codomain_inst_ref = try codomain.selectByIdentifier(0, codomain_selector);

    const assoc_selector = instance_selector.@"2";
    const assoc_inst_ref = try associative.selectByIdentifier(0, assoc_selector);

    self.relate(domain_inst_ref, traversal_phrase, codomain_inst_ref, assoc_inst_ref);
}
}

```

Association Referential Consistency

```

<<association public functions>>=
/// The `isConsistent` function returns `true` if the referential constraints
/// of the association are consistent with the populations of the
/// participating relational classes and `false` otherwise.
pub fn isConsistent(
    _: Self,
) bool {
    return if (is_associative)
        assoc_matrix1.isAssociationConsistent() and
        assoc_matrix2.isAssociationConsistent()
    else
        assoc_matrix1.isAssociationConsistent();
}

```

Association Traversal

We use the term, *traversal*, to describe computing the image or preimage of an association over a RelClass domain or codomain instance set. Traversal may proceed from the domain to the codomain or, conversely, from the codomain to the domain. Functions to compute the *complement* of a traversal is also given. If traversing an association gives a related set of instances, then the complement is a convenient way to ask for all the unrelated instances.

Traversal is always in terms of sets and we introduce some local naming conventions to distinguish between the sets involved.

```

<<association declarations>>=
pub const DomainInstanceSet = DomainClass.InstanceSet;
pub const CodomainInstanceSet = CodomainClass.InstanceSet;
pub const AssociativeInstanceSet = if (is_associative)
    AssociativeClass.InstanceSet
else
    void;

```

We want an interface that allows for conventional method-style chaining to make it convenient for traversing a sequence of associations. That means we must be able to determine the data type of the returned `InstanceSet` at compile time. That determination is made more complicated by having to consider the four variations on how association traversal is specified.

The following cases must be considered:

reflexive / non-reflexive

For reflexive association, a traversal phrase must be given since that is the only way to resolve the ambiguity of which direction the traversal is intended. Since the class is the same for a reflexive association, the destination class name is not sufficient. For non-reflexive associations, the specification of a traversal phrase is optional since specifying a destination class determines the direction.

simple / compound

If the association requires an associative class for its reification, then it is possible to traverse to the associative class itself, i.e. a new path in the traversal exists. For this case, the associative class name may be used to determine the direction of traversal for non-reflexive associations. Again, if the association is reflexive, then a traversal phrase is required.

traversal phrase provided

The traversal function interface allows an optional traversal phrase to be supplied. In some cases the traversal phrase is required, but may be supplied whether required or not.

destination class tag provided

Like a traversal phrase, the traversal function interface allows for providing the class tag for the traversal destination. In some cases it is simpler to provide the destination class and omit the traversal phrase.

There are 16 possible outcomes for the above four conditions. The first two, reflexive and compound, are bound at compile time when the `Association` function is invoked. The other two are provided as optional values through the traversal function arguments. It is always possible to supply both a traversal phrase and a destination class tag and they are evaluated to ensure the supplied values are consistent with the `Association` definition. In many cases, only one of the traversal phrase or destination class tag need be provided. The interface is intended to be flexible to the known properties of the `Association`.

Previously, we discussed the `TraversalPhrase` enumeration and its role in indicating traversal direction. The `TraversalDestinationClass` enumeration follows the same pattern of mapping model notation, in this case class tags, to the role the `RelClass` plays in the association.

```
<<association declarations>>=
// maps class id tag to the role of the class in the association
pub const TraversalDestClass = dblk: {
    var fields: []const EnumField = &[_]EnumField{
        .{
            .name = @tagName(DomainClass.class_id),
            .value = @intFromEnum(DestinationRole.domain),
        },
    };
    if (!is_reflexive) {
        fields = fields ++ &[_]EnumField{
            .{
                .name = @tagName(CodomainClass.class_id),
            }
        };
    }
}
```

```

        .value = @intFromEnum(DestinationRole.codomain),
    },
};

if (is_associative) {
    fields = fields ++ &[_]EnumField{
        .{
            .name = @tagName(AssociativeClass.class_id),
            .value = @intFromEnum(DestinationRole.associative),
        },
    };
}

break :dblk @Type(.{
    .@enum" = .{
        .tag_type = u2,
        .fields = fields,
        .decls = &.{},
        .is_exhaustive = true,
    },
});
};

```

The mapping of the class tag to the role of the RelClass in the association is easily performed since the enumeration values are the same.

```

<<association declarations>>=
const DestinationRole = enum(u2) {
    domain,
    codomain,
    associative,

    pub fn fromDestClass(
        comptime dest_class: TraversalDestClass,
    ) DestinationRole {
        return @enumFromInt(@intFromEnum(dest_class));
    }

    pub fn toDestClass(
        comptime role: DestinationRole,
    ) TraversalDestClass {
        return @enumFromInt(@intFromEnum(role));
    }
};

```

With 16 possible outcomes for the four predicates that determine the destination of a traversal, we can compute, at `comptime`, a 4-bit number that represents the outcome of the implied decision tree.

```

<<association private functions>>=
const TraversalDestSpec = enum(u4) {
    // The enumeration names are a mnemonic representing

```

```

// the reflexive, compound, phrase, and destination specified
// for a traversal. Lower case letters indicate the proposition
// is false; upper case indicate true.
rcpd = 0b0000,
rcpD = 0b0001,
rcPd = 0b0010,
rcPD = 0b0011,
rCpd = 0b0100,
rCpD = 0b0101,
rCPd = 0b0110,
rCPD = 0b0111,
Rcpd = 0b1000,
RcpD = 0b1001,
RcPd = 0b1010,
RcPD = 0b1011,
RCpd = 0b1100,
RCpD = 0b1101,
RCPd = 0b1110,
RCPD = 0b1111,

// The `init` function computes the traversal destination from the
// provided phrase and destination class information. These two
// arguments are combined with knowledge of the reflexivity and
// compound nature of the association definition.
pub fn init(
    comptime opt_traversal_phrase: ?TraversalPhrase,
    comptime opt_dest_class: ?TraversalDestClass,
) TraversalDestSpec {
    // A packed struct allows us to compute each of the four propositions
    // independently.
    const TraversalConditions = packed struct(u4) {
        d: u1,
        p: u1,
        c: u1,
        r: u1,
    };
    const conditions: TraversalConditions = .{
        .d = if (opt_dest_class != null) 1 else 0,
        .p = if (opt_traversal_phrase != null) 1 else 0,
        .c = if (is_associative) 1 else 0,
        .r = if (is_reflexive) 1 else 0,
    };
    // The result is just the enumeration tag that matches
    // numerically the computed conditions.
    return @enumFromInt(@as(u4, @bitCast(conditions)));
}
};


```

```

<<association public functions>>=
pub fn InferDestRelClass(
    comptime opt_traversal_phrase: ?TraversalPhrase,

```

```

    comptime opt_dest_class: ?TraversalDestClass,
) type {
    const dest_spec = comptime TraversalDestSpec.init(opt_traversal_phrase,
opt_dest_class);
    const dir = comptime
        if (opt_traversal_phrase) |traversal_phrase|
            TraversalDirection.fromPhrase(traversal_phrase)
        else {};
    const role = comptime
        if (opt_dest_class) |dest_class|
            DestinationRole.fromDestClass(dest_class)
        else {};

    switch (dest_spec) {
        .rcpd, .rCpd => @compileError("must specify at least a transversal phrase or class
destination"),
        .rcpD => {
            return if (role == .domain) DomainClass else CodomainClass;
        },
        .rcPd, .rCPd, .RcPd, .RCPd => {
            return if (dir == .domain_to_codomain) CodomainClass else DomainClass;
        },
        .rcPD, .RcPD => {
            return if (dir == .domain_to_codomain and role == .codomain)
                CodomainClass
            else if (dir == .codomain_to_domain and role == .domain)
                DomainClass
            else
                @compileError("mismatch between phrasing and destination class");
        },
        .rCpD => {
            return switch (role) {
                .domain => DomainClass,
                .codomain => CodomainClass,
                .associative => AssociativeClass,
            };
        },
        .rCPD, .RCPD => {
            return if (dir == .domain_to_codomain and role == .codomain)
                CodomainClass
            else if (dir == .codomain_to_domain and role == .domain)
                DomainClass
            else if (role == .associative)
                AssociativeClass
            else
                @compileError("mismatch between phrasing and destination class");
        },
        .Rcpd, .RcpD, .RCpd, .RCpD =>
            @compileError("reflexive association must specify a traversal phrase"),
    }
}

```

```

<<association public functions>>=
pub fn traverse(
    _: Self,
    inst_set: anytype,
    comptime opt_traversal_phrase: ?TraversalPhrase,
    comptime opt_dest_class: ?TraversalDestClass,
) InferDestRelClass(opt_traversal_phrase, opt_dest_class).InstanceSet {
    const SrcRelClass = @TypeOf(inst_set.classvar);
    const DestRelClass = comptime InferDestRelClass(opt_traversal_phrase, opt_dest_class);

    const dest_spec = comptime TraversalDestSpec.init(opt_traversal_phrase,
opt_dest_class);
    const dir = comptime
        if (opt_traversal_phrase) |traversal_phrase|
            TraversalDirection.fromPhrase(traversal_phrase)
        else {};
    const role = comptime
        if (opt_dest_class) |dest_class|
            DestinationRole.fromDestClass(dest_class)
        else {};

    return .initFromTupleRefSet(
        switch (dest_spec) {
            .rcpd, .rCpd, .Rcpd, .RcpD, .RCpd, .RCpD => unreachable,
            .rcpD =>
                if (SrcRelClass == DestRelClass) @compileError("association is not
reflexive")
                else if (SrcRelClass == DomainClass and DestRelClass == CodomainClass)
                    assoc_matrix1.image(inst_set.tuple_ref_set)
                else if (SrcRelClass == CodomainClass and DestRelClass == DomainClass)
                    assoc_matrix1.preimage(inst_set.tuple_ref_set)
                else unreachable,
            .rcPd, .rcPD, .RcPd, .RcPD =>
                switch (dir) {
                    .domain_to_codomain => assoc_matrix1.image(inst_set.tuple_ref_set),
                    .codomain_to_domain => assoc_matrix1.preimage(inst_set.tuple_ref_set),
                },
            .rCpD, .rCPD =>
                if (SrcRelClass == AssociativeClass) // associative to either domain or
codomain
                    switch (role) {
                        // associative to domain
                        .domain => assoc_matrix1.image(inst_set.tuple_ref_set),
                        // associative to codomain
                        .codomain => assoc_matrix2.image(inst_set.tuple_ref_set),
                        .associative =>
                            @compileError("cannot traverse from associative class to
itself"),
                    }
                else

```

```

switch (role) { // role is the role played by the destination class
    // codomain to domain
    // two hops through the associative class
    // codomain to associative to domain
    .domain =>
        assoc_matrix1.image(assoc_matrix2.preimage(inst_set
.tuple_ref_set)),


        // domain to codomain
        // two hops through the associative class
        // domain to associative to codomain
    .codomain =>
        assoc_matrix2.image(
            assoc_matrix1.preimage(
                inst_set.tuple_ref_set,
            ),
        ),


        // domain or codomain to associative
    .associative =>
        // one hop -- domain to associative
        if (SrcRelClass == DomainClass)
            assoc_matrix1.preimage(inst_set.tuple_ref_set)
        // one hop -- codomain to associative
        else if (SrcRelClass == CodomainClass)
            assoc_matrix2.preimage(inst_set.tuple_ref_set)
        else unreachable,
    },


    .rCPd, .RCPd =>
        // With no specified destination, traversal to the associative class is not
possible.
        if (SrcRelClass == AssociativeClass) // associative to either domain or
codomain
            switch (dir) {
                // one hop from associative to codomain
                .domain_to_codomain => assoc_matrix2.image(inst_set.tuple_ref_set),


                // one hop from associative to domain
                .codomain_to_domain => assoc_matrix1.image(inst_set.tuple_ref_set),
            }
        else
            switch (dir) {
                // two hops through the associative class
                // domain to associative to codomain
                .domain_to_codomain =>
                    assoc_matrix2.image(
                        assoc_matrix1.preimage(
                            inst_set.tuple_ref_set,
                        ),
                    ),


                    // two hops through the associative class

```

```

        // codomain to associative to domain
        .codomain_to_domain =>
            assoc_matrix1.image(
                assoc_matrix2.preimage(
                    inst_set.tuple_ref_set,
                ),
            ),
        },
    .RCPD =>
        // associative to either domain or codomain depending upon the traversal
phrase
        if (SrcRelClass == AssociativeClass)
            switch (dir) {
                // one hop from associative to codomain
                .domain_to_codomain => assoc_matrix2.image(inst_set.tuple_ref_set),
                // one hop from associative to domain
                .codomain_to_domain => assoc_matrix1.image(inst_set.tuple_ref_set),
            }
        else if (DestRelClass == AssociativeClass)
            switch (dir) {
                // one hop from domain to associative
                .domain_to_codomain => assoc_matrix1.preimage(inst_set
.tuple_ref_set),

                // one hop from codomain to associative
                .codomain_to_domain => assoc_matrix2.preimage(inst_set
.tuple_ref_set),
            }
        else
            switch (dir) {
                // two hops: domain to associative to codomain
                .domain_to_codomain =>
                    assoc_matrix2.image(
                        assoc_matrix1.preimage(
                            inst_set.tuple_ref_set,
                        ),
                    ),
                // two hops: codomain to associative to domain
                .codomain_to_domain =>
                    assoc_matrix1.image(
                        assoc_matrix2.preimage(
                            inst_set.tuple_ref_set,
                        ),
                    ),
            },
        },
    );
}
}

```

<<instance reference declarations>>=

```

pub fn traverseAssociation(
    inst_ref: InstanceRef,
    comptime relationship: anytype,
    comptime opt_traversal_phrase: ?@TypeOf(relationship).TraversalPhrase,
    comptime opt_dest_class: ?@TypeOf(relationship).TraversalDestClass,
) @TypeOf(relationship).InferDestRelClass(opt_traversal_phrase, opt_dest_class).InstanceSet
{
    const inst_set = inst_ref.instanceSetFromInstanceRef();
    return inst_set.traverseAssociation(relationship, opt_traversal_phrase,
opt_dest_class);
}

```

```

<<instance set public functions>>=
pub fn traverseAssociation(
    inst_set: InstanceSet,
    relationship: anytype,
    comptime opt_traversal_phrase: ?@TypeOf(relationship).TraversalPhrase,
    comptime opt_dest_class: ?@TypeOf(relationship).TraversalDestClass,
) @TypeOf(relationship).InferDestRelClass(opt_traversal_phrase, opt_dest_class).InstanceSet
{
    return relationship.traverse(inst_set, opt_traversal_phrase, opt_dest_class);
}

```

Tests

The following test uses the **Input Point** and **Point Scaling** example [discussed previously](#) in the section on binary relations. We reproduce the class diagram here for convenience.

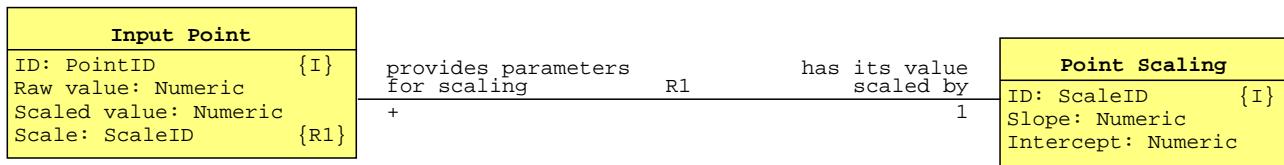


Figure 40. Class diagram for tests

```

<<association module tests>>=
test "association tests" {
    // Define the population that matches the example population.
    // First the codomain.
    // Map model data types to Zig data types.
    const ScaleID = ClassGeneratedId;
    const Numeric = i32;
    const PointScaling = RelClass(
        .point_scaling,
        struct {
            id: ScaleID = generated_id_default,
            slope: Numeric,

```

```

        intercept: Numeric,
    },
    &.{  

        &{.id},  

    },  

    &.{},  

    .{},  

10,  

    &.{  

        .{ .s2, .{ .slope = 2, .intercept = 4 } }, // 0  

        .{ .s4, .{ .slope = 4, .intercept = -3 } }, // 1
    },  

);
const point_scaling = PointScaling.init(0);

// Then the domain.
const PointID = ClassGeneratedID;
const InputPoint = RelClass(
    .input_point,
    struct {
        id: PointID = generated_id_default,
        raw_value: Numeric = 0,
        scaled_value: Numeric = 0,
    },
    &.{  

        &{.id},  

    },  

    &.{},  

    .{},  

20,  

    &.{  

        .{ .rv_27, .{ .raw_value = 27, .scaled_value = 58 } }, // 0  

        .{ .rv_42, .{ .raw_value = 42, .scaled_value = 88 } }, // 1  

        .{ .rv_10, .{ .raw_value = 10, .scaled_value = 37 } }, // 2
    },
);
const input_point = InputPoint.init(1);

// An Association is specification between data types.
const r1_spec = AssociationSpec{
    .DomainRelClass = InputPoint,
    .codomain_ref = .@"+",
    .codomain_phrase = .@"provides parameters for scaling",
    .assoc_tag = .r1,
    .domain_phrase = .@"has its value scaled by",
    .domain_ref = .@"1",
    .CodomainRelClass = PointScaling,
};
const R1 = Association(
    &r1_spec,
    &.{ // initial_assocs
        .{ .rv_27, .s2 }, // input_point.0 -> point_scaling.0

```

```

        .{ .rv_42, .s2 }, // input_point.1 -> point_scaling.0
        .{ .rv_10, .s4 }, // input_point.2 -> point_scaling.1
    },
);
const r1: R1 = .{};

// Make sure we managed to obtain a consistent population.
try testing.expect(r1.isConsistent());

// At this point, the classes are created and properly associated.
// Select some instances and traverse the Association.
const scaling_0 = point_scaling.selectInitialInstance(.s2);
const scaling_1 = point_scaling.selectInitialInstance(.s4);

const pt_0 = input_point.selectInitialInstance(.rv_27);
const pt_2 = input_point.selectInitialInstance(.rv_10);

const pt_0_scaler = pt_0.traverse(r1, @"has its value scaled by", null);
try testing.expectEqual(@as(usize, 1), pt_0_scaler.cardinality());
try testing.expectEqual(@as(isize, 2), pt_0_scaler.readAttribute(.slope));
try testing.expectEqual(@as(isize, 4), pt_0_scaler.readAttribute(.intercept));

const pt_2_scaler = pt_2.traverseAssociation(r1, null, .point_scaling);
try testing.expectEqual(@as(usize, 1), pt_2_scaler.cardinality());
try testing.expectEqual(@as(isize, 4), pt_2_scaler.readAttribute(.slope));
try testing.expectEqual(@as(isize, -3), pt_2_scaler.readAttribute(.intercept));

const scale_0_pts = scaling_0.traverseAssociation(r1, null, .input_point);
try testing.expectEqual(@as(usize, 2), scale_0_pts.cardinality());

const scale_1_pts = scaling_1.traverseAssociation(r1, null, .input_point);
try testing.expectEqual(@as(usize, 1), scale_1_pts.cardinality());
try testing.expectEqual(@as(isize, 10), scale_1_pts.readAttribute(.raw_value));
try testing.expectEqual(@as(isize, 37), scale_1_pts.readAttribute(.scaled_value));

// Find all the points that have the same scaler as pt_0.
// Association traversal composes, so starting at pt_0 -> Point Scaling
// gives the *one* related Point Scaling. Going back to Input Point gives
// the *set* of Input Points that all have the same scaling, i.e. the
// one that scales pt_0.
const same_scaler =
    pt_0.traverseAssociation(r1, null, .point_scaling).
        traverseAssociation(r1, null, .input_point);
try testing.expectEqual(@as(usize, 2), same_scaler.cardinality());
}

```

Reflexive associations sometimes require particular mental gymnastics because the domain and codomain are the same relational class. Consider the following UML class diagram.

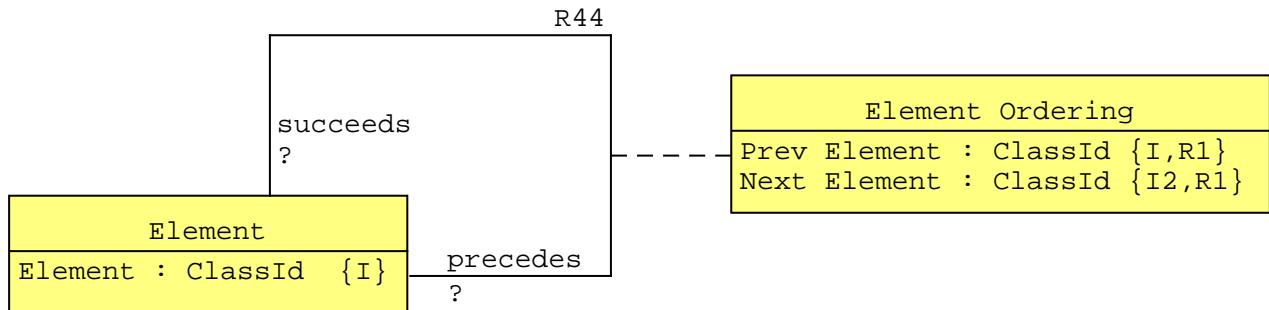


Figure 41. Reflexive Association Example

This is a reflexive association (also called a homogeneous binary relation) that is at-most-one to at-most-one. This association can be used as a *serial relation*, imposing an order on the domain elements, by properly relating the class instances and by careful traversal of the association. It can be considered as a relational version of a linear list.

Formally, the class diagram uses an associative class, **Element Ordering**, to realize the relationship. In this implementation, a association adjacency matrix provides all the information that the associative class does. Since the associative class has no other attributes, it can be eliminated.

```

<<association module tests>>=
test "reflexive associations" {
    const Element = RelClass(
        .element,
        struct { id: u8 },
        &{&.{.id}},
        &{},
        .{},
        20,
        &{
            .{ .id0, .{ .id = 0 } },
            .{ .id1, .{ .id = 1 } },
            .{ .id2, .{ .id = 2 } },
            .{ .id3, .{ .id = 3 } },
        },
    );
    const element = Element.init(0);

    const r44_spec = AssociationSpec{
        .DomainRelClass = Element,
        .codomain_ref = .@"?",
        .codomain_phrase = .succeeds,
        .assoc_tag = .r44,
        .domain_phrase = .precedes,
        .domain_ref = .@"?",
        .CodomainRelClass = Element,
    };
}

```

```

const R44 = Association(
    &r44_spec,
    &.{},
);
const r44: R44 = .{};

// Populate the association using instance selection
// Here, we form the sequence: 0 -- 1 -- 2 -- 3
try r44.relateBySelection(.succeeds, &{
    .{ .{.id = 0}, {.id = 1}, },
    .{ .{.id = 1}, {.id = 2}, },
    .{ .{.id = 2}, {.id = 3}, },
});
try testing.expect(r44.isConsistent());

// Note that the element at the head never appears in the set of
// related instances on the ".succeeds" side and the element at
// the tail never appears in the set of related instances on the
// ".precedes" side. For this example, 0 is the head of the sequence
// and 3 is the tail.

// To never appear as a .succeeds instance, means the head must appear in the
// complement of the domain traversal in the `succeeds` direction.
var head_element = r44.traverse(element.selectAll(), .succeeds, null).complement();
try testing.expectEqual(@as(u8, 1), head_element.cardinality());
try testing.expectEqual(@as(u8, 0), head_element.readAttribute(.id));

// Inverting the traversal direction computes the element at the tail.
var tail_element = r44.traverse(element.selectAll(), .precedes, null).complement();
try testing.expectEqual(@as(u8, 1), tail_element.cardinality());
try testing.expectEqual(@as(u8, 3), tail_element.readAttribute(.id));

// Traversing in the `succeeds` direction with the head element gives the next
instance
// in the sequence.
var after_head = r44.traverse(head_element, .succeeds, null);
try testing.expectEqual(@as(u8, 1), after_head.cardinality());
try testing.expectEqual(@as(u8, 1), after_head.readAttribute(.id));

// There is no element before the head, so traversing in the `precedes` direction
// starting at the head is empty.
// Traversing from the codomain with the head is the predecessor function.
var before_head = r44.traverse(head_element, .precedes, null);
try testing.expect(before_head.empty());

// There is no element after the tail.
var after_tail = r44.traverse(tail_element, .succeeds, null);
try testing.expect(after_tail.empty());

// The element before the tail is 2.
var before_tail = r44.traverse(tail_element, .precedes, null);

```

```

try testing.expectEqual(@as(usize, 1), before_tail.cardinality());
try testing.expectEqual(@as(u8, 2), before_tail.readAttribute(.id));

// Association traversal is composable in the mathematical sense.
// f . g(x) => g(f(x))
const third = head_element.
    traverseAssociation(r44, .succeeds, null). // first traversal => 0 - 1
    traverseAssociation(r44, .succeeds, null). // second traversal => 1 - 2
    traverseAssociation(r44, .succeeds, null); // third traversal => 2 - 3
try testing.expectEqual(@as(usize, 1), third.cardinality());
try testing.expectEqual(@as(u8, 3), third.readAttribute(.id));

// By rerelecting the elements in the sets, we can obtain a new ordering.
// Previous order: 0 -- 1 -- 2 -- 3
// New order: 0 -- 2 -- 1 -- 3
const e_0 = try element.selectByIdentifier(0, {.id = 0});
const e_1 = try element.selectByIdentifier(0, {.id = 1});
const e_2 = try element.selectByIdentifier(0, {.id = 2});
const e_3 = try element.selectByIdentifier(0, {.id = 3});

r44.rerelate(e_0, .succeeds, e_1, e_2); // 0 -- 1 is now 0 -- 2
r44.rerelate(e_1, .succeeds, e_2, e_3); // 1 -- 2 is now 1 -- 3
r44.rerelate(e_2, .succeeds, e_3, e_1); // 2 -- 3 is now 2 -- 1

try testing.expect(r44.isConsistent());

// Now the element after the head element is 2.
after_head = head_element.traverseAssociation(r44, .succeeds, null);
try testing.expectEqual(@as(usize, 1), after_head.cardinality());
try testing.expectEqual(@as(u8, 2), after_head.readAttribute(.id));

// The element before the tail is 1.
before_tail = tail_element.traverseAssociation(r44, .precedes, null);
try testing.expectEqual(@as(usize, 1), before_tail.cardinality());
try testing.expectEqual(@as(u8, 1), before_tail.readAttribute(.id));
}

```

Association Use Cases

There are 10 distinct combinations of constraints for associations where each participant is a distinct class. We show 10 test cases to illustrate how association traversal works and how consistency checking ensures that the association constraints are met. For each case, a set diagram is presented to help visualize the association. The set **A** is the domain and the set **B** is the codomain. The population and labeling of the sets in the diagram are mirrored in the program code.

One to One

One-to-one (**1—1**) associations are, mathematically, bijective functions. This means that every element in the domain is related to exactly one element of the codomain and *vice versa*.

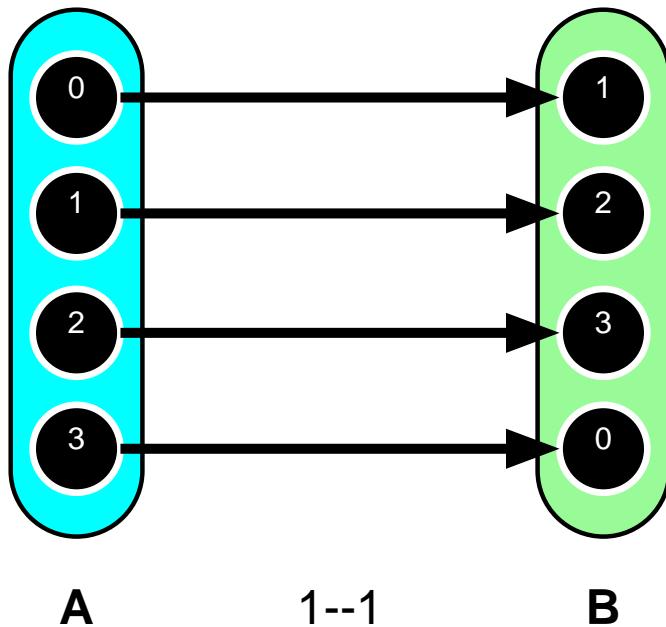


Figure 42. 1 to 1 association as a set diagram

```

<<association module tests>>=
test "one to one association" {
    const A = RelClass(
        .a,
        struct { id: u8, which: u8 = 'A' },
        &.{{&.{.id}}},
        &.{},
        .{ },
        20,
        &.{{
            .{ .id0, .{ .id = 0 } },
            .{ .id1, .{ .id = 1 } },
            .{ .id2, .{ .id = 2 } },
            .{ .id3, .{ .id = 3 } },
        }},
    );
    const a = A.init(0);

    const B = RelClass(
        .b,
        struct { id: u8, which: u8 = 'B' },
        &.{{&.{.id}}},
        &.{},
        .{ },
        20,
        &.{{
            .{ .id0, .{ .id = 0 } },
            .{ .id1, .{ .id = 1 } },
        }},
    );
}

```

```

        .{ .id2, .{ .id = 2 } },
        .{ .id3, .{ .id = 3 } },
    },
);
const b = B.init(1);

// A 1-"passive" -R1- "active"-1 B
const r1_spec = AssociationSpec{
    .DomainRelClass = A,
    .codomain_ref = .@"1",
    .codomain_phrase = .passive,
    .assoc_tag = .r1,
    .domain_phrase = .active,
    .domain_ref = .@"1",
    .CodomainRelClass = B,
};
const R1 = Association(
    &r1_spec,
    &.{{
        .{ .id0, .id1 },
        .{ .id1, .id2 },
        .{ .id2, .id3 },
        .{ .id3, .id0 },
    }},
);
const r1: R1 = .{};

try testing.expect(r1.isConsistent());

// Find the B element related to A.1, i.e. B.2
const a_1 = try a.selectByIdentifier(0, .{ .id = 1 });
const b_related = a_1.traverseAssociation(r1, null, .b);
try testing.expectEqual(@as(usize, 1), b_related.cardinality());
try testing.expectEqual(@as(u8, 2), b_related.readAttribute(.id));
try testing.expectEqual('B', b_related.readAttribute(.which));

// Find the A element that is related to B.3, i.e. A.2
const b_3 = try b.selectByIdentifier(0, .{ .id = 3 });
const a_related = b_3.traverseAssociation(r1, null, .a);
try testing.expectEqual(@as(usize, 1), a_related.cardinality());
try testing.expectEqual(@as(u8, 2), a_related.readAttribute(.id));
try testing.expectEqual('A', a_related.readAttribute(.which));

// Unrelate A.1 and B.2
// This makes the adjacency matrix referentially inconsistent.
const b_2 = try b.selectByIdentifier(0, .{ .id = 2 });
r1.unrelate(a_1, .active, b_2, null);
try testing.expect(!r1.isConsistent());
}

```

At-Most-One to One

At-most-one to one ($?--1$) associations are injective functions. Every A is related to exactly one B (function), and every related B is related to only one A (injective) but not all B's are necessarily related.

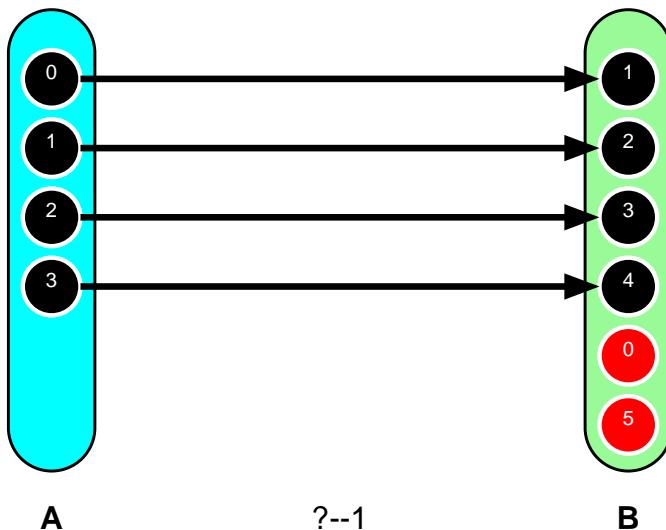


Figure 43. At most 1 to 1 association as a set diagram

```
<<association module tests>>=
test "at-most-one to one" {
    const A = RelClass(
        .a,
        struct { id: u8, which: u8 = 'A' },
        &.{{&{.id}}},
        &{},
        .{},
        20,
        &.{.
            .{ .id0, .{.id = 0} },
            .{ .id1, .{.id = 1} },
            .{ .id2, .{.id = 2} },
            .{ .id3, .{.id = 3} },
        },
    );
    const a = A.init(0);

    const B = RelClass(
        .b,
        struct { id: u8, which: u8 = 'B' },
        &.{{&{.id}}},
        &{},
        .{},
        20,
        &.{.
```

```

    .{ .id0, .{.id = 0} },
    .{ .id1, .{.id = 1} },
    .{ .id2, .{.id = 2} },
    .{ .id3, .{.id = 3} },
    .{ .id4, .{.id = 4} },
    .{ .id5, .{.id = 5} },
  },
);
const b = B.init(1);

// A ?-"passive" -R1- "active"-1 B
const r1_spec = AssociationSpec{
  .DomainRelClass = A,
  .codomain_ref = .@"?",
  .codomain_phrase = .passive,
  .assoc_tag = .r1,
  .domain_phrase = .active,
  .domain_ref = .@"1",
  .CodomainRelClass = B,
};
const R1 = Association(
  &r1_spec,
  // All A's must be related to one B.
  // B.0 and B.5 are unrelated to any A.
  &.{.
    .{ .id0, .id1 }, // 0 -> 1
    .{ .id1, .id2 }, // 1 -> 2
    .{ .id2, .id3 }, // 2 -> 3
    .{ .id3, .id4 }, // 3 -> 4
  },
);
const r1: R1 = .{};

try testing.expect(r1.isConsistent());

// Find the B element related to A.1, i.e. B.2
const a_1 = try a.selectByIdentifier(0, .{ .id = 1 });
const b_related = a_1.traverseAssociation(r1, null, .b);
try testing.expectEqual(@as(u8, 1), b_related.cardinality());
try testing.expectEqual(@as(u8, 2), b_related.readAttribute(.id));
try testing.expectEqual('B', b_related.readAttribute(.which));

// Find the A element that is related to B.3, i.e. A.2
const b_3 = try b.selectByIdentifier(0, .{ .id = 3 });
const a_related = b_3.traverseAssociation(r1, null, .a);
try testing.expectEqual(@as(u8, 1), a_related.cardinality());
try testing.expectEqual(@as(u8, 2), a_related.readAttribute(.id));
try testing.expectEqual('A', a_related.readAttribute(.which));

// Find the unrelated B's (red B's), i.e. B.0 and B.5
const b_unrelated = r1.traverse(a.selectAll(), null, .b).complement();

```

```

try testing.expectEqual(@as(usize, 2), b_unrelated.cardinality());
try testing.expect(b_unrelated.contains(try b.selectByIdentifier(0, .{ .id = 0 })));
try testing.expect(b_unrelated.contains(try b.selectByIdentifier(0, .{ .id = 5 })));

// Unrelate A.1 from B.2
// This makes the association referentially inconsistent
// because A.1 is not related to any element of B.
// The fact that B.2 is unrelated is fine.
const b_2 = try b.selectByIdentifier(0, .{ .id = 2 });
r1.unrelate(b_2, .passive, a_1, null);
try testing.expect(!r1.isConsistent());

// Restore consistency by relating A.1 to B.5
const b_5 = try b.selectByIdentifier(0, .{ .id = 5 });
r1.relate(a_1, .active, b_5, null);
try testing.expect(r1.isConsistent());

// Relating A.1 back to B.2 again causes the association
// to become inconsistent because A.1 is related to more than one
// codomain element.
r1.relate(a_1, .active, b_2, null);
try testing.expect(!r1.isConsistent());

// Simply deleting A.1 restores consistency, leaving B.2 and B.5
// unrelated. Deleting A.1 is equivalent to unrelating A.1
// from any codomain element and removing it from the domain universe set.
_= r1.unrelateInstance(a_1, .active);
a_1.destroy();
try testing.expect(r1.isConsistent());
}

```

At-Least-One to One

At-least-one to one (**+--1**) associations are surjective functions. This means all the A's must be related to exactly one B (function), and all the B's must be related to some A (surjective, i.e. the B set is covered by the A set) but any given B may be related to multiple A's.

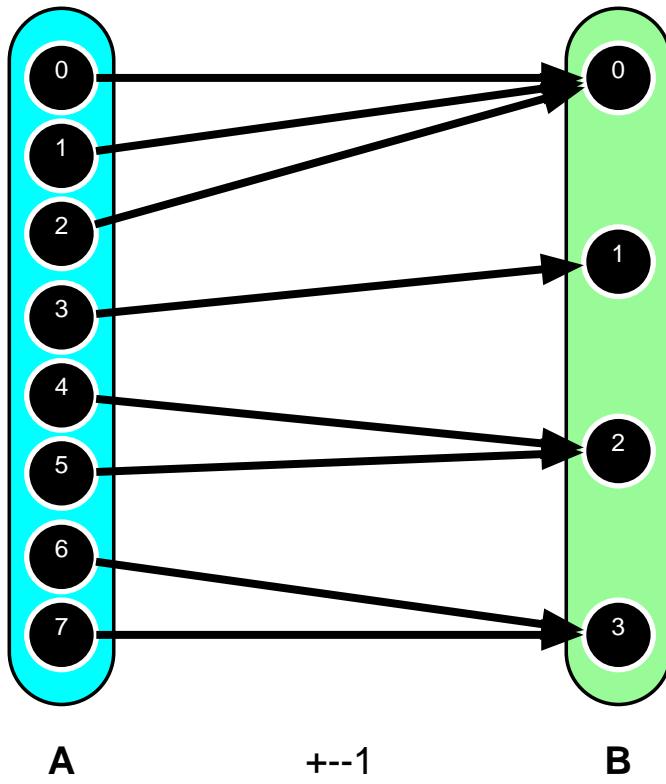


Figure 44. At least 1 to 1 association as a set diagram

```

<<association module tests>>
test "at-least-one to one" {
    const A = RelClass(
        .a,
        struct { id: u8, which: u8 = 'A' },
        &.{{&.{{id}}}},
        &.{},
        .{},
        20,
        &{
            .{ .id0, .{{id = 0}} },
            .{ .id1, .{{id = 1}} },
            .{ .id2, .{{id = 2}} },
            .{ .id3, .{{id = 3}} },
            .{ .id4, .{{id = 4}} },
            .{ .id5, .{{id = 5}} },
            .{ .id6, .{{id = 6}} },
            .{ .id7, .{{id = 7}} },
        },
    );
    const a = A.init(0);

    const B = RelClass(
        .b,

```

```

struct { id: u8, which: u8 = 'B' },
&.{&.{.id}},
&.{},
.&{},
20,
&.{.
    .{ .id0, .{.id = 0} },
    .{ .id1, .{.id = 1} },
    .{ .id2, .{.id = 2} },
    .{ .id3, .{.id = 3} },
},
);
const b = B.init(1);

// A +-"passive" -R1- "active"-1 B
const r1_spec = AssociationSpec{
    .DomainRelClass = A,
    .codomain_ref = .@"+",
    .codomain_phrase = .passive,
    .assoc_tag = .r1,
    .domain_phrase = .active,
    .domain_ref = .@"1",
    .CodomainRelClass = B,
};
const R1 = Association(&r1_spec, &.{});
const r1: R1 = .{};

// All the A's must be related.
// Note we relate several A's to the same B

// A.0, A.1, A.2 -> B.0
const b_0 = b.selectInitialInstance(.id0);
r1.relate(a.selectInitialInstance(.id0), .active, b_0, null);
r1.relate(a.selectInitialInstance(.id1), .active, b_0, null);
r1.relate(a.selectInitialInstance(.id2), .active, b_0, null);

// A.3 -> B.1
r1.relate(
    a.selectInitialInstance(.id3),
    .active,
    b.selectInitialInstance(.id1),
    null,
);
// A.4, A.5 -> B.2
const b_2 = b.selectInitialInstance(.id2);
r1.relate(a.selectInitialInstance(.id4), .active, b_2, null);
r1.relate(a.selectInitialInstance(.id5), .active, b_2, null);

// A.6, A.7 -> B.3
const b_3 = b.selectInitialInstance(.id3);
r1.relate(a.selectInitialInstance(.id6), .active, b_3, null);

```

```

r1.relate(a.selectInitialInstance(.id7), .active, b_3, null);

try testing.expect(r1.isConsistent());

// Find the B (there's always exactly one for this type of association)
// related to A.1
const a_1 = a.selectInitialInstance(.id1);
const b_related = a_1.traverseAssociation(r1, null, .b);
try testing.expectEqual(@as(usize, 1), b_related.cardinality());

// Find the set of A's that are related to B.2
// There is always at least one and in this case there are two, i.e. A.4 and A.5.
const a_related = b_2.traverseAssociation(r1, null, .a);
try testing.expectEqual(@as(usize, 2), a_related.cardinality());
try testing.expect(a_related.contains(try a.selectByIdentifier(0, {.id = 4})));
try testing.expect(a_related.contains(try a.selectByIdentifier(0, {.id = 5})));

// Unrelate A.1
// This makes the association referentially inconsistent
// i.e. this makes A.1 red.
const b_relatedto_a_1 = r1.unrelateInstance(a_1, .active);
try testing.expectEqual(@as(usize, 1), b_relatedto_a_1.cardinality());
try testing.expect(b_relatedto_a_1.contains(b.selectInitialInstance(.id0)));
try testing.expect(!r1.isConsistent());

// Consistency can be restored by deleting A.1.
// This is because B.0 is also related to A.0 and A.2
a_1.destroy();
try testing.expect(r1.isConsistent());

// Deleting A.3 causes inconsistency because B.1 is only related
// to A.3 or, in other words, deleting A.3 turns B.1 to red and
// no red elements are allowed in a --1 association.
const a_3 = a.selectInitialInstance(.id3);
const b_relatedto_a_3 = r1.unrelateInstance(a_3, .active);
try testing.expectEqual(@as(usize, 1), b_relatedto_a_3.cardinality());
try testing.expect(b_relatedto_a_3.contains(b.selectInitialInstance(.id1)));
a_3.destroy();
try testing.expect(!r1.isConsistent());
}

```

Any to One

The any to one (--1) association is mathematically a non-surjective function. This means that each A is related to one B (function) but some B's may not be related to any A (non-surjective, *i.e.* the B set is not covered), and multiple A's may be related to the same B. It's the fourth possibility for arranging the two sets when the association relation is a function.

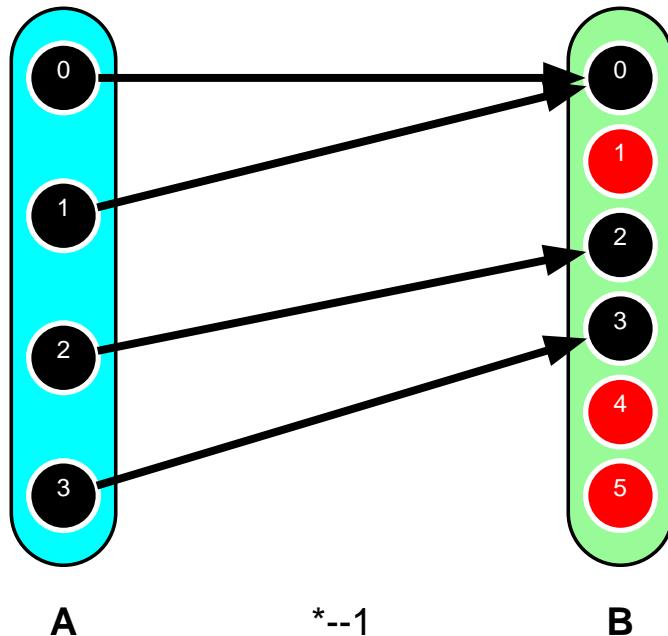


Figure 45. Any to 1 association as a set diagram

```
<<association module tests>>=
test "any to one" {
    const A = RelClass(
        .a,
        struct { id: u8, which: u8 = 'A' },
        &.{{&.{.id}}},
        &.{},
        .{ },
        20,
        &.{{
            .{ .id0, {.id = 0} },
            .{ .id1, {.id = 1} },
            .{ .id2, {.id = 2} },
            .{ .id3, {.id = 3} },
        }},
    );
    const a = A.init(0);

    const B = RelClass(
        .b,
        struct { id: u8, which: u8 = 'B' },
        &.{{&.{.id}}},
        &.{},
        .{ },
        20,
        &.{{
            .{ .id0, {.id = 0} },
            .{ .id1, {.id = 1} },
        }},
    );
}
```

```

        .{ .id2, {.id = 2} },
        .{ .id3, {.id = 3} },
        .{ .id4, {.id = 4} },
        .{ .id5, {.id = 5} },
    },
);
const b = B.init(1);

// A *-"passive" -R1- "active"-1 B
const r1_spec = AssociationSpec{
    .DomainRelClass = A,
    .codomain_ref = .@"",
    .codomain_phrase = .passive,
    .assoc_tag = .r1,
    .domain_phrase = .active,
    .domain_ref = .@"1",
    .CodomainRelClass = B,
};
const R1 = Association(&r1_spec, &.{});
const r1: R1 = .{};

// Relate all the domain elements to the codomain elements.
// Note we relate several A's to the same B

// A.0, A.1 -> B.0
const b_0 = b.selectInitialInstance(.id0);
r1.relate(a.selectInitialInstance(.id0), .active, b_0, null);
r1.relate(a.selectInitialInstance(.id1), .active, b_0, null);

// A.2 -> B.2
r1.relate(
    a.selectInitialInstance(.id2),
    .active,
    b.selectInitialInstance(.id2),
    null,
);
// A.3 -> B.3
r1.relate(
    a.selectInitialInstance(.id3),
    .active,
    b.selectInitialInstance(.id3),
    null,
);

// Codomain elements B.1, B.4, and B.5 are unrelated.

try testing.expect(r1.isConsistent());

// Find the B (there's always only one in this case) related to A.1
const a_1 = a.selectInitialInstance(.id1);
const b_related = a_1.traverseAssociation(r1, null, .b);
try testing.expectEqual(@as(usize, 1), b_related.cardinality());

```

```

try testing.expect(b_related.contains(try b.selectByIdentifier(0, .{ .id = 0 })));

// Find the set of A's related to B.0
const a_related = b_0.traverseAssociation(r1, null, .a);
try testing.expectEqual(@as(usize, 2), a_related.cardinality());
try testing.expect(a_related.contains(try a.selectByIdentifier(0, .{ .id = 0 })));
try testing.expect(a_related.contains(try a.selectByIdentifier(0, .{ .id = 1 })));

// Find the unrelated B's (red B's)
const b_unrelated = r1.traverse(a.selectAll(), null, .b).complement();
try testing.expect(b_unrelated.contains(try b.selectByIdentifier(0, .{ .id = 1 })));
try testing.expect(b_unrelated.contains(try b.selectByIdentifier(0, .{ .id = 4 })));
try testing.expect(b_unrelated.contains(try b.selectByIdentifier(0, .{ .id = 5 })));

// Find the unrelated A's. There can be none because the association
// relation is a total function.
const a_unrelated = r1.traverse(b.selectAll(), null, .a).complement();
try testing.expect(a_unrelated.empty());

// Unrelate A.1 from B.0
// This makes the association referentially inconsistent.
r1.unrelate(a_1, .active, b_0, null);
try testing.expect(!r1.isConsistent());

// Deleting A.1 makes the association consistent again.
// In fact deleting any A instance leaves the association consistent.
_ = r1.unrelateInstance(a_1, .active);
a_1.destroy();
try testing.expect(r1.isConsistent());
}

```

At-Most-One to At-Most-One

At-most-one to at-most-one (?--?) associations are injective partial functions. The association is a partial function because some domain elements are not mapped. For a total function, all domain elements are mapped to some codomain element. When we say that this association is an injective partial function, we mean that if the domain is divided into two subsets, one containing the related elements and the other containing the unrelated elements, then the relation is an injective function when only the related subset of the domain is considered. This line of reasoning applies to the other two partial functions in the following sections. Note if you perform the same partitioning of the codomain into related and unrelated elements, then this association can be considered a bijective function on the related domain elements and the related codomain subset. We don't use that perspective here since we use the term, *partial function*, to highlight the fact that there are elements of the domain that are disassociated from any codomain element and that fact alone makes the association a partial function.

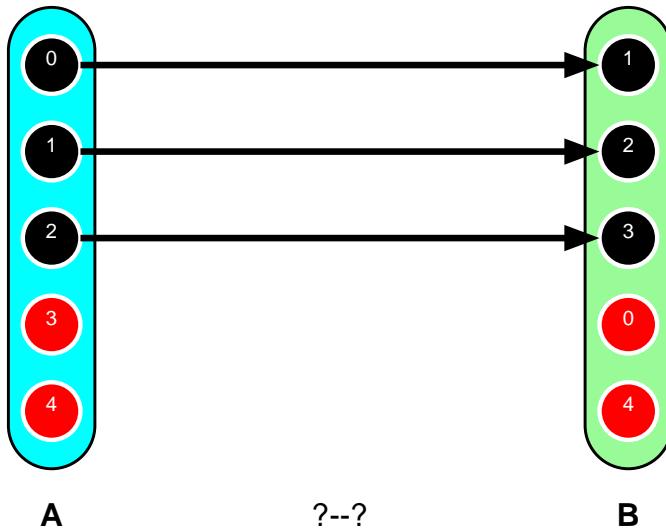


Figure 46. At most 1 to at most 1 association as a set diagram

```
<<association module tests>>
test "at-most-one to at-most-one" {
    const A = RelClass(
        .a,
        struct { id: u8, which: u8 = 'A' },
        &.{{&{.id}}},
        &.{},
        .{ },
        20,
        &.{.
            .{ .id0, .{.id = 0} },
            .{ .id1, .{.id = 1} },
            .{ .id2, .{.id = 2} },
            .{ .id3, .{.id = 3} },
            .{ .id4, .{.id = 4} },
        },
    );
    const a = A.init(0);
    const B = RelClass(
        .b,
        struct { id: u8, which: u8 = 'B' },
        &.{{&{.id}}},
        &.{},
        .{ },
        20,
        &.{.
            .{ .id0, .{.id = 0} },
            .{ .id1, .{.id = 1} },
            .{ .id2, .{.id = 2} },
            .{ .id3, .{.id = 3} },
            .{ .id4, .{.id = 4} },
        },
    );
}
```

```

);
const b = B.init(1);

// A ?-"passive" -R1- "active"-? B
const r1_spec = AssociationSpec{
    .DomainRelClass = A,
    .codomain_ref = .@"?",
    .codomain_phrase = .passive,
    .assoc_tag = .r1,
    .domain_phrase = .active,
    .domain_ref = .@"?",
    .CodomainRelClass = B,
};
const R1 = Association(&r1_spec, &.{});
const r1: R1 = .{};

// Relate 3 elements in both the domain and codomain.

// A.0 -> B.1
r1.relate(
    a.selectInitialInstance(.id0),
    .active,
    b.selectInitialInstance(.id1),
    null,
);
// A.1 -> B.2
r1.relate(
    a.selectInitialInstance(.id1),
    .active,
    b.selectInitialInstance(.id2),
    null,
);
// A.2 -> B.3
r1.relate(
    a.selectInitialInstance(.id2),
    .active,
    b.selectInitialInstance(.id3),
    null,
);

// Elements A.3 and A.4 are unrelated.
// Elements B.0 and B.4 are unrelated.

try testing.expect(r1.isConsistent());

// Find the B related to A.1, i.e. B.2
const a_1 = a.selectInitialInstance(.id1);
const b_related = a_1.traverseAssociation(r1, null, .b);
try testing.expectEqual(@as(u32, 1), b_related.cardinality());
try testing.expect(b_related.contains(try b.selectByIdentifier(0, .{ .id = 2 })));

// Find the A related to B.3, i.e. A.2

```

```

const b_3 = b.selectInitialInstance(.id3);
const a_related = b_3.traverseAssociation(r1, null, .a);
try testing.expectEqual(@as(usize, 1), a_related.cardinality());
try testing.expect(a_related.contains(try a.selectByIdentifier(0, {.id = 2 })));

// Find the unrelated B's (red B's), i.e. B.0 and B.4
const red_b_set = r1.traverse(a.selectAll(), null, .b).complement();
try testing.expectEqual(@as(usize, 2), red_b_set.cardinality());
try testing.expect(red_b_set.contains(try b.selectByIdentifier(0, {.id = 0 })));
try testing.expect(red_b_set.contains(try b.selectByIdentifier(0, {.id = 4 })));

// Find the unrelated A's (red A's), i.e. A.3 and A.4
const red_a_set = r1.traverse(b.selectAll(), null, .a).complement();
try testing.expectEqual(@as(usize, 2), red_a_set.cardinality());
try testing.expect(red_a_set.contains(try a.selectByIdentifier(0, {.id = 3 })));
try testing.expect(red_a_set.contains(try a.selectByIdentifier(0, {.id = 4 })));
}

```

At-Least-One to At-Most-One

At least one-to-at most one (+--?) associations are surjective partial functions. This means that some domain elements are disassociated (i.e. there can be red A's), all B's are associated to at least one A, and multiple A's may be associated to the same B.

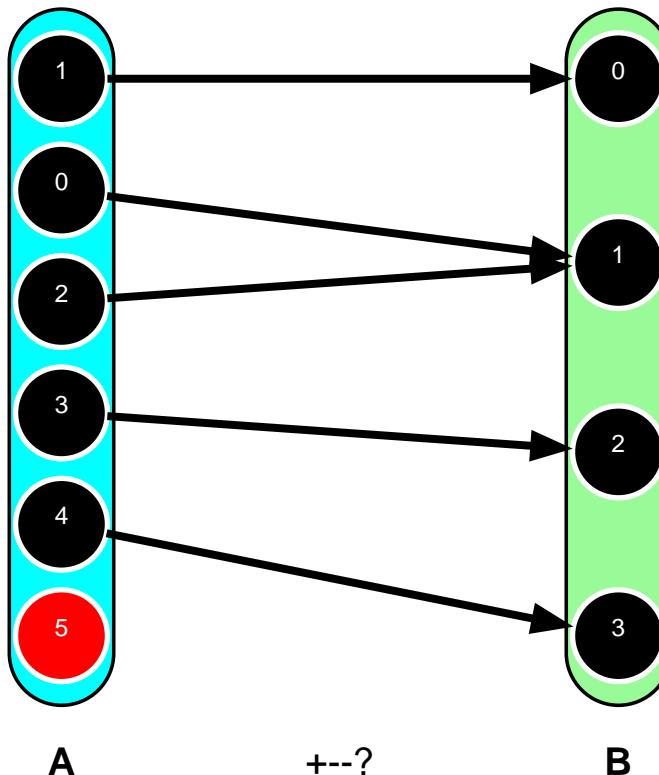


Figure 47. At least 1 to at most 1 association as a set diagram

```

<<association module tests>>=
test "at-least-one to at-most-one" {
    const A = RelClass(
        .a,
        struct { id: u8, which: u8 = 'A' },
        &.{&.{.id}},
        &.{},
        .{},
        20,
        &.{.
            .{ .id0, .{.id = 0} },
            .{ .id1, .{.id = 1} },
            .{ .id2, .{.id = 2} },
            .{ .id3, .{.id = 3} },
            .{ .id4, .{.id = 4} },
            .{ .id5, .{.id = 5} },
        },
    );
    const a = A.init(0);

    const B = RelClass(
        .b,
        struct { id: u8, which: u8 = 'B' },
        &.{&.{.id}},
        &.{},
        .{},
        20,
        &.{.
            .{ .id0, .{.id = 0} },
            .{ .id1, .{.id = 1} },
            .{ .id2, .{.id = 2} },
            .{ .id3, .{.id = 3} },
        },
    );
    const b = B.init(1);

    // A +--? B
    // A +- "passive" -R1- "active"-? B
    const r1_spec = AssociationSpec{
        .DomainRelClass = A,
        .codomain_ref = .@"+",
        .codomain_phrase = .passive,
        .assoc_tag = .r1,
        .domain_phrase = .active,
        .domain_ref = .@"?",
        .CodomainRelClass = B,
    };
    const R1 = Association(&r1_spec, &.{});
    const r1: R1 = .{};

    // Relate 5 of the domain elements.
    // A.5 is unrelated.
}

```

```

// All 4 of the codomain elements must be related.

// A.1 -> B.0
r1.relate(
    try a.selectByIdentifier(0, .{ .id = 1 }),
    .active,
    try b.selectByIdentifier(0, .{ .id = 0 }),
    null,
);
// A.0, A.2 -> B.1
r1.relate(
    try a.selectByIdentifier(0, .{ .id = 0 }),
    .active,
    try b.selectByIdentifier(0, .{ .id = 1 }),
    null,
);
r1.relate(
    try a.selectByIdentifier(0, .{ .id = 2 }),
    .active,
    try b.selectByIdentifier(0, .{ .id = 1 }),
    null,
);
// A.3 -> B.2
r1.relate(
    try a.selectByIdentifier(0, .{ .id = 3 }),
    .active,
    try b.selectByIdentifier(0, .{ .id = 2 }),
    null,
);
// A.4 -> B.3
r1.relate(
    try a.selectByIdentifier(0, .{ .id = 4 }),
    .active,
    try b.selectByIdentifier(0, .{ .id = 3 }),
    null,
);

try testing.expect(r1.isConsistent());

// Find the B related to A.1, i.e. B.0
const a_1 = try a.selectByIdentifier(0, .{ .id = 1 });
const b_related = a_1.traverseAssociation(r1, null, .b);
try testing.expectEqual(@as(usize, 1), b_related.cardinality());
try testing.expect(b_related.contains(try b.selectByIdentifier(0, .{ .id = 0 })));

// Find the A's related to B.1, i.e. A.0 and A.2
// There is always at least one and in this case there are 2.
const b_1 = try b.selectByIdentifier(0, .{ .id = 1 });
const a_related = b_1.traverseAssociation(r1, null, .a);
try testing.expectEqual(@as(usize, 2), a_related.cardinality());
try testing.expect(a_related.contains(try a.selectByIdentifier(0, .{ .id = 0 })));
try testing.expect(a_related.contains(try a.selectByIdentifier(0, .{ .id = 2 })));

```

```

// Find the unrelated B's (red B's).
// (there are none since the association is surjective).
const red_b_set = r1.traverse(a.selectAll(), null, .b).complement();
try testing.expectEqual(@as(usize, 0), red_b_set.cardinality());

// Find the unrelated A's (red A's), i.e. A.5
const red_a_set = r1.traverse(b.selectAll(), null, .a).complement();
try testing.expectEqual(@as(usize, 1), red_a_set.cardinality());
try testing.expect(red_a_set.contains(try a.selectByIdentifier(0, {.id = 5})));

// Unrelate A.2 from B.1
// This leaves the association consistent because
// A.0 is still associated with B.1.
const a_2 = try a.selectByIdentifier(0, {.id = 2});
r1.unrelate(a_2, .active, b_1, null);
try testing.expect(r1.isConsistent());

// Unrelate A.4 from B.3
// This makes the association referentially inconsistent
// because only A.4 was related to B.3.
const a_4 = try a.selectByIdentifier(0, {.id = 4});
const b_3 = try b.selectByIdentifier(0, {.id = 3});
r1.unrelate(a_4, .active, b_3, null);
try testing.expect(!r1.isConsistent());

// Consistency is restored if we relate A.5 to B.3
// B.3 (the one just unrelated that caused the inconsistency).
// This leaves A.4 unrelated (now a red A), but that is allowed for
// this multiplicity and conditionality.
const a_5 = try a.selectByIdentifier(0, {.id = 5});
r1.relate(a_5, .active, b_3, null);
try testing.expect(r1.isConsistent());

// We can get back to the original associations by re-relating
// B.3 from A.5 back to A.4.
r1.rerelate(b_3, .passive, a_5, a_4);
try testing.expect(r1.isConsistent());
}

```

Any to At-Most-One

Any to at-most-one (*--?) associations are non-surjective partial functions. This means that some of the A's may be unrelated, some of the B's may be unrelated, and multiple A's may be related to the same B.

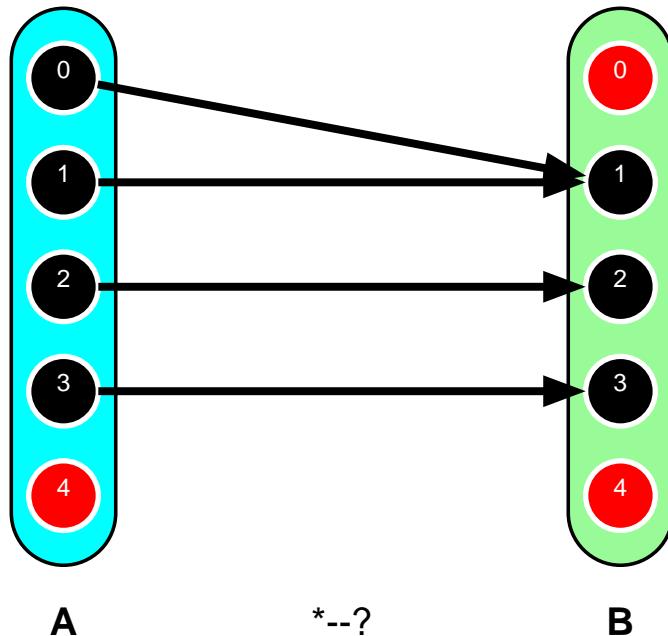


Figure 48. Any to at most 1 association as a set diagram

```
<<association module tests>>=
test "any to at-most-one" {
    const A = RelClass(
        .a,
        struct { id: u8, which: u8 = 'A' },
        &.{&{.id}},
        &{},
        .{},
        20,
        &{
            .{ .id0, .{.id = 0} },
            .{ .id1, .{.id = 1} },
            .{ .id2, .{.id = 2} },
            .{ .id3, .{.id = 3} },
            .{ .id4, .{.id = 4} },
        },
    );
    const a = A.init(0);

    const B = RelClass(
        .b,
        struct { id: u8, which: u8 = 'B' },
        &.{&{.id}},
        &{},
        .{},
        20,
        &{
            .{ .id0, .{.id = 0} },

```

```

        .{ .id1, .{.id = 1} },
        .{ .id2, .{.id = 2} },
        .{ .id3, .{.id = 3} },
        .{ .id4, .{.id = 4} },
    },
);
const b = B.init(1);

// A *--? B
// A *-"passive" -R1- "active"-? B
const r1_spec = AssociationSpec{
    .DomainRelClass = A,
    .codomain_ref = .@"",
    .codomain_phrase = .passive,
    .assoc_tag = .r1,
    .domain_phrase = .active,
    .domain_ref = .@"",
    .CodomainRelClass = B,
};
const R1 = Association(&r1_spec, &.{});
const r1: R1 = .{};

// Relate 4 of the available A's.
// A.4 is unrelated
// Relate 3 of the available B's.
// B.0 and B.4 are unrelated.
r1.relateBySelection(
    .active,
    &.{.
        .{ .{.id = 0}, .{.id = 1} }, // A.0 -> B.1
        .{ .{.id = 1}, .{.id = 1} }, // A.1 -> B.1
        .{ .{.id = 2}, .{.id = 2} }, // A.2 -> B.2
        .{ .{.id = 3}, .{.id = 3} }, // A.3 -> B.3
    },
) catch unreachable;

try testing.expect(r1.isConsistent());

// Find the B related to A.1
const a_1 = try a.selectByIdentifier(0, .{ .id = 1 });
const b_related = a_1.traverseAssociation(r1, null, .b);
try testing.expectEqual(@as(usize, 1), b_related.cardinality());
try testing.expect(b_related.contains(try b.selectByIdentifier(0, .{ .id = 1 })));

// Find the A's related to B.1
const b_1 = try b.selectByIdentifier(0, .{ .id = 1 });
const a_related = b_1.traverseAssociation(r1, null, .a);
try testing.expectEqual(@as(usize, 2), a_related.cardinality());
try testing.expect(a_related.contains(try a.selectByIdentifier(0, .{ .id = 0 })));
try testing.expect(a_related.contains(try a.selectByIdentifier(0, .{ .id = 1 })));

// Find the B's not related to any A (red B's).

```

```

const red_b_set = r1.traverse(a.selectAll(), null, .b).complement();
try testing.expectEqual(@as(usize, 2), red_b_set.cardinality());
try testing.expect(red_b_set.contains(try b.selectByIdentifier(0, {.id = 0 })));
try testing.expect(red_b_set.contains(try b.selectByIdentifier(0, {.id = 4 })));

// Find the A's not related to any B (red A's).
const red_a_set = r1.traverse(b.selectAll(), null, .a).complement();
try testing.expectEqual(@as(usize, 1), red_a_set.cardinality());
try testing.expect(red_a_set.contains(try a.selectByIdentifier(0, {.id = 4 })));

// Relate A.1 to B.3
// This makes A.1 refer to multiple B's and that violates consistency.
const b_3 = try b.selectByIdentifier(0, {.id = 3 });
r1.relate(a_1, .active, b_3, null);
try testing.expect(!r1.isConsistent());
}

```

At-Least-One to At-Least-One

The next three conditionality and multiplicity cases are set-valued functions. This means that the image of the domain under the relation can return a set of cardinality greater than one.

At-least-one to at-least-one (+++) associations are surjective set-valued functions. This means all elements are related, that domain elements may refer to multiple codomain elements, and codomain elements may be associated to multiple domain elements. The domain and codomain elements are closely associated. In the degenerate case where all elements are associated to each other, the binary relation is the Cartesian product of the domain and codomain and the adjacency matrix contains *true* in every cell.

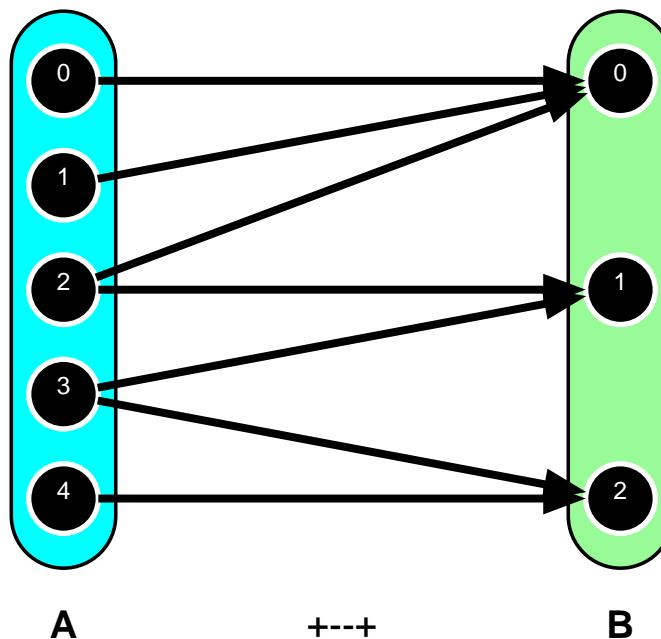


Figure 49. At least 1 to at least 1 association as a set diagram

```

<<association module tests>>=
test "at-least-one to at-least-one" {
    const A = RelClass(
        .a,
        struct { id: u8, which: u8 = 'A' },
        &.{{&.{.id}}},
        &.{},
        .{ },
        20,
        &.{
            .{ .id0, .{ .id = 0, .which = 'A' } },
            .{ .id1, .{ .id = 1, .which = 'A' } },
            .{ .id2, .{ .id = 2, .which = 'A' } },
            .{ .id3, .{ .id = 3, .which = 'A' } },
            .{ .id4, .{ .id = 4, .which = 'A' } },
        },
    );
    const a = A.init(0);

    const B = RelClass(
        .b,
        struct { id: u8, which: u8 = 'B' },
        &.{{&.{.id}}},
        &.{},
        .{ },
        20,
        &.{
            .{ .id0, .{ .id = 0, .which = 'B' } },
            .{ .id1, .{ .id = 1, .which = 'B' } },
            .{ .id2, .{ .id = 2, .which = 'B' } },
        },
    );
    const b = B.init(1);

    // A +-"passive" -R1- "active"--+ B
    const r1_spec = AssociationSpec{
        .DomainRelClass = A,
        .codomain_ref = .@"+",
        .codomain_phrase = .passive,
        .assoc_tag = .r1,
        .domain_phrase = .active,
        .domain_ref = .@"+",
        .CodomainRelClass = B,
    };
    const R1 = Association(
        &r1_spec,
        // Relate all the elements. To be consistent all elements must be somehow related.
        &.{.
            // A.0, A.1, A.2 -> B.0
            .{ .id0, .id0},
            .{ .id1, .id0},
            .{ .id2, .id0},
        }
    );
}

```

```

        // A.2 -> B.1
        .{ .id2, .id1},
        // A.3 -> B.1, B.2
        .{ .id3, .id1},
        .{ .id3, .id2},
        // A.4 -> B.2
        .{ .id4, .id2},
    },
);
const r1: R1 = .{};

try testing.expect(r1.isConsistent());

// Find the B's (there's always at least one in this case)
// related to A.2
const a_2 = a.selectInitialInstance(.id2);
const b_related = a_2.traverseAssociation(r1, null, .b);
try testing.expectEqual(@as(usize, 2), b_related.cardinality());
try testing.expect(b_related.contains(try b.selectByIdentifier(0, .{ .id = 0 })));
try testing.expect(b_related.contains(try b.selectByIdentifier(0, .{ .id = 1 })));

// Find the A's related to B.0
const b_0 = b.selectInitialInstance(.id0);
const a_related = b_0.traverseAssociation(r1, null, .a);
try testing.expectEqual(@as(usize, 3), a_related.cardinality());
try testing.expect(a_related.contains(try a.selectByIdentifier(0, .{ .id = 0 })));
try testing.expect(a_related.contains(try a.selectByIdentifier(0, .{ .id = 1 })));
try testing.expect(a_related.contains(try a.selectByIdentifier(0, .{ .id = 2 })));

// Unrelate A.3 from B.1
// The relation remains consistent because A.3 remains associated
// to a B, namely B.2.
r1.unrelate(
    a.selectInitialInstance(.id3),
    .active,
    b.selectInitialInstance(.id1),
    null,
);
try testing.expect(r1.isConsistent());

// Unrelate A.4 from B.2.
// This makes the association referentially inconsistent because
// A.4 was only associated to a single B (i.e. unrelating turns A red).
r1.unrelate(
    a.selectInitialInstance(.id4),
    .active,
    b.selectInitialInstance(.id2),
    null,
);
try testing.expect(!r1.isConsistent());
}

```

At-Least-One to Any

At-least-one to any (+--*) associations are surjective set-valued partial functions. This means that all codomain elements are associated to one or more domain elements (i.e. surjective). But, a domain element may *not* be associated to any codomain element (i.e. partial function).

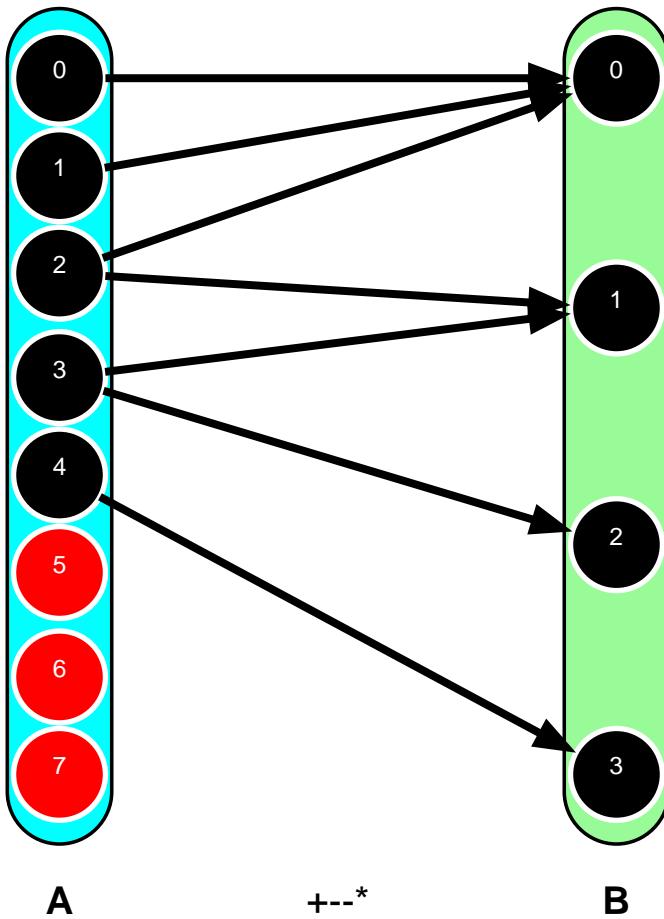


Figure 50. At least 1 to any association as a set diagram

```
<<association module tests>>=
test "at-least-one to any" {
    const A = RelClass(
        .a,
        struct {
            id: u8,
            which: u8 = 'A',
        },
        &.{&.{.id}},
        &.{},
        .{},
        20,
        &.{.
            .{ .id0, .{ .id = 0 } },
            .{ .id1, .{ .id = 1 } },
            .{ .id2, .{ .id = 2 } },
            .{ .id3, .{ .id = 3 } },
            .{ .id4, .{ .id = 4 } },
            .{ .id5, .{ .id = 5 } },
            .{ .id6, .{ .id = 6 } },
            .{ .id7, .{ .id = 7 } },
        }
    )
}
```

```

        .{ .id2, .{ .id = 2 } },
        .{ .id3, .{ .id = 3 } },
        .{ .id4, .{ .id = 4 } },
        .{ .id5, .{ .id = 5 } },
        .{ .id6, .{ .id = 6 } },
        .{ .id7, .{ .id = 7 } },
    },
);
const a = A.init(0);

const B = RelClass(
    .b,
    struct {
        id: u8,
        which: u8 = 'B',
    },
    &.{&.{.id}},
    &.{},
    .{},
    20,
    &.{{
        .{ .id0, .{ .id = 0 } },
        .{ .id1, .{ .id = 1 } },
        .{ .id2, .{ .id = 2 } },
        .{ .id3, .{ .id = 3 } },
    }},
),
const b = B.init(1);

// A +-* B
// A +- "passive" -R1- "active"-* B
const r1_spec = AssociationSpec{
    .DomainRelClass = A,
    .codomain_ref = .@"+",
    .codomain_phrase = .passive,
    .assoc_tag = .r1,
    .domain_phrase = .active,
    .domain_ref = .@"*",
    .CodomainRelClass = B,
};
const R1 = Association(&r1_spec, &{
    // Relate a subset of A's, while making some of the
    // A's related to the same B.
    // Note we also relate several A's to the same B.
    .{ .id0, .id0 },
    .{ .id1, .id0 },
    .{ .id2, .id0 },
    .{ .id2, .id1 },
    .{ .id3, .id1 },
    .{ .id3, .id2 },
    .{ .id4, .id3 },
    // A.5, A.6, and A.7 are unrelated.
}

```

```

});  

const r1: R1 = .{};  
  

try testing.expect(r1.isConsistent());  
  

// Find the B's (there's always at least one in this case)  

// related to A.2  

const a_2 = a.selectInitialInstance(.id2);  

const b_related = a_2.traverseAssociation(r1, null, .b);  

try testing.expectEqual(@as(usize, 2), b_related.cardinality());  

try testing.expect(b_related.contains(try b.selectByIdentifier(0, .{ .id = 0 })));  

try testing.expect(b_related.contains(try b.selectByIdentifier(0, .{ .id = 1 })));  
  

// Find the A's related to B.3  

const b_3 = b.selectInitialInstance(.id3);  

const a_related = b_3.traverseAssociation(r1, null, .a);  

try testing.expectEqual(@as(usize, 1), a_related.cardinality());  

try testing.expect(a_related.contains(try a.selectByIdentifier(0, .{ .id = 4 })));  
  

// Find the unrelated A's (red A's).  

const a_unrelated = r1.traverse(b.selectAll(), null, .a).complement();  

try testing.expectEqual(@as(usize, 3), a_unrelated.cardinality());  

try testing.expect(a_unrelated.contains(try a.selectByIdentifier(0, .{ .id = 5 })));  

try testing.expect(a_unrelated.contains(try a.selectByIdentifier(0, .{ .id = 6 })));  

try testing.expect(a_unrelated.contains(try a.selectByIdentifier(0, .{ .id = 7 })));  
  

// Unrelate A.4 from B.3  

// This causes the relation to be inconsistent because no A  

// is associated to B.3 (i.e. B.3 turns red).  

const a_4 = a.selectInitialInstance(.id4);  

r1.unrelate(a_4, .active, b_3, null);  

try testing.expect(!r1.isConsistent());
}

```

Any to Any

Any to any (*--*) associations are non-surjective set-valued partial functions. This multiplicity and conditionality is the weakest constraint that can be placed on an association. Any arrangement of domain and codomain elements ends up being referentially consistent.

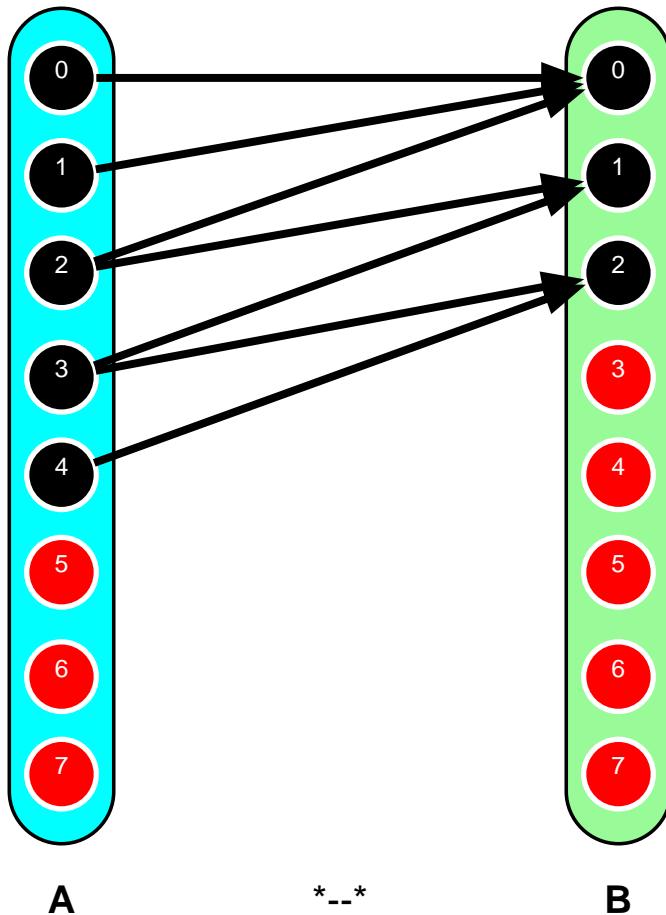


Figure 51. Any to any association as a set diagram

```
<<association module tests>>=
test "any to any" {
    const A = RelClass(
        .a,
        struct { id: u8, which: u8 = 'A' },
        &.{&.id}),
        &.{},
        .{},
        20,
        &.{ 
            .{ .id0, .{ .id = 0 }},
            .{ .id1, .{ .id = 1 }},
            .{ .id2, .{ .id = 2 }},
            .{ .id3, .{ .id = 3 }},
            .{ .id4, .{ .id = 4 }},
            .{ .id5, .{ .id = 5 }},
            .{ .id6, .{ .id = 6 }},
            .{ .id7, .{ .id = 7 }},
        },
    );
}
```

```

const a = A.init(0);

const B = RelClass(
    .b,
    struct { id: u8, which: u8 = 'B' },
    &.{&.{.id}},
    &.{},
    .{ },
    20,
    &.{{
        .{ .id0, .{ .id = 0 }},
        .{ .id1, .{ .id = 1 }},
        .{ .id2, .{ .id = 2 }},
        .{ .id3, .{ .id = 3 }},
        .{ .id4, .{ .id = 4 }},
        .{ .id5, .{ .id = 5 }},
        .{ .id6, .{ .id = 6 }},
        .{ .id7, .{ .id = 7 }},
    }},
);
const b = B.init(1);

// A *-"passive" -R1- "active"-* B
const r1_spec = AssociationSpec{
    .DomainRelClass = A,
    .codomain_ref = .@"",
    .codomain_phrase = .passive,
    .assoc_tag = .r1,
    .domain_phrase = .active,
    .domain_ref = .@"",
    .CodomainRelClass = B,
};
const R1 = Association(&r1_spec, &{  

    // Relate a subset of A's, while making some of the  

    // A's related to the same B.  

    .{ .id0, .id0 },
    .{ .id1, .id0 },
    .{ .id2, .id0 },
    .{ .id2, .id1 },
    .{ .id3, .id1 },
    .{ .id3, .id2 },
    .{ .id4, .id2 },
});
const r1: R1 = .{};

// Note this test is superfluous. There is no way to violate
// referential consistency for an binary relation that is any-to-any.
try testing.expect(r1.isConsistent());

// Find the B's related to A.1
const a_1 = a.selectInitialInstance(.id1);
const b_related = a_1.traverseAssociation(r1, null, .b);

```

```

try testing.expectEqual(@as(usize, 1), b_related.cardinality());
try testing.expect(b_related.contains(try b.selectByIdentifier(0, {. .id = 0 })));

// Find the A's related to B.1
const b_1 = b.selectInitialInstance(.id1);
const a_related = b_1.traverseAssociation(r1, null, .a);
try testing.expectEqual(@as(usize, 2), a_related.cardinality());
try testing.expect(a_related.contains(try a.selectByIdentifier(0, {. .id = 2 })));
try testing.expect(a_related.contains(try a.selectByIdentifier(0, {. .id = 3 })));

// Find the A's unrelated to B.1.
var a_unrelated = b_1.traverseAssociation(r1, null, .a).complement();
try testing.expectEqual(@as(usize, 6), a_unrelated.cardinality());
try testing.expect(a_unrelated.contains(try a.selectByIdentifier(0, {. .id = 0 })));
try testing.expect(a_unrelated.contains(try a.selectByIdentifier(0, {. .id = 1 })));
try testing.expect(a_unrelated.contains(try a.selectByIdentifier(0, {. .id = 4 })));
try testing.expect(a_unrelated.contains(try a.selectByIdentifier(0, {. .id = 5 })));
try testing.expect(a_unrelated.contains(try a.selectByIdentifier(0, {. .id = 6 })));
try testing.expect(a_unrelated.contains(try a.selectByIdentifier(0, {. .id = 7 })));

// Find all the unrelated A's (red A's).
a_unrelated = r1.traverse(b.selectAll(), null, .a).complement();
try testing.expectEqual(@as(usize, 3), a_unrelated.cardinality());
try testing.expect(a_unrelated.contains(try a.selectByIdentifier(0, {. .id = 5 })));
try testing.expect(a_unrelated.contains(try a.selectByIdentifier(0, {. .id = 6 })));
try testing.expect(a_unrelated.contains(try a.selectByIdentifier(0, {. .id = 7 })));

// Find all the unrelated B's (red B's).
const b_unrelated = r1.traverse(a.selectAll(), null, .b).complement();
try testing.expectEqual(@as(usize, 5), b_unrelated.cardinality());
try testing.expect(b_unrelated.contains(try b.selectByIdentifier(0, {. .id = 3 })));
try testing.expect(b_unrelated.contains(try b.selectByIdentifier(0, {. .id = 4 })));
try testing.expect(b_unrelated.contains(try b.selectByIdentifier(0, {. .id = 5 })));
try testing.expect(b_unrelated.contains(try b.selectByIdentifier(0, {. .id = 6 })));
try testing.expect(b_unrelated.contains(try b.selectByIdentifier(0, {. .id = 7 })));
}

```

Code layout

```

<<association.zig>>=
/// The `association.zig` module implements the concept of a
/// *Relational Class Association*. An Association is a relationship between two
/// Relation Class types. It specifies the multiplicity and conditionality constraints
/// on the instances of the participating classes. An Association may be _navigated_
/// between its participating Relational Classes. This is logically equivalent to
/// applying the binary relation that the Association represents.
<<edit warning>>
<<copyright info>>

const std = @import("std");
const assert = std.debug.assert;

```

```

const meta = std.meta;
const EnumField = std.builtin.Type.EnumField;

const builtin = @import("builtin");
const testing = if (builtin.os.tag == .freestanding)
    @import("resee_testing")
else
    std.testing;

const state_model = @import("state_model.zig");
const StateModelSpec = state_model.StateModelSpec;
const assigner = @import("assigner.zig");
const Assigner = assigner.Assigner;
const MultiAssigner = assigner.MultiAssigner;

pub const version = "1.0.0-a1";

<<association module declarations>>

<<association module public functions>>

<<association module private functions>>

<<association module tests>>

```

Just My Type

In this section, we show the implementation of a generalization, the other form of relationship between relational classes. A generalization represents the notion of separating a relation class into distinct types. The subclasses of a generalization represent an equivalence class giving precise rules how the superclass is partitioned.

Representing a Generalization

A Generalization is a relationship between RelClass entities that is implemented with adjacency matrices. We import the required modules.

```

<<generalization module public functions>>=
const class = @import("class.zig");
const RelClass = class.RelClass;
const isRelClassType = class.isRelClassType;
const ClassGeneratedId = class.ClassGeneratedId;
const generated_id_default = class.generated_id_default;

const binary_relation = @import("binary_relation.zig");
const GeneralizationAdjMatrix = binary_relation.GeneralizationAdjMatrix;

```

Using our established pattern, we define a type function to create the Generalization data type.

```

<<generalization module public functions>>=
/// The `Generalization` function is a type function that generates a namespace type
/// for a generalization relationship.
/// The `GenSuperclass` argument is a namespace type as returned
/// from the `RelClass` function and is the data type of the superclass (codomain) of the
/// generalization. The `GenSubclasses` argument is a slice of namespace types representing
/// the subclasses (partitioned domain) of the generalization.
pub fn Generalization(
    comptime gen_tag: @Type(.enum_literal),
    comptime GenSuperclass: type,
    comptime GenSubclasses: []const type,
    comptime initial_gens: []const struct {
        @Type(.enum_literal), // partition tag
        @Type(.enum_literal), // InitialInstanceId tag value corresponding to the partition
        GenSuperclass.InitialInstanceId, // InitialInstanceId enum value for the superclass
    },
) type {
    if (!isRelClassType(GenSuperclass))
        @compileError("GenSuperclass argument is not a RelClass");

    validateGeneralizationSubclasses(GenSuperclass, GenSubclasses);

    return struct {
        const Self = @This();

        <<generalization variables>>
        <<generalization declarations>>
        <<generalization public functions>>
        <<generalization private functions>>
    };
}

```

We enforce some rules on the arguments to `Generalization` to catch invalid input arguments and give a better error message.

```

<<generalization module private functions>>=
/// The `validateGeneralizationSubclasses` function examines domain specifiers
/// given by the `subclass_specs` argument to ensure that partitions are proper
/// `RelClass` types and that no subclass type is the same as the superclass.
/// specifier to ensure that
fn validateGeneralizationSubclasses(
    comptime Super: type,
    comptime Subs: []const type,
) void {
    if (Subs.len < 2)
        @compileError("generalization relationships require at least two subclasses");

    inline for (Subs) |Subclass| {
        if (!isRelClassType(Subclass))
            @compileError("subclass, " ++ @tagName(Subclass.class_id) ++ " is not a

```

```

RelClass");

    if (Subclass == Super)
        @compileError(
            "a subclass may not be the same class as its superclass: '" ++
            @tagName(Super.class_id) ++
            ";");
}
}

```

```

<<generalization variables>>=
/// The data type of the adjacency matrix for the generalization.
pub const GenAdjMatrix = DefineGenAdjMatrix();
pub const gen_adj_matrix: GenAdjMatrix = .{};


```

To create the required generalization adjacency matrix, we must create arguments to the GeneralizationAdjMatrix type function. The adjacency matrix is only concerned with the various counts of the set elements that are to be related. The following function creates the necessary capacity arguments.

```

<<generalization private functions>>=
/// The `DefineGenAdjMatrix` function is a type function which returns an
/// adjacency matrix type which is suitable for tracking the generalization
/// given by the `SuperclassType` and `subclass_specs` arguments.
fn DefineGenAdjMatrix() type {
    const PartitionDesc = struct { @Type(.enum_literal), type };
    var partition_desc: []const PartitionDesc = &[_]PartitionDesc{};
    inline for (GenSubclasses) |Subclass| {
        partition_desc = partition_desc ++ &[_]PartitionDesc{
            .{ Subclass.class_id, Subclass.ClassVar }, ①
        };
    }
    return GeneralizationAdjMatrix(GenSuperclass.ClassVar, partition_desc, initial_gens);
}


```

- ① The names given to the domain partitions of the adjacency matrices is the same set of names that are attributed to the subclasses of the generalization relationships.

A generalization adjacency matrix defines an enumerated type corresponding to the names given to the domain partitions. We want to use the enumerated type defined by the adjacency matrix for identifying partitions as the same identifier for subclasses.

```

<<generalization declarations>>=
pub const SubclassId = GenAdjMatrix.PartitionId;


```

```

<<generalization declarations>>=
pub const gen_id = gen_tag;


```

```
pub var gen_number: u8 = undefined;
```

```
<<generalization public functions>>=
pub fn init(
    number: u8,
) Self {
    gen_number = number;
    return .{};
}
```

```
<<generalization module declarations>>=
pub const TraversalPhrase = enum(u1) {
    from_subclass, // subclass to superclass
    to_subclass, // superclass to subclass
};
```

Following our previous practice, we include some meta-information in the result type that enables checking if a given type is indeed an Association.

```
<<generalization variables>>=
pub const is_generalization: void = {};
```

```
<<generalization module public functions>>=
/// The `isGeneralizationType` function returns `true` if the value of `GeneralizationType`
/// argument was generated by the `Generalization` type function and `false` otherwise.
pub fn isGeneralizationType(
    comptime GeneralizationType: type,
) bool {
    return @typeInfo(GeneralizationType) == .@("struct" and
        @hasDecl(GeneralizationType, "is_generalization"));
}
```

Tests

The following generalization tests uses the r1 generalization from a [previous diagram](#). Since we use the same set of classes and generalization description, we factor them into a literate program chunk.

```
<<generalization test setup>>=
const Vehicle = RelClass(
    .vehicle,
    struct {
        id: ClassGeneratedId = generated_id_default,
        make: []const u8,
        model: []const u8,
    },
    &.{&.{.id}},
    &.{},
);
```

```

    .{},

    10,
    &.{},
);

const Car = RelClass(
    .car,
    struct { id: ClassGeneratedId, door_count: u8 },
    &.{&.{.id}},
    &.{},
    .{},

    5,
    &.{},
);

const Truck = RelClass(
    .truck,
    struct { id: ClassGeneratedId, gross_weight: u32 },
    &.{&.{.id}},
    &.{},
    .{},

    5,
    &.{},
);

const Bus = RelClass(
    .bus,
    struct { id: ClassGeneratedId, passenger_capacity: u8 },
    &.{&.{.id}},
    &.{},
    .{},

    10,
    &.{},
);

const R1 = Generalization(
    .r1,
    Vehicle,
    &.{ Car, Truck, Bus },
    &.{},
);

```

```

<<generalization module tests>>=
test "generalization initialization" {
    <<generalization test setup>>

    try testing.expectEqual(@as(usize, 3), enums.values(R1.SubclassId).len);
}

```

Forming Generalizations Between Superclass and Subclass Instances

A generalization relationship as defined here, relates `RelClass` typed variables. We carry forward the `InstanceSet` notion used by `RelClass` variables in their function interfaces. Recall, an `InstanceSet` uses a bit-set as a means to reference 0 or more relvar tuple sets.

```
<<generalization declarations>>=
pub const Superclass = GenSuperclass;
pub const SuperclassInstanceSet = Superclass.InstanceSet;
pub const SuperclassInstanceRef = Superclass.InstanceRef;
pub const SuperclassTupleRefSet = Superclass.TupleRefSet;
```

Of necessity, the `InstanceSet` data type for a subclass is known by its `SubclassId`.

```
<<generalization declarations>>=
pub const Subclasses = GenSubclasses;

/// The `SubclassInstanceSet` function returns the `InstanceSet` data type
/// for the subclass identified by the `subclass_id` argument.
pub fn SubclassInstanceSet(
    comptime subclass_id: SubclassId,
) type {
    return GenSubclasses[@intFromEnum(subclass_id)].InstanceSet;
}

/// The `SubclassInstanceRef` function returns the `InstanceRef` data type
/// for the subclass identified by the `subclass_id` argument.
pub fn SubclassInstanceRef(
    comptime subclass_id: SubclassId,
) type {
    return GenSubclasses[@intFromEnum(subclass_id)].InstanceRef;
}

/// The `SubclassTupleRefSet` function returns the `TupleRefSet` data type
/// for the subclass identified by the `subclass_id` argument.
pub fn SubclassTupleRefSet(
    comptime subclass_id: SubclassId,
) type {
    return GenSubclasses[@intFromEnum(subclass_id)].TupleRefSet;
}
```

The pattern of the relate and unrelated functions follows that used for Association relationships.

```
<<generalization public functions>>=
/// The `relate` function modifies the given generalization so as to
```

```

/// bind the single instance from the `subclass_inst` instance set whose
/// type is given by `subclass_id` to the single instance given by the
/// `superclass_inst` and _vice versa_. Note that given the nature of a
/// generalization relationship, the cardinality of both `subclass_inst`
/// and `superclass_inst` must be one.
pub fn relate(
    _: Self,
    comptime subclass_id: SubclassId,
    subclass_inst_ref: SubclassInstanceRef(subclass_id),
    superclass_inst_ref: SuperclassInstanceRef,
) void {
    gen_adj_matrix.relate(subclass_id, subclass_inst_ref.tuple_ref, superclass_inst_ref
    .tuple_ref);
}

```

```

<<generalization public functions>>=
/// The `unrelate` function modifies the given generalization so as to
/// unbind the single instance from the `subclass_inst` instance set whose
/// type is given by `subclass_id` to the single instance given by the
/// `superclass_inst` and _vice versa_. Note that given the nature of a
/// generalization relationship, the cardinality of both `subclass_inst`
/// and `superclass_inst` must be one.
pub fn unrelate(
    _: Self,
    comptime subclass_id: SubclassId,
    subclass_inst: SubclassInstanceRef(subclass_id),
    superclass_inst: SuperclassInstanceRef,
) void {
    gen_adj_matrix.unrelate(subclass_id, subclass_inst.tuple_ref, superclass_inst.
    tuple_ref);
}

```

As with associations, there are use cases where a superclass or subclass instance is known and it needs to be unrelated from its corresponding instance across the generalization. Two convenience functions are supplied.

```

<<generalization public functions>>=
/// The `unrelateSuperclassInstances` function modifies the
/// given generalization so as to unbind all the instances in the
/// `superclass_insts` argument from their related subclass instances.
pub fn unrelateSuperclassInstances(
    _: Self,
    superclass_insts: anytype,
) void {
    const SuperclassInstsType = @TypeOf(superclass_insts);
    if (SuperclassInstsType == SuperclassInstanceRef) {
        gen_adj_matrix.unrelateCodomainTuple(superclass_insts.tuple_ref);
    }
}

```

```

} else if (SuperclassInstsType == SuperclassInstanceSet) {
    var codomain_iter = superclass_insts.iterator();
    while (codomain_iter.next()) |codomain_tuple| {
        gen_adj_matrix.unrelateCodomainTuple(codomain_tuple.tuple_ref);
    }
} else {
    @compileError("unexpected type for superclass_insts: " ++
        @typeName(SuperclassInstsType));
}
}

```

```

<<generalization public functions>>=
/// The `unrelateSubclassInstances` function modifies the
/// given generalization so as to unbind all the instances in the
/// `subclass_insts` argument from their related superclass instances.
/// The subclass instances referenced by the `subclass_insts` argument
/// are identified by the value of the `subclass_id` argument.
pub fn unrelateSubclassInstances(
    _: Self,
    comptime subclass_id: SubclassId,
    subclass_insts: anytype,
) void {
    const SubclassInstsType = @TypeOf(subclass_insts);
    if (SubclassInstsType == SubclassInstanceRef(subclass_id)) {
        _ = gen_adj_matrix.unrelateDomainTuple(subclass_id, subclass_insts.tuple_ref);
    } else if (SubclassInstsType == SubclassInstanceSet(subclass_id)) {
        var domain_iter = subclass_insts.iterator();
        while (domain_iter.next()) |domain_tuple| {
            _ = gen_adj_matrix.unrelateDomainTuple(subclass_id, domain_tuple.tuple_ref);
        }
    } else {
        @compileError("unexpected type for subclass_insts: " ++
            @typeName(SubclassInstsType));
    }
}

```

Generalization Referential Consistency

As with **Associations**, we are interested in ensuring that a **Generalization** is referentially consistent in the face of processing that may have modified the underlying adjacency matrices. Like **Associations** it is necessary to know the ‘universe’ of tuples in the participating **RelClass** instances. In this usage, the universe is a bit-set that defines the subset of relvar tuples that are defined for the relvar underlying a **RelClass**. The only complication here is the need to gather up the universes for the subclasses and pass them as a slice to the adjacency matrix function that evaluates generalization consistency.

```

<<generalization public functions>>=

```

```

/// The `isConsistent` function returns `true` if the given `Generalization`
/// is referentially consistent and `false` otherwise.
pub fn isConsistent(
    _: Self,
) bool {
    return gen_adj_matrix.isGeneralizationConsistent();
}

```

Tests

```

<<generalization module tests>>
test "generalization consistency" {
    <<generalization test setup>>

    const vehicle = Vehicle.init(0);
    const car = Car.init(1);
    const truck = Truck.init(2);
    const bus = Bus.init(3);
    const r1: R1 = .{};

    const v1 = vehicle.create(.{
        .make = "Toyota",
        .model = "Prius",
    });
    const c1 = car.create(.{
        .id = v1.readAttribute(.id),
        .door_count = 4,
    });
    r1.relate(.car, c1, v1);
    try testing.expect(r1.isConsistent());
    try testing.expectEqual(.car, r1.classify(v1));

    r1.unrelateSubclassInstances(.car, c1);
    try testing.expect(!r1.isConsistent());
    c1.destroy();
    v1.destroy();
    try testing.expect(r1.isConsistent());

    const v2 = vehicle.create(.{
        .make = "Peterbuilt",
        .model = "P-1000",
    });
    const t2 = truck.create(.{
        .id = v2.readAttribute(.id),
        .gross_weight = 40_000,
    });
    r1.relate(.truck, t2, v2);
    try testing.expect(r1.isConsistent());
    try testing.expectEqual(.truck, r1.classify(v2));
}

```

```

const t3 = truck.create({.
    .id = generated_id_default,
    .gross_weight = 30_000,
});
r1.reclassify(.truck, t2, .truck, t3);
try testing.expect(!r1.isConsistent());
t2.destroy();
try testing.expect(r1.isConsistent());
try testing.expectEqual(.truck, r1.classify(v2));

const b1 = bus.create({.
    .id = generated_id_default,
    .passenger_capacity = 24,
});
r1.reclassify(.truck, t3, .bus, b1);
t3.destroy();
try testing.expect(r1.isConsistent());
try testing.expectEqual(.bus, r1.classify(v2));

r1.unrelateSuperclassInstances(v2);
try testing.expect(!r1.isConsistent());
b1.destroy();
v2.destroy();
try testing.expect(r1.isConsistent());
}

```

Handling Generalization Subclasses

The two function provided by the generalization adjacency matrix used to handle generalization subclasses are brought out as Generalization function which use InstanceSet typed arguments.

```

<<generalization public functions>>=
/// The `classify` function determines which subclass holds the subclass
/// instance related to `superclass_inst`. The cardinality of the
/// `superclass_inst` instance set must be one.
pub fn classify(
    _: Self,
    superclass_inst: SuperclassInstanceRef,
) SubclassId {
    return gen_adj_matrix.classify(superclass_inst.tuple_ref);
}

```

```

<<generalization public functions>>=
/// The `reclassify` function modifies the generalization so as to
/// unrelate the `prev_subclass_id` and `prev_subclass_inst` from its
/// related superclass instance and establish a new generalization
/// to the same superclass element given by `new_subclass_id`

```

```

/// and `new_subclass_inst`. Note the cardinality of both
/// `prev_subclass_inst` and `new_subclass_inst` must be one.
pub fn reclassify(
    _: Self,
    comptime prev_subclass_id: SubclassId,
    prev_subclass_inst: SubclassInstanceRef(prev_subclass_id),
    comptime new_subclass_id: SubclassId,
    new_subclass_inst: SubclassInstanceRef(new_subclass_id),
) void {
    gen_adj_matrix.reclassify(
        prev_subclass_id,
        prev_subclass_inst.tuple_ref,
        new_subclass_id,
        new_subclass_inst.tuple_ref,
    );
}

```

Generalization Traversal

Because of the disjoint union nature of a generalization relationship, traversal always requires knowledge of which subclass is involved.

When traversing from a subclass to a superclass, it is necessary to know from which subclass the traversal begins. The traversal to a superclass always yields exactly one superclass instance for each instance that participates in the traversal.

```

<<generalization public functions>>=
/// The `traverseFromSubclass` function traverses the given generalization
/// relationship from the subclass given by the `subclass_id` argument
/// using the instances given by the `subclass_insts` argument.
/// The return value is a superclass instance set of the related
/// superclass instances. Note that the cardinality of the returned
/// superclass instance set is always the same as the cardinality of the
/// `subclass_insts` instance set.
pub fn traverseFromSubclass(
    _: Self,
    comptime subclass_id: SubclassId,
    subclass_insts: SubclassInstanceSet(subclass_id),
) SuperclassInstanceSet {
    return .initFromTupleRefSet(gen_adj_matrix.image(subclass_id, subclass_insts
    .tuple_ref_set));
}

```

When traversing from a superclass to a subclass, it is necessary to know to which subclass the traversal should be directed. Traversal from a superclass to a subclass can yield no subclass instances for a particular subclass. Traversal from a superclass to all of the subclasses of the generalization always returns exactly one subclass instance for one of the subclasses and none for the remaining subclasses.

```

<<generalization public functions>>=
/// The `traverseToSubclass` function traverses the given generalization
/// relationship from the instances given by the `superclass_insts`
/// argument to the subclass identified by the `subclass_id` argument.
/// The return value is a subclass instance set of those subclass
/// instances that were related to any superclass instance in
/// `superclass_insts`. Note that the cardinality of the returned
/// subclass instance set may range from at most the cardinality of
/// `superclass_insts` down to zero.
pub fn traverseToSubclass(
    _: Self,
    comptime subclass_id: SubclassId,
    superclass_insts: SuperclassInstanceSet,
) SubclassInstanceSet(subclass_id) {
    return .initFromTupleRefSet(gen_adj_matrix.preimage(subclass_id, superclass_insts
    .tuple_ref_set));
}

```

```

<<generalization private functions>>=
pub fn InferDestRelClass(
    comptime phrase: TraversalPhrase,
    comptime subclass_id: SubclassId,
) type {
    return if (phrase == .from_subclass)
        Superclass
    else
        GenSubclasses[@intFromEnum(subclass_id)];
}

```

```

<<generalization public functions>>=
pub fn traverse(
    self: Self,
    comptime phrase: TraversalPhrase,
    comptime subclass_id: SubclassId,
    inst_set: anytype,
) InferDestRelClass(phrase, subclass_id).InstanceSet {
    return if (phrase == .to_subclass)
        self.traverseToSubclass(subclass_id, inst_set)
    else
        self.traverseFromSubclass(subclass_id, inst_set);
}

```

```

<<instance reference declarations>>=
pub fn traverseGeneralization(
    inst_ref: InstanceRef,
    comptime relationship: anytype,
    comptime phrase: generalization.TraversalPhrase,
    comptime subclass_id: @TypeOf(relationship).SubclassId,
) @TypeOf(relationship).InferDestRelClass(phrase, subclass_id).InstanceSet {

```

```

const inst_set = inst_ref.instanceSetFromInstanceRef();
return relationship.traverse(phrase, subclass_id, inst_set);
}

```

```

<<instance set public functions>>=
pub fn traverseGeneralization(
    inst_set: InstanceSet,
    comptime relationship: anytype,
    comptime phrase: generalization.TraversalPhrase,
    comptime subclass_id: @TypeOf(relationship).SubclassId,
) @TypeOf(relationship).InferDestRelClass(phrase, subclass_id).InstanceSet {
    return relationship.traverse(phrase, subclass_id, inst_set);
}

```

Tests

```

<<generalization module tests>>=
test "generalization traversal" {
    <<generalization test setup>>

    const vehicle: Vehicle = .{};
    const car: Car = .{};
    const truck: Truck = .{};
    const r1: R1 = .{};

    const v1 = vehicle.create(.{
        .make = "Toyota",
        .model = "Prius",
    });
    const c1 = car.create(.{
        .id = v1.readAttribute(.id),
        .door_count = 4,
    });
    r1.relate(.car, c1, v1);
    try testing.expectEqual(.car, r1.classify(v1));

    var cl_related = c1.traverse(r1, .from_subclass, .car);
    try testing.expectEqual(@as(usize, 1), cl_related.cardinality());
    try testing.expectEqualStrings("Toyota", cl_related.readAttribute(.make));
    try testing.expectEqualStrings("Prius", cl_related.readAttribute(.model));

    r1.unrelate(.car, c1, v1);
    cl_related = c1.traverseGeneralization(r1, .from_subclass, .car);
    try testing.expectEqual(@as(usize, 0), cl_related.cardinality());

    const v2 = vehicle.create(.{
        .make = "Peterbuilt",
        .model = "P-1000",
    });
}

```

```

const t2 = truck.create(.{
    .id = v2.readAttribute(.id),
    .gross_weight = 40_000,
});
r1.relate(.truck, t2, v2);

const t2_related = v2.traverseGeneralization(r1, .to_subclass, .truck);
try testing.expectEqual(@as(usize, 1), t2_related.cardinality());
try testing.expectEqual(@as(usize, 40_000), t2_related.readAttribute(.gross_weight));
}

```

The next set of tests use the multiple generalization case presented [previously](#).

```

<<generalization module tests>>=
test "multiple generalization tests" {
    const TailNumber = []const u8;
    const TakeoffType = enum { tilt_wing, tilt_rotor, thrust };

    const RotaryWingAircraft = RelClass(
        .rotary_wing_aircraft,
        struct { id: TailNumber, make: []const u8, model: []const u8 },
        &.{&.{.id}},
        &.{},
        .{},
        7,
        &.{},
        );
    const rotary_wing_aircraft = RotaryWingAircraft.init(0);

    const Helicopter = RelClass(
        .helicopter,
        struct { id: TailNumber, rotor_count: u8 },
        &.{&.{.id}},
        &.{},
        .{},
        4,
        &.{},
        );
    const helicopter = Helicopter.init(1);

    const VtolAircraft = RelClass(
        .vtol_aircraft,
        struct { id: TailNumber, type: TakeoffType },
        &.{&.{.id}},
        &.{},
        .{},
        3,
        &.{},
        );
    const vtol_aircraft = VtolAircraft.init(2);
}

```

```

const FixedWingAircraft = RelClass(
    .fixed_wing_aircraft,
    struct { id: TailNumber, make: []const u8, model: []const u8 },
    &.{&.{.id}},
    &.{},
    .{},
    7,
    &.{},
);
const fixed_wing_aircraft = FixedWingAircraft.init(3);

const Airplane = RelClass(
    .airplane,
    struct { id: TailNumber, wing_count: u8 },
    &.{&.{.id}},
    &.{},
    .{},
    4,
    &.{},
);
const airplane = Airplane.init(3);

const R1 = Generalization(
    .r1,
    RotaryWingAircraft,
    &.{ Helicopter, VtolAircraft },
    &.{},
);
const r1: R1 = .{};

const R2 = Generalization(
    .r2,
    FixedWingAircraft,
    &.{ Airplane, VtolAircraft },
    &.{},
);
const r2: R2 = .{};

// Create a population and relate the instances across the generalizations.

const rw_1 = rotary_wing_aircraft.create(.{
    .id = "N001",
    .make = "Bell",
    .model = "429",
});
const heli_1 = helicopter.create(. { .id = "N001", .rotor_count = 2 });
r1.relate(.helicopter, heli_1, rw_1);
try testing.expect(r1.isConsistent());

const fw_1 = fixed_wing_aircraft.create(.{
    .id = "N501",
}

```

```

    .make = "Cessna",
    .model = "525",
});
const airp_1 = airplane.create({ .id = "N501", .wing_count = 1 });
r2.relate(.airplane, airp_1, fw_1);
try testing.expect(r2.isConsistent());

const vtol_1 = vtol_aircraft.create({ .id = "N432", .type = .tilt_wing });
const rw_2 = rotary_wing_aircraft.create({
    .id = "N432",
    .make = "Airbus",
    .model = "Vahana",
});
const fw_2 = fixed_wing_aircraft.create({
    .id = "N432",
    .make = "Airbus",
    .model = "Vahana",
});
r1.relate(.vtol_aircraft, vtol_1, rw_2);
r2.relate(.vtol_aircraft, vtol_1, fw_2);
try testing.expect(r1.isConsistent());
try testing.expect(r2.isConsistent());

// Traverse R1 from Rotary Winged Aircraft 1 to Helicopter
// to find all the helicopters (one in this case).
const helicopters = rw_1.traverseGeneralization(r1, .to_subclass, .helicopter);
try testing.expectEqual(@as(usize, 1), helicopters.cardinality());
try testing.expectEqual(@as(u8, 2), helicopters.readAttribute(.rotor_count));

// Traverse R2 from Fixed Winged Aircraft 1 to Airplane
// to find all the airplanes (one in this case).
const airplanes = fw_1.traverseGeneralization(r2, .to_subclass, .airplane);
try testing.expectEqual(@as(usize, 1), airplanes.cardinality());
try testing.expectEqual(@as(u8, 1), airplanes.readAttribute(.wing_count));

// Traverse R1 to find all the vtol aircraft (one in this case).
const vtol_by_r1 = rw_2.traverseGeneralization(r1, .to_subclass, .vtol_aircraft);
try testing.expectEqual(@as(usize, 1), vtol_by_r1.cardinality());
try testing.expectEqual(.tilt_wing, vtol_by_r1.readAttribute(.type));

// Traverse R2 to find all the vtol aircraft (one in this case).
const vtol_by_r2 = fw_2.traverseGeneralization(r2, .to_subclass, .vtol_aircraft);
try testing.expectEqual(@as(usize, 1), vtol_by_r2.cardinality());
try testing.expectEqual(.tilt_wing, vtol_by_r2.readAttribute(.type));

// The vtol instance found by traversing R1 is the same as found traversing R2.
try testing.expect(vtol_by_r1.eql(vtol_by_r2));

// Traverse R1 to find all the vtol aircraft and then traverse R2 to
// find the fixed wing aircraft counterpart. Generalization compose in the manner.
const fw_by_r2 = rw_2
    .traverse(r1, .to_subclass, .vtol_aircraft)

```

```

        .traverse(r2, .from_subclass, .vtol_aircraft);
    try testing.expect(fw_by_r2.eql(fw_2.instanceSetFromInstanceRef()));

    // Unrelating the vtol instance, makes the generalization inconsistent.
    r1.unrelateSubclassInstances(.vtol_aircraft, vtol_1);
    try testing.expect(!r1.isConsistent());
    r2.unrelateSubclassInstances(.vtol_aircraft, vtol_1);
    try testing.expect(!r2.isConsistent());
}

}

```

Code layout

```

<<generalization.zig>>=
//! The `generalization.zig` module implements the concept of a
//! *Relational Class Generalization*.
//! A Generalization is a relationship between a Relational Class and at least
//! two other Relational Classes that are equivalence classes.
//! A Generalization has specific multiplicity and conditionality constraints
//! which are maintained on the instances of the participating Relational Classes.
//! A Generalization may be _navigated_ between its participating Relational Classes.
//! This is logically equivalent to applying the partitioning relation that the
//! Generalization represents.
<<edit warning>>
<<copyright info>>

const std = @import("std");
const builtin = @import("builtin");
const testing = if (builtin.os.tag == .freestanding)
    @import("resee_testing")
else
    std.testing;
const enums = std.enums;
const meta = std.meta;

const assert = std.debug.assert;
const panic = std.debug.panic;

pub const version = "1.0.0-a1";

<<generalization module declarations>>
<<generalization module public functions>>
<<generalization module private functions>>
<<generalization module tests>>

```

Summary

This chapter has shown the design and implementation of the data management facilities in ReSEE.

The model execution concepts state that the data facet of a domain is based on the relational theory of data. This implementation contains a direct, in-memory implementation of relational algebra that is used as the base for defining all the application data management facilities. Again, we reiterate that we are *not* building a RDMS, but rather a relational data mechanism to accommodate encoding application logic into data and reasoning on the data in the schema.

There is no explicit requirement that the implementation be accomplished directly by a relational algebra library. However, using relational algebra as the base lowers the cognitive burden for understanding how the execution model data handling operates. The execution model data facilities are implemented as a type system based on the **RelClass**, **Association**, and **Generalization** types. These types operate on **InstanceSet** objects and are based directly on an unadorned relation algebra type scheme.

As we stated at the beginning of this part of the book, it is our intent to factor all the data management functionality into ReSEE. This implies that there are no *ad hoc* data structures or half-baked algebras that are used to hold and operate on application data. All application data is held in **RelClass** type variables which are related to each other via **Associations** and / or **Generalizations**. Application data groupings are *specified* in terms of the number and data types of their attributes. All resulting relational classes are subject to the same algebraic operations. In this manner, the operations that can be performed on the application data are independent of the structure of that data and the results are based only on the data values contained in the relational classes.

This chapter has also shown our commitment to constrained execution. All application data is held as sets and the required identity constraints are enforced. The relationships between **RelClass** classes are also constrained. **Associations** and **Generalizations** both provide a means to detect if processing has failed to maintain the invariant conditions of the relationships.

In the next chapter, we show the counterpart of data management, namely execution sequencing.

[1] We will expand the notion of generated identifier values to make them unique within various contexts.

[2] There are ten associations between two different classes that have distinct values for the multiplicity and conditionality.

[3] Because this association is one-to-one, discarding the **LED pin** and moving the **Button pin** attribute to **Blinky LED** is equally valid.

[4] NULL, of course, does complicate matters and we are not interested in such complications.

[5] Also called *homogeneous* binary relations.

Sequencing Execution

In the last chapter, the design for managing application data was shown. The design is based on the relational model of data, extending the relational model to handle of the practicalities of building application software for our target platform. The data management design has three important constructs:

Relational Class

An extended relation variable that represents the abstraction of some entity in a subject matter domain.

Association

An abstraction of binary relations between Relational Classes.

Generalization

An abstraction of set partitioning between Relational Classes.

With these three ideas, we can construct a schema that specifies the logical characteristics of the subject matter domain. The schema is constrained by both the identify of individual class instances and by referential consistency between class instances for those classes participating in an association or generalization relationship.

It was important to design the data management facilities of ReSEE first. The class schema and its population of class instances provide the complete statement of self-consistent truth about the subject matter domain. We can ask questions about the data values and compare the results provided by the domain data population with the reality of the subject matter that the domain is intended to capture. The logic of the subject matter, as encoded in its data, can be exercised in our heads and the answer to any well-formed query of the data must be true. This is all very reassuring, as it goes a long way to building a firm foundation for reasoning about the subject matter of the domain. But relational classes and their relationships are only declarative of the subject matter logic and do not specifically direct any computations. For a reactive system, we must monitor the system environment and, as a reaction to changes in that environment, produce some side effect into the environment. Producing those side effects into the environment requires some form of computation.

It is now time to consider how execution is sequenced and how synchronization between different parts of the execution is accomplished. Each of the above three data management constructs has an associated means to specify the computational aspects of the construct. In the next sections, we examine the manner in which computation is sequenced and synchronized for each of the data management construct.

Relational Class Behavior

In a domain model, Relational Classes serve two primary purposes. Many classes provide descriptive data that is used by other classes. They have no other behavior and do not actively participate in producing the external behavior of the domain. One usage is to consider the specification classes as the mini-database which the application carries as part of its own logical specification of the environment with which it deals. These classes can be thought of as *passive*. An example of this type of class, which we have already seen, is the **Point Scaling** class. This class holds important descriptive information about the slope and intercept of a linear scaling expression. Whatever mechanism is used to compute the scaling of an **Input Point** is parameterized by the attribute values of instances of **Point Scaling** and the association between the two classes.

Other classes exhibit behavior that changes over time and can influence the behavior of other classes. We model that behavior by attaching a state model to *active classes* to model the *life cycle*.

of the class. We have seen the examples of the **Bouncy Button** and **Blinky LED** previously. That behavior was specified using a state model to capture the time-varying behavior and interactions of the two classes.

Note especially that state models are only used to specify the life cycle behavior of the relational class and *not* a larger set of classes or for some functional decomposition. The overall active behavior of the domain is the composition of the life cycles of the active classes and is achieved both by the computation carried out by individual state actions as well as the interactions between the state models.

One can consider that all classes have a behavior. For passive classes, the behavior is trivial: the class comes into existence in a single state that has no action and it responds to no events. It remains in existence until either it is deleted or the program terminates. Classes with trivial state models are not considered further.

For *active* classes, time-varying behavior is described and captured in a state model.

Active Class Creation Operations

Creating an active class involves additional work to manage the current state of the instance state machine. Also, active state may be created asynchronously. The functions in this section handle these cases.

Creation functions are also provided that return an error on failure rather than creating a panic condition. Note, that the vast majority of cases *do not* benefit from returning an error since there is often no recovery that can be made. State actions assume they run on a “perfect” platform. The error return creation functions are useful for those cases where the instance creation is the result of interactions external to the domain and the only viable recovery is to discard requests that exceed the configured memory to prevent a panic condition.

```
<<state model declarations>>
const RelVarError = @import("relvar.zig").RelVarError;
```

Active Class Initialization Operations

When a Relational Class also has an associated state model, the storage for the state machines current state and for any continuation events that a state machine might generate have to be initialized properly. These function are provided to support instance creation for relational classes that have an state model.

```
<<state model public functions>>
/// The `initInState` function initializes the current state of the
/// instance given by, `inst_ref`, to be the state given by, `state_id`.
/// This function also initializes the storage for the state machine's
/// continuation event.
pub fn initInState(
    _: Self,
    inst_ref: InstanceRef,
    state_id: StateId,
) void {
    current_states[inst_ref.tuple_ref.tuple_id] = state_id;
    continuation_events[inst_ref.tuple_ref.tuple_id] = null;
```

```
}
```

```
<<state model public functions>>=
/// The `initInDefaultState` function initializes the current state of the
/// instance given by `inst_ref` to the default initial state.
pub fn initInDefaultState(
    self: Self,
    inst_ref: InstanceRef,
) void {
    self.initInState(inst_ref, initial_state);
}
```

```
<<state model public functions>>=
/// The `initInPseudoInitialState` function initializes the current state of the
/// instance given by `inst_ref` to the pseudo-initial state.
pub fn initInPseudoInitialState(
    self: Self,
    inst_ref: InstanceRef,
) void {
    assert(has_pseudo_initial);
    self.initInState(inst_ref, .@"@");
}
```

Active Class Interface

The presence of a state model for a Relational Classes means that some functions require additional code to implement so that the state model may be taken into consideration. In an attempt to mask such differences between the use of active and passive classes, some function are given a polymorphic interface as shown below.

```
<<class declarations>>=
pub const has_state_model = class_options.opt_state_model_spec != null;

pub const ClassModel = if (has_state_model)
    StateModel(Self, class_options.opt_state_model_spec.?)
else
    void;

pub const classmodel = if (has_state_model)
    ClassModel{}
else
    void;
```

```
<<class public functions>>=
pub fn receiveEvent(
    _: Self,
    event: Event,
) bool {
    return if (has_state_model)
```

```

        classmodel.receiveEvent(event)
    else
        @compileError(@tagName(class_id) ++ ": class has no state model");
}

pub const Event = if (has_state_model)
    ClassModel.Event
else
    void;

pub const StateId = if (has_state_model)
    ClassModel.StateId
else
    void;

pub const TargetStateId = if (has_state_model)
    ClassModel.TargetStateId
else
    void;

pub const EventId = if (has_state_model)
    ClassModel.EventId
else
    void;

```

Tests

Light Bulb

This test uses a simple three state model to describe the life cycle of a light bulb. The light bulbs start in the **off** state and can be turned **on** and **off** by events. The **broken** state is a final state for light bulb. The state diagram is shown as follows:

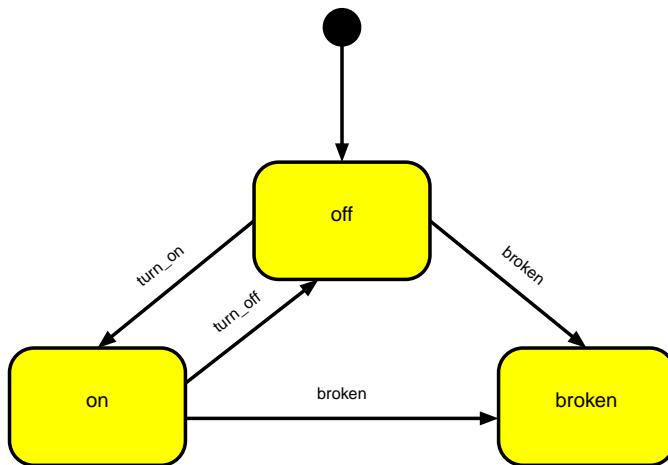


Figure 52. State Diagram for a Light Bulb

The transition matrix shows the complete set of transitions for the light bulb. Notice that all the transitions that were not represented in the diagram are ignored.

	turn_off	turn_on	broken
off	ignore	on	broken
on	off	ignore	broken
broken	ignore	ignore	ignore

Table 13. Light Bulb Transition Matrix

With the transition matrix determined, the specification of the state model is a direct translation of the information in the diagram and transition matrix.

```
<<active class module tests>>=
const light_state_model_spec: StateModelSpec = .{
    .state_names = &.{ "off", "on", "broken" },
    .event_names = &.{ "turn_on", "turn_off", "broken" },
    .transitions = &.{ {
        "off", &.{ {
            "turn_on", "on" },
            {"broken", "broken" } },
        },
    },
    .{
        "on", &.{ {
            "turn_off", "off" },
            {"broken", "broken" } },
        },
    },
    .default_initial_state = "off",
    .StateActivities = light_activities,
    .unspecified_transition = "ignore",
    .terminal_states = &.{},
};


```

The specification is simplified by setting the `unspecified_transition` to “ignore”.

The behavior we use for this test is to increment counters when the “off” and “on” states are entered. The heading of the Relational Class contains a couple of counters.

```
<<active class module tests>>=
const LightHeading = struct {
    id: ClassGeneratedId = generated_id_default,
    on_count: usize = 0,
    on_maximum: usize = 5,
    off_count: usize = 0,
};


```

Note that identifying light bulbs in this simple example is arbitrary, so we simply use a class generated identifier.

The actions for the “off” and “on” states increment their associated counter. For the `off` state, the counter is incremented by one. For the `on` state, the counter is incremented by the value of an event parameter. In both cases, we log the value of the instance after the update operation. Additionally, the “on” state tests if the maximum number of times the light has gone on is exceeded, and if so, signals `.broken` as a continuation event.

```
<<active class module tests>>=
const light_activities = struct {
    const InstanceRef = Light.InstanceRef;
    const log = std.log;

    pub fn off(
        inst: InstanceRef,
    ) void {
        inst.updateAttribute(.off_count, inst.readAttribute(.off_count) + 1);
        log.debug("in 'off' state:\n{any}", .{
            inst.dereference(),
        });
    }

    pub fn on(
        inst: InstanceRef,
        incr: usize,
    ) void {
        const new_on_count = inst.readAttribute(.on_count) + incr;
        inst.updateAttribute(.on_count, new_on_count);
        if (new_on_count >= inst.readAttribute(.on_maximum)) {
            inst.continueWith(.broken, .{});
        }
        log.debug("in 'on' state:\n{any}", .{
            inst.dereference(),
        });
    }

    pub fn broken(
        inst: InstanceRef,
    ) void {
        log.debug("in 'broken' state:\n{any}", .{
            inst.dereference(),
        });
    }
};
```

With the class heading, state model specification, and action functions determined, we can now create the active light class.

```
<<active class module tests>>=
const Light = RelClass(
    .light, // class_tag
    LightHeading, // Heading
    &.{ // identifiers
        &.{ .id }, // id 0
```

```

    },
    &{}, // dep_attr_props
    .{
        .opt_state_model_spec = &light_state_model_spec, // StateModelSpec
    }, // class options
    10, // max_capacity
    &{}, // initial_population
);

```

```

<<active class module tests>>=
test "light class model" {
    const LightStateModel = Light.ClassModel;
    const light_state_model: LightStateModel = .{};
    try testing.expect(!LightStateModel.has_pseudo_initial);
    try testing.expectEqual(LightStateModel.StateId.off, LightStateModel.initial_state);
    try testing.expectEqual(LightStateModel.TargetStateId.ignore, LightStateModel
.default_transition);
    try testing.expectEqual(
        LightStateModel.TargetStateId.on,
        light_state_model.transition(
            LightStateModel.StateId.off,
            LightStateModel.EventId.turn_on,
        ),
    );
    try testing.expectEqual(
        LightStateModel.TargetStateId.ignore,
        light_state_model.transition(
            LightStateModel.StateId.broken,
            LightStateModel.EventId.turn_on,
        ),
    );
}

```

```

<<active class module tests>>=
test "create active light instances" {
    const light = Light.init(0);

    const light_1 = light.create(.{});
    defer light_1.destroy();

    try testing.expectEqual(@as(usize, 0), light_1.readAttribute(.on_count));
    try testing.expectEqual(@as(usize, 0), light_1.readAttribute(.off_count));
}

```

```

<<active class module tests>>=
test "light instances receive events" {
    testing.log_level = .info;

    const light: Light = .{};

    try testing.expectEqual(@as(usize, 0), light.readAttribute(.on_count));
    try testing.expectEqual(@as(usize, 0), light.readAttribute(.off_count));
}

```

```

const light_1 = light.create(.{});
defer light_1.destroy();

// off - turn_on -> on
_= light.receiveEvent(light_1.makeEvent(.turn_on, .{ 4 }));
try testing.expectEqual(@as(usize, 4), light_1.readAttribute(.on_count));
try testing.expectEqual(@as(usize, 0), light_1.readAttribute(.off_count));

// on - turn_on -> ignore
_= light.receiveEvent(light_1.makeEvent(.turn_on, .{ 2 }));
try testing.expectEqual(@as(usize, 4), light_1.readAttribute(.on_count));
try testing.expectEqual(@as(usize, 0), light_1.readAttribute(.off_count));

// on - turn_off -> off
_= light.receiveEvent(light_1.makeEvent(.turn_off, .{}));
try testing.expectEqual(@as(usize, 4), light_1.readAttribute(.on_count));
try testing.expectEqual(@as(usize, 1), light_1.readAttribute(.off_count));

// off - turn_on -> on -> broken
_= light.receiveEvent(light_1.makeEvent(.turn_on, .{ 1 }));
try testing.expectEqual(@as(usize, 5), light_1.readAttribute(.on_count));
try testing.expectEqual(@as(usize, 1), light_1.readAttribute(.off_count));

// broken - turn_on -> ignore
_= light.receiveEvent(light_1.makeEvent(.turn_on, .{ 1 }));
try testing.expectEqual(@as(usize, 5), light_1.readAttribute(.on_count));
try testing.expectEqual(@as(usize, 1), light_1.readAttribute(.off_count));
}

```

Asynchronous Connection

This test demonstrates asynchronous creation using a creation event. The following is the state diagram for the test class.

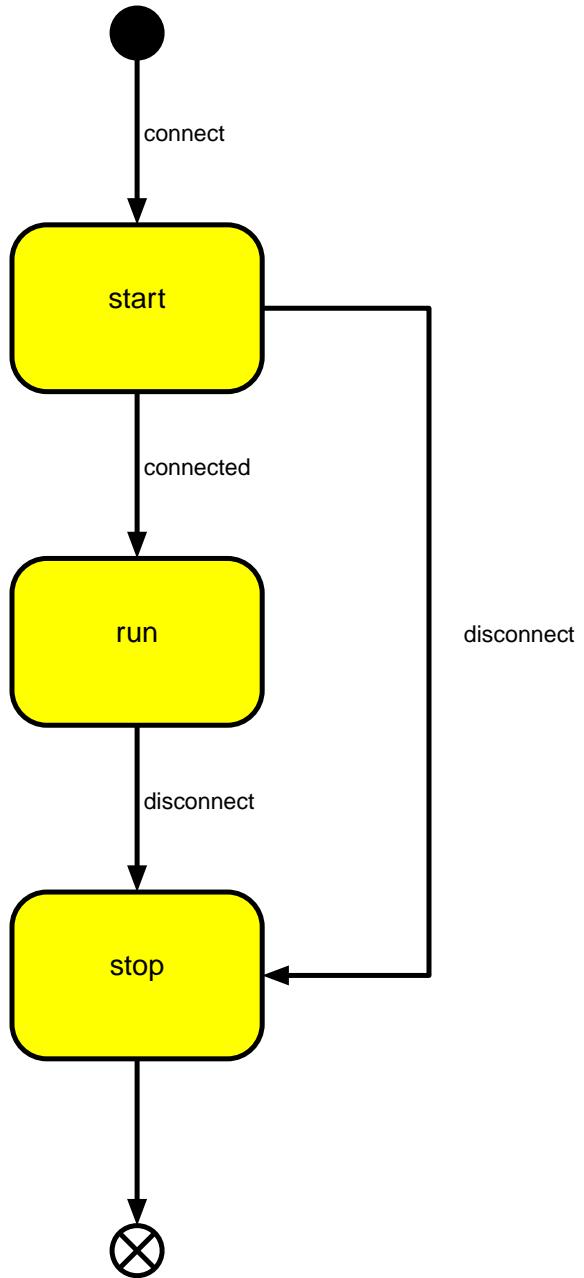


Figure 53. State Diagram for a Connection

In this diagram, the arrow labeled, **connect**, shows the transition from the pseudo-initial state. The arrow from **stop** to the crossed-circle shows that **stop** is a terminal state and the instance is deleted after the action function for **stop** is executed.

As before, the transition matrix shows the complete specification for the state model.

	connect	connected	disconnect
@	start	never	never
start	never	run	stop
run	never	never	stop

Table 14. Connection Transition Matrix

The transition matrix gives us the information to specify the state model.

```
<<active class module tests>>=
const conn_state_model_spec: StateModelSpec = .{
    .state_names = &.{ "start", "run", "stop" },
    .event_names = &.{ "connect", "connected", "disconnect" },
    .transitions = &.{

        .{
            "@", &.{

                .{ "connect", "start" },
            },
        },
        .{
            "start", &.{

                .{ "connected", "run" },
                .{ "disconnect", "stop" },
            },
        },
        .{
            "run", &.{

                .{ "disconnect", "stop" },
            },
        },
    },
    .default_initial_state = "start",
    .StateActivities = connection_activities,
    .unspecified_transition = "never",
    .terminal_states = &.{ "stop" },
};
```

For this simple test, there are no descriptive attributes and we use a class generated identifier.

```
<<active class module tests>>=
const ConnectionHeading = struct {
    id: ClassGeneratedId = generated_id_default,
};
```

The state actions only produce some debug output.

```
<<active class module tests>>=
const connection_activities = struct {
    const InstanceRef = Connection.InstanceRef;
```

```

const log = std.log;

pub fn start(
    inst: InstanceRef,
) void {
    log.debug("in 'start' state:\n{any}", .{
        inst.dereference(),
    });
}

pub fn run(
    inst: InstanceRef,
) void {
    log.debug("in 'run' state:\n{any}", .{
        inst.dereference(),
    });
}

pub fn stop(
    inst: InstanceRef,
) void {
    log.debug("in 'stop' state:\n{any}", .{
        inst.dereference(),
    });
}
);

```

```

<<active class module tests>>=
const Connection = RelClass(
    .connection, // class tag
    ConnectionHeading, // Heading
    &.{ // identifiers
        &{.id}, // id 0
    },
    &{}, // dependent attribute properties
    .{
        .opt_state_model_spec = &conn_state_model_spec, // StateModelSpec
    },
    10, // max capacity
    &{}, // initial population
);

```

```

<<active class module tests>>=
test "connection class model" {
    const ConnStateModel = Connection.ClassModel;
    const conn_state_model: ConnStateModel = .{};

    try testing.expect(ConnStateModel.has_pseudo_initial);
    try testing.expectEqual(ConnStateModel.StateId.start, ConnStateModel.initial_state);
    try testing.expectEqual(ConnStateModel.TargetStateId.never, ConnStateModel.default_transition);

```

```

try testing.expectEqual(
    ConnStateModel.TargetStateId.run,
    conn_state_model.transition(
        ConnStateModel.StateId.start,
        ConnStateModel.EventId.connected,
    ),
);
try testing.expectEqual(
    ConnStateModel.TargetStateId.never,
    conn_state_model.transition(
        ConnStateModel.StateId.run,
        ConnStateModel.EventId.connect,
    ),
);
}

```

```

<<active class module tests>>=
test "active connection class" {
    testing.log_level = .info;
    const connection = Connection.init(0);

    const conn_1 = connection.createAsync(.{});
    defer conn_1.destroy();

    var terminal_trans = connection.receiveEvent(conn_1.makeEvent(.connect, .{}));
    try testing.expectEqual(false, terminal_trans);

    terminal_trans = connection.receiveEvent(conn_1.makeEvent(.connected, .{}));
    try testing.expectEqual(false, terminal_trans);

    terminal_trans = connection.receiveEvent(conn_1.makeEvent(.disconnect, .{}));
    try testing.expectEqual(true, terminal_trans);

    const conn_2 = connection.createAsync(.{});
    defer conn_2.destroy();

    terminal_trans = connection.receiveEvent(conn_2.makeEvent(.connect, .{}));
    try testing.expectEqual(false, terminal_trans);

    terminal_trans = connection.receiveEvent(conn_2.makeEvent(.disconnect, .{}));
    try testing.expectEqual(true, terminal_trans);
}

```

Association Behavior

It is common that an association is used to represent the allocation of a resource and, usually, there are not sufficient resources to satisfy all requests simultaneously. Such situations give rise to the notion of a competitive association where resource requests must be carefully matched with resource usage. Resources are used temporarily and then released for subsequent re-use.

A state model is used to serialize relating two class instances across a competitive association. The

state model provides the means to create an association between class instances in a single run-to-completion action function, thus insuring that any potential concurrency effects are properly serialized. Such a state model, attached directly to an association, is called an *assigner*.

When a state model is used to relate class instances, it is a corollary that the two participating classes also have state models. An instance of a participating class which requires the resource signals its request to the association assigner and waits until it receives an event indicating that a resource has been assigned to it. Similarly, the class which represents the resource can signal when the resource is available and then wait until it is assigned to fulfill a request. The association assigner mediates the interactions of the request and resource instances and arranges for an instance of the competitive association to be created in an action function that runs to completion. It is the interactions of the three state machines, one attached to each participating instance and one attached to the association itself, that establish a protocol for resolving the resource competition.

In the simplest case, all the requests for a resource and all the available resources are considered fungible. The primary action of the assigner state model is to select any pending request and pair it with any available resource. Other policy decisions may be dictated by the requirements. For example, when selecting a resource, the assigner may enforce a first-come-first-served policy for requests by managing a serial association with the requesting class or, perhaps, time stamping the request's arrival. For the simple case of fungible requests and resources, which is the most common, the association assigner state model has only a single state machine. The assigner is considered to be a *single assigner*. Because the single assigner is the simplest way to manage competition, it is specialized to that role.

A more complicated case is also present in some domains. In this case, the resources are not considered fungible. For example, a given resource may have a physical location that prevents it from being used to satisfy all potential requests. Requests must be directed to the correct set of resources that satisfy the physical location constraints. The additional constraints are realized in the model by a third class participating in the assignment protocol. The third class is used to *partition* the participating classes of the association. The additional requirements which make the resource non-fungible are carried in the semantics of the additional associations between the partitioning class and the requesting class and between the partitioning class and the resource class. This type of assigner is called a *multiple assigner* and there is one assigner state machine (specified by a state model attached to the association) for each instance of the partitioning class.

Single Assigner Example

The archetypical example of this requirement considers what happens to Customers and Clerks in a conventional (and somewhat old-fashioned) storefrontnote:[The details of this example are from the original work that described these concepts. In an attempt to avoid an overly abstract discussion, the Customer and Clerk classes are intended as concrete proxies whose roles are readily understood.]. A Customer requires service from a Clerk and, conversely, a Clerk provides service to a Customer. We can represent the situation with a class diagram.

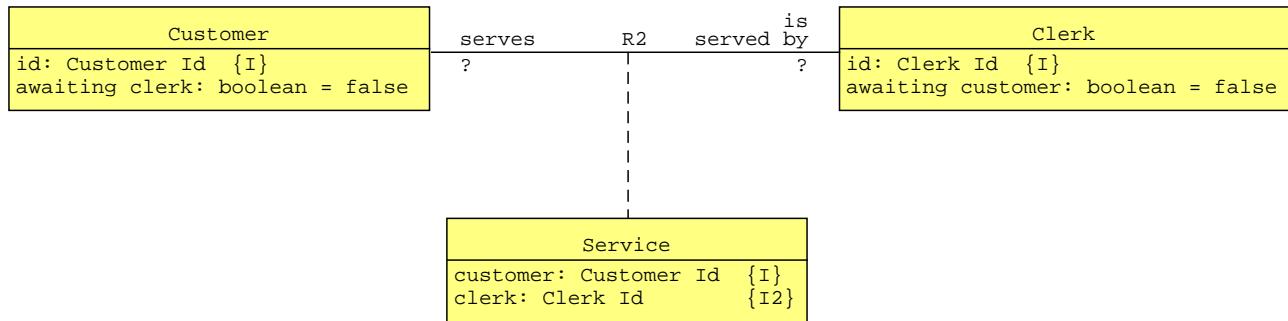


Figure 54. Customer / Clerk Association

With Customers coming and going and Clerks finishing service for Customers, we need to be careful about how Customers and Clerks are assigned to each other across R2. The requirement is to create instances of R2, and consequently instances of Service, in a single action. This is to ensure that the at-most-one to at-most-one multiplicity and conditionality is maintained. Customers and Clerks must be paired together and having Customer instances associate on their own to some Clerk instance that is trying to associate itself to some Customer would result in conflict under any concurrent operation.

The single assigner state model (for which only one state machine ever exists) is shown in the following figure. We do not show the interactions of the assigner with the state machines for **Customer** and **Clerk**. At this point, we have not described the manner in which events are signaled between state machines. We will return to all of this later, but, for now, we focus on the state model for the assigner and point out where the interactions with the **Customer** and **Clerk** instances takes place.

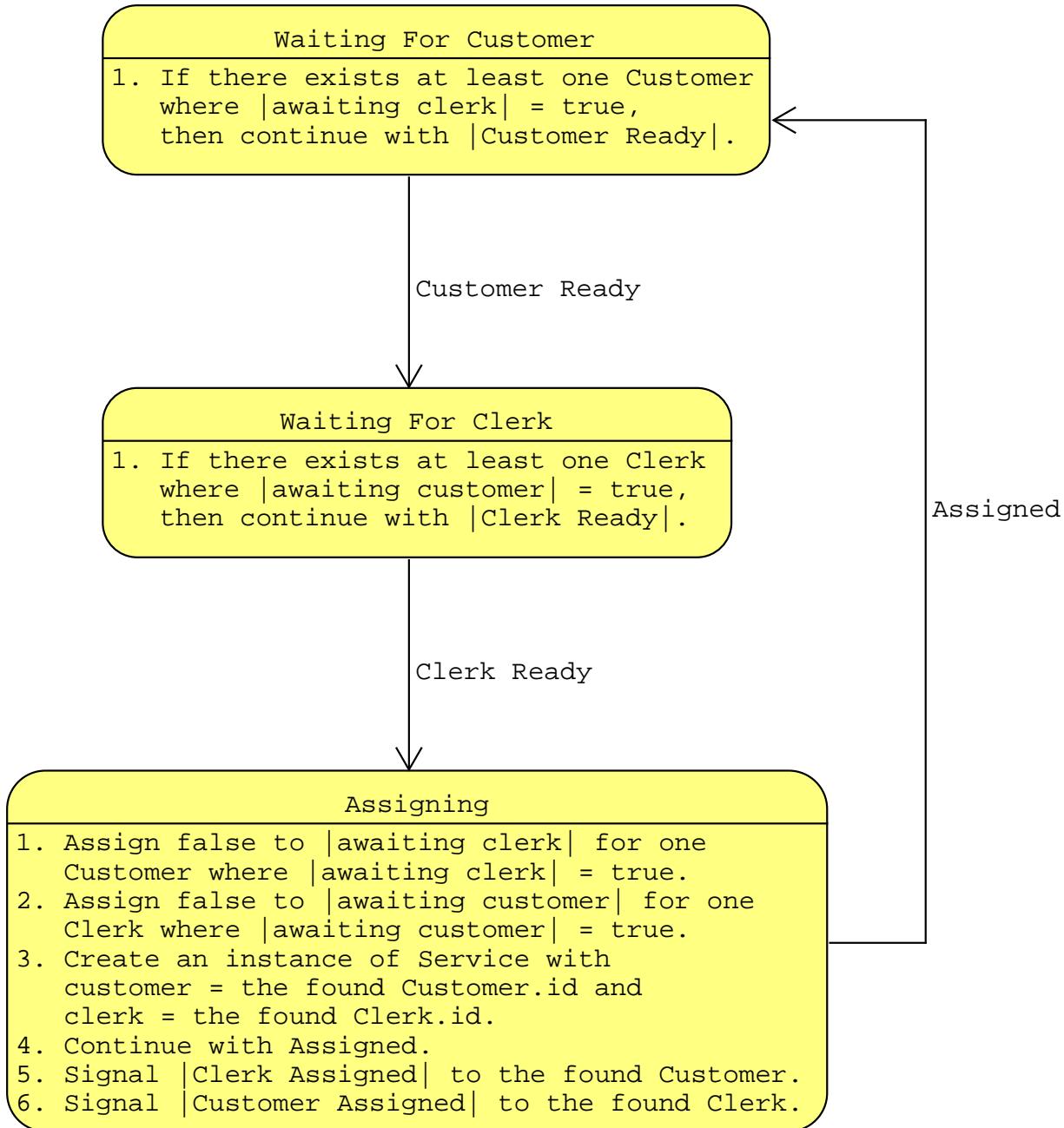


Figure 55. Customer / Clerk Assigner State Model

The assignment protocol operates by having each participating class hold a boolean attribute indicating its status. In this example, when a **Customer** requires the help of a **Clerk**, a state action of the **Customer** state model sets the **awaiting clerk** attribute to **true** and signals the **Customer Ready** event to the **R2** assigner state machine. When a **Clerk** is available to serve a **Customer**, a state action of the **Clerk** state model sets the **awaiting customer** attribute to **true** and signals the **Clerk Ready** to the **R2** assigner state machine. The transitions of the **R2** assigner state model finally arrives at a state where it finds both a **Customer** and a **Clerk** whose statuses indicates that the association could be formed between them. The **Assigning** action function in the **R2** assigner state model then forms the association and negates the statuses in a single action that runs to completion.

Single Assigner Implementation

The implementation strategy for a single assigner is to use the state model code that we have already shown and pair it with a Relational Class that has only a single instance. This state model and singleton class are attached to an association by means of the interface shown [below](#).

```
<<assigner module public functions>>=
pub fn Assigner(
    comptime assigner_tag: @Type(.enum_literal),
    comptime state_model_spec: *const StateModelSpec,
) type {
    return struct {
        const Self = @This();

        pub const AssignerClass = RelClass(
            assigner_tag, // class_tag
            struct { id: void = {} }, // Heading ①
            &.{ // identifiers
                &.{ .id }, // id 0
            },
            &.{},
            .{
                .opt_state_model_spec = state_model_spec,
                .load_factor = 100,
            },
            1, // max_capacity
            &.{ // initial_population
                .{ .singleton, {} },
            },
        );
    };

    pub const assigner_class: AssignerClass = .{};

    pub fn select(
        _: Self,
    ) AssignerClass.InstanceRef {
        return assigner_class.selectInitialInstance(.singleton);
    }

    pub fn receiveEvent(
        _: Self,
        event: Event,
    ) bool {
        return assigner_class.receiveEvent(event);
    }

    pub const InstanceRef = AssignerClass.InstanceRef;
    pub const InstanceSet = AssignerClass.InstanceSet;
    pub const Event = AssignerClass.Event;
    pub const EventId = AssignerClass.EventId;
};

}
```

- ① The only attribute is an identifier and its type is `void`. The `void` type insures that there can be only one instance since void types can only assume one value, i.e. `void{}`.

Tests

Single Assigner Tests

To test the single assigner code, the previous example is implemented. Note that in the implementation we use naming conventions that are more appropriate to Zig.

Start by defining the Relation Classes, `customer` and `clerk`. Note that there is no implementation presence for the Service class. Since the Service class contains no information other than the referential attributes to the customer and clerk participants, the adjacency matrix implementation of R2 suffices to record the necessary information in the ReSEE platform model.

```
<<association module tests>>=
const Customer = RelClass(
    .customer,
    struct {
        // Customers are identified arbitrarily
        id: ClassGeneratedId = generated_id_default,
        awaiting_clerk: bool = false,
    },
    &.{ &.{ .id } },
    &.{},
    .{},
    5,
    &.{ 
        // To have contention for the clerk, we start with multiple customers.
        .{ .customer1, .{} },
        .{ .customer2, .{} },
        .{ .customer3, .{} },
    },
);
;

const Clerk = RelClass(
    .clerk,
    struct {
        // Clerks are identified arbitrarily
        id: ClassGeneratedId = generated_id_default,
        awaiting_customer: bool = false,
    },
    &.{ &.{ .id } },
    &.{},
    .{},
    5,
    &.{ 
        // In this store, there is only one lonely clerk.
        .{ .clerk1, .{} },
    },
);
;
```

It is vitally important to completely fill the transition matrix for a state model. Every transition must be considered and a completed transition matrix demonstrates that every transition is covered. The state diagram is essential when designing a state model and trying to step through the transitions in your head. A transition matrix is essential when implementing a state model and ensures state machine execution is properly constrained. For the Customer / Clerk assigner state model, the following table shows the transition matrix.

	customer_ready	clerk_ready	assigned
waitingForCustomer	waitForClerk	ignore	ignore
waitForClerk	ignore	assigning	ignore
assigning	never (1)	never (1)	waitForCustomer

Table 15. Customer / Clerk Assigner State Transition Matrix

(1) Since the `assigned` event is unconditionally signaled from the `assigning` state as a continuation event and since a continuation event is always received in preference to an interactive event, `assigning` is a transitory state and always transitions out of the `assigning` state at the end of its action function. It is logically impossible to receive the `customer_ready` or `clerk_ready` events when in the `assigning` state. If these never transitions are detected at run time, something is terribly wrong.

Given the transition matrix, the state model specification follows directly.

```
<<association module tests>>=
const r2_state_model_spec: StateModelSpec = .{
    .state_names = &.{ "waitForCustomer", "waitForClerk", "assigning" },
    .event_names = &.{ "customer_ready", "clerk_ready", "assigned" },
    .transitions = &.{ {
        .{
            "waitForCustomer", &.{ {
                .{ "customer_ready", "waitForClerk" },
            },
        },
        .{
            "waitForClerk", &.{ {
                .{ "clerk_ready", "assigning" },
            },
        },
        .{
            "assigning", &.{ {
                .{ "assigned", "waitForCustomer" },
                .{ "customer_ready", "never" },
                .{ "clerk_ready", "never" },
            },
        },
    },
    .default_initial_state = "waitForCustomer",
    .StateActivities = r2_assigner_activities,
    .unspecified_transition = "ignore",
    .terminal_states = &.{},
}
```

```
};
```

The action functions for the **R2** assigner are direct transliteration of the structured description from the graphic.

```
<<association module tests>>=
const r2_assigner_activities = struct {
    const log = std.log;
    const customer: Customer = .{};
    const clerk: Clerk = .{};
    const r2: R2 = .{};
    const InstanceRef = R2.InstanceRef;

    pub fn waitingForCustomer(
        self: InstanceRef,
    ) void {
        log.debug("in 'waitingForCustomer' state: customer =\n{any}\nclerk=\n{any}\n", .{
            customer.selectAll().relValueOf(),
            clerk.selectAll().relValueOf(),
        });

        if (customer.selectEqual(.awaiting_clerk, true).notEmpty()) {
            self.continueWith(.customer_ready, .{});
        }
    }

    pub fn waitingForClerk(
        self: InstanceRef,
    ) void {
        log.debug("in 'waitingForClerk' state: customer =\n{any}\nclerk=\n{any}\n", .{
            customer.selectAll().relValueOf(),
            clerk.selectAll().relValueOf(),
        });

        if (clerk.selectEqual(.awaiting_customer, true).notEmpty()) {
            self.continueWith(.clerk_ready, .{});
        }
    }

    pub fn assigning(
        self: InstanceRef,
    ) void {
        log.debug("in 'assigning' state: customer =\n{any}\nclerk=\n{any}\n", .{
            customer.selectAll().relValueOf(),
            clerk.selectAll().relValueOf(),
        });

        const ready_customer = customer.selectEqual(.awaiting_clerk, true).limit(1);
        const ready_clerk = clerk.selectEqual(.awaiting_customer, true).limit(1);
        if (ready_customer.notEmpty() and ready_clerk.notEmpty()) {
            ready_customer.updateAttribute(.awaiting_clerk, false);
            ready_clerk.updateAttribute(.awaiting_customer, false);
        }
    }
}
```

```

r2.relate(
    ready_customer.instanceRefFromInstanceSet(),
    .@"is served by",
    ready_clerk.instanceRefFromInstanceSet(),
    null,
);
;

// In practice, customer and clerk would both have state models
// and the assigner would signal them at this point to indicate
// the assignment. For testing, the extended interactions are omitted.
} else {
    unreachable; ①
}

self.continueWith(.assigned, .{});
}
};


```

- ① If this point in the execution is reached, there is a quandary over what should happen. The implication is that somehow the `awaiting_clerk` or `awaiting_customer` status values that drove the state transitions to this state have somehow become false as if a customer or clerk had changed their mind. There is no participant in `r2` to assign. This is deemed a violation of the assigner protocol and the code insists that it cannot happen. For situations where some cancellation facility is required, an alternative is to wait for the assignment and then immediately release the resource.

Finally, R2 is declared as an *at most one* to *at most one* association between **Customer** and **Clerk** with an assigner as given by the state model specification.

```

<<association module tests>>=
// Customer ?-"serves" -R2- "is served by"-? Clerk
const r2_spec = AssociationSpec{
    .DomainRelClass = Customer,
    .codomain_ref = .@"?",
    .codomain_phrase = .serves,
    .assoc_tag = .r2,
    .domain_phrase = .@"is served by",
    .domain_ref = .@"?",
    .CodomainRelClass = Clerk,
    .assigner_state_model = &r2_state_model_spec,
};

const R2 = Association(
    &r2_spec,
    &.{},
);

```

For testing purposes, we take over the event generation and reception for the Customer / Clerk state model. As stated previously, we have not yet described how state machine signal events to each other. In the test scenario, we use the class functions and event reception operation that have already been shown. The next chapter, describes how classes work together in a subject matter domain.

```

<<association module tests>>=
test Assigner {
    testing.log_level = .info;

    const customer = Customer.init(0);
    const clerk = Clerk.init(1);
    const r2: R2 = .{};

    // To demonstrate contention, all the customers arrive at once.
    const customer_1 = customer.selectInitialInstance(.customer1);
    customer_1.updateAttribute(.awaiting_clerk, true);
    const r2_assigner = r2.select();
    _ = r2.receiveEvent(r2_assigner.makeEvent(.customer_ready, .{}));

    const customer_2 = customer.selectInitialInstance(.customer2);
    customer_2.updateAttribute(.awaiting_clerk, true);
    _ = r2.receiveEvent(r2_assigner.makeEvent(.customer_ready, .{}));

    const customer_3 = customer.selectInitialInstance(.customer3);
    customer_3.updateAttribute(.awaiting_clerk, true);
    _ = r2.receiveEvent(r2_assigner.makeEvent(.customer_ready, .{}));

    // Clerk finishes with previous customer.
    const clerk_1 = clerk.selectInitialInstance(.clerk1);
    clerk_1.updateAttribute(.awaiting_customer, true);
    _ = r2.receiveEvent(r2_assigner.makeEvent(.clerk_ready, .{}));

    // When the `clerk_ready` event is received, there are ready customers
    // and the first assignment is made. The assignment leaves the other
    // customers still waiting.
    var waiting_customers = customer.selectEqual(.awaiting_clerk, true);
    try testing.expectEqual(@as(usize, 2), waiting_customers.cardinality());
    // The assignment marks the clerk as busy.
    try testing.expectEqual(false, clerk_1.readAttribute(.awaiting_customer));

    // Verify that the relationship was created.
    try testing.expect(r2.isConsistent());
    const assigned_customer = clerk_1.traverseAssociation(r2, null, .customer);
    try testing.expectEqual(@as(usize, 1), assigned_customer.cardinality());
    try testing.expectEqual(false, assigned_customer.readAttribute(.awaiting_clerk));

    // Clerk finishes with the first customer and deletes the instance of r2
    // that represents the service assignment.
    var prev_customer = r2.unrelateInstance(clerk_1, .serves);
    try testing.expectEqual(@as(usize, 1), prev_customer.cardinality());

    // The clerk is now ready to serve another customer.
    clerk_1.updateAttribute(.awaiting_customer, true);
    _ = r2.receiveEvent(r2_assigner.makeEvent(.clerk_ready, .{}));

    // Since there are waiting customers, another assignment is made.
    waiting_customers = customer.selectEqual(.awaiting_clerk, true);

```

```

try testing.expect(r2.isConsistent());
// Only one customer remains.
try testing.expectEqual(@as(usize, 1), waiting_customers.cardinality());
try testing.expectEqual(false, clerk_1.readAttribute(.awaiting_customer));

// Again the clerk finishes and posts that he is ready for another customer.
// Since there remains one customer waiting, another assignment is made.
prev_customer = r2.unrelateInstance(clerk_1, .serves);
try testing.expectEqual(@as(usize, 1), prev_customer.cardinality());
clerk_1.updateAttribute(.awaiting_customer, true);
_ = r2.receiveEvent(r2_assigner.makeEvent(.clerk_ready, .{}));

waiting_customers = customer.selectEqual(.awaiting_clerk, true);
try testing.expect(r2.isConsistent());
// No more customers are waiting.
try testing.expectEqual(@as(usize, 0), waiting_customers.cardinality());
try testing.expectEqual(false, clerk_1.readAttribute(.awaiting_customer));

// Finally, after the third customer is finished, the clerk posts
// as ready, but no assignments are made.
prev_customer = r2.unrelateInstance(clerk_1, .serves);
try testing.expectEqual(@as(usize, 1), prev_customer.cardinality());
clerk_1.updateAttribute(.awaiting_customer, true);
_ = r2.receiveEvent(r2_assigner.makeEvent(.clerk_ready, .{}));
try testing.expect(r2.isConsistent());

const current_customer = clerk_1.traverseAssociation(r2, null, .customer);
try testing.expectEqual(@as(usize, 0), current_customer.cardinality());
}

```

Multiple Assigner Example

Consider a large store which divided up into different departments that sell quite different goods. For example, let's assume there is a housewares department and a women's shoes department and perhaps many more. Clerks that work in the various departments have specialized knowledge about the products sold in that department. It is store policy that clerks must only service customers that are shopping in the same department as they work.

The following class diagram shows the arrangement to enforce this policy.

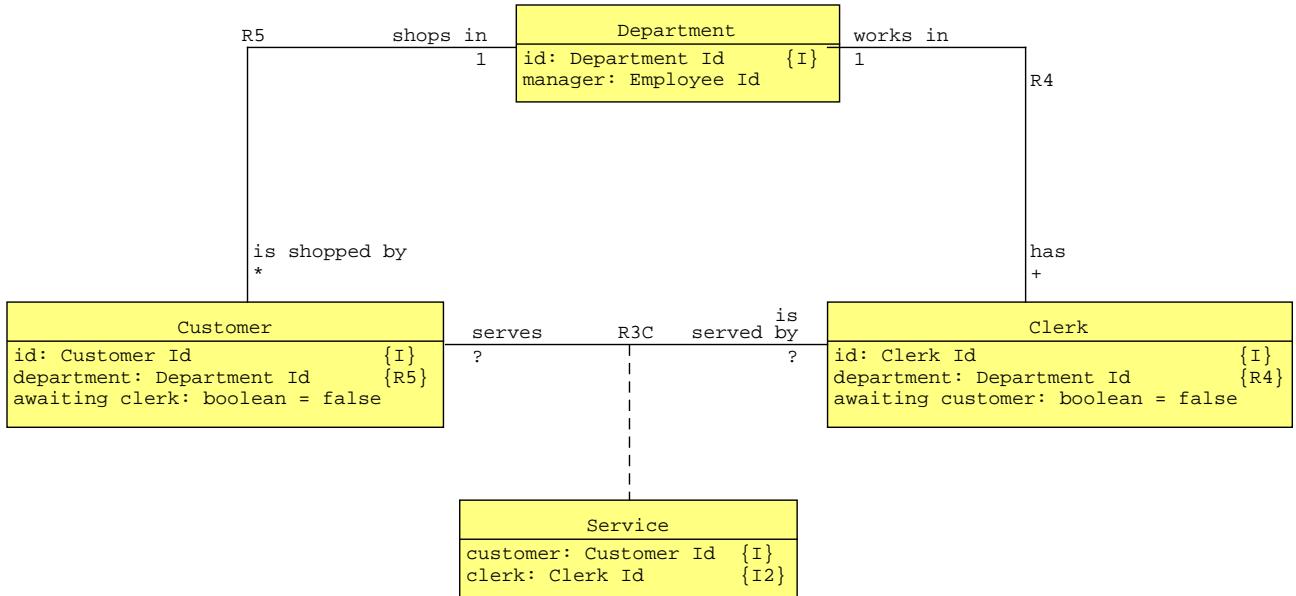


Figure 56. Customer / Clerk / Department Associations

Note first that there is the circular set of associations. This means that the associations are not completely independent of each other. The referential attributes of Service must be constrained in their values by the actions of the assigner to ensure that a Clerk is assigned only to Customers shopping in the department in which the Clerk works. Consider association R3C. We have labeled the association with a C to signify that the make up of the association is constrained. The store policy that requires Clerks to serve only those Customers that are shopping in the Department in which the Clerk works is logically equivalent to requiring that traversing from Customer to Clerk across R3C and then on to Department across R4 must yield the same Department as traversing from Customer directly to Department across R5. The store policy restricts the degrees of freedom inherent in the circular associations. To meet the requirement, a separate R3C assigner state machine is used for each Department since the competition between Customers and Clerks is *partitioned* on a per Department instance basis. Another way to think of this situation is to say that the Department class partitions the Customer and Clerk classes into equivalence classes. A separate assigner per Department is used to ensure that instances of R3C (and Service) are created such that any Customer assigned to a Clerk was shopping in the same Department in which that Clerk works.

The state model for the multiple assigner case is essentially the same as for the single assigner case. The significant difference is that now there are multiple assigner instances and those instances use the same identifying attributes as the partitioning class, Department in this case. The following figure shows the multiple assigner state model.

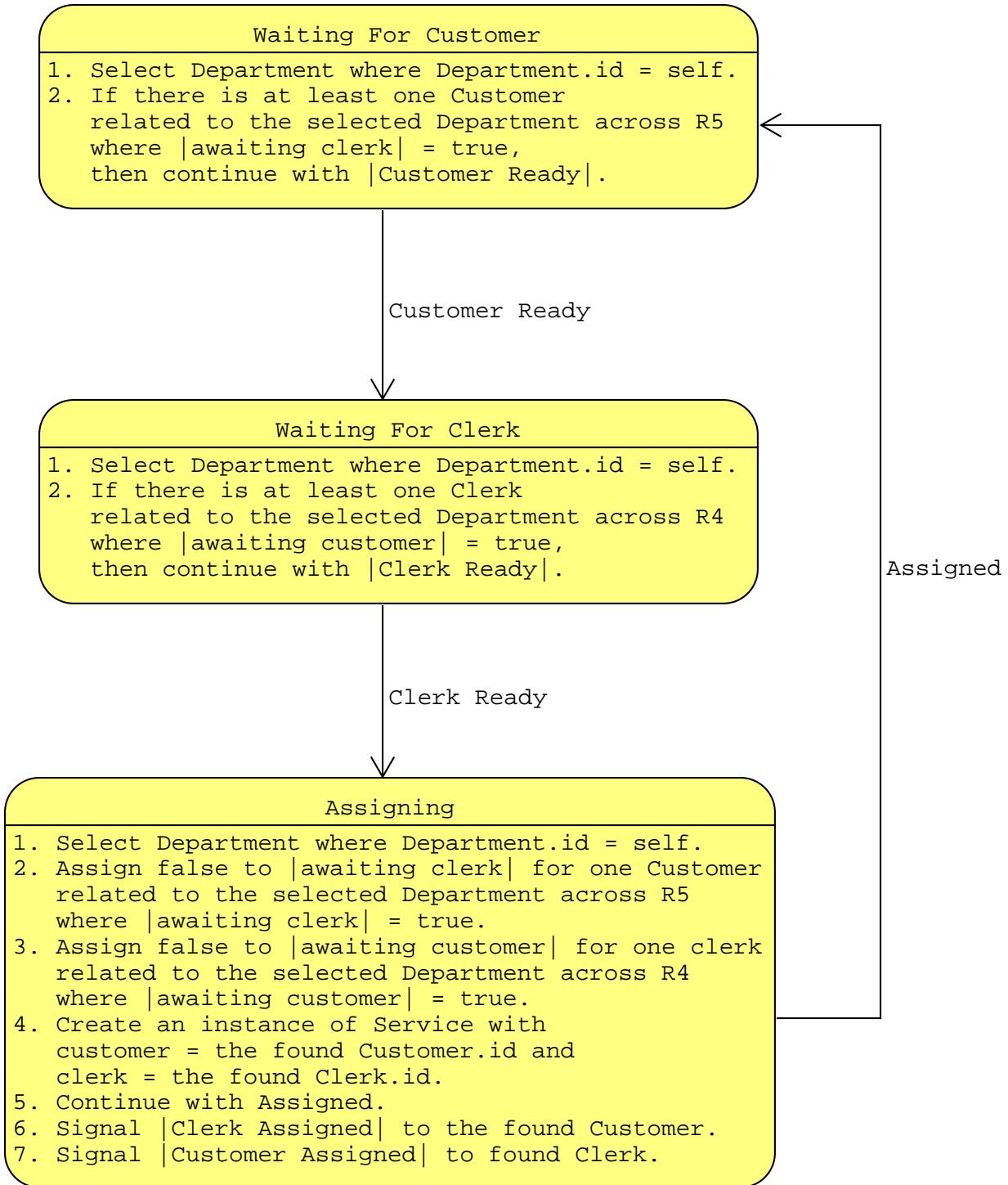


Figure 57. Customer / Clerk / Department Assigner State Model

Note that the assigner state machine instances are identified by the same identifying attributes as the Department. This provides the means to consider only those Customers and Clerks that are associated with the Department, which is identified in the same manner as the R3C assigner instances.

Multiple Assigner Implementation

The implementation of a multiple assigner follows closely from that of a single assigner. Since a multiple assigner has more than one instance and that instance is identified by the same attributes as the partitioning class, we can create the Relation Class that holds the assigner instances using the identifier for the partition class.

```
<<assigner module public functions>>=
pub fn MultiAssigner(
    comptime assigner_tag: @Type(.enum_literal),
    comptime state_model_spec: *const StateModelSpec,
    comptime Partition: type,
) type {
    if (!isRelClassType(Partition))
        @compileError("Partition argument is not a RelClass type");

    return struct {
        const Self = @This();

        const Heading = PartitionClass.IdentifierAttributes(0);
        pub const AssignerClass = RelClass(
            assigner_tag, // class_tag
            Heading, // Heading
            &.{ // identifiers
                enums.values(RelTuple(Heading).AttributeId),
            },
            &.{},
            .{
                .opt_state_model_spec = state_model_spec,
            },
            Partition.capacity, // max_capacity
            &{}, // initial_population
        );
        pub const assigner_class: AssignerClass = .{};
        pub const PartitionClass = Partition;
        pub const partition_class: PartitionClass = .{};

        /// The `init` function initializes the instance of the multiple assigner
        /// to the current set of instances of the partition class. This function
        /// must be invoked before the assigner is used.
        pub fn init() void {
            const partition_insts = partition_class.selectAll();
            var inst_iter = partition_insts.iterator();
            while (inst_iter.next()) |inst_ref| {
                // Create attribute values from the partition class which contain only the
                // fields of the 0th identifier.
                const assigner_attrs = PartitionClass.IdentifierValues(
                    0,
                    inst_ref.dereference(),
                );
                _ = assigner_class.create(assigner_attrs);
            }
        }
    };
}
```

```

    }

    /// The `selectPartitionInstance` returns an `InstanceSet`
    /// of cardinality one referencing the assigner partition
    /// class that matches the identifier of the multiple assigner
    /// instance itself. This function provides a way for multiple
    /// assigner state models to find the corresponding instance
    /// of the partition class and therefore have further access to
    /// the class data model.
    pub fn selectPartitionInstance(
        inst: InstanceRef,
    ) PartitionClass.InstanceRef {
        const id_values =
            PartitionClass.IdentifierValues(0, inst.dereference());
        return partition_class.selectByIdentifier(0, id_values) catch unreachable;
    }

    pub fn receiveEvent(
        _: Self,
        event: Event,
    ) bool {
        return assigner_class.receiveEvent(event);
    }

    pub const InstanceRef = AssignerClass.InstanceRef;
    pub const InstanceSet = AssignerClass.InstanceSet;
    pub const Event = AssignerClass.Event;
    pub const EventId = AssignerClass.EventId;
};

}

```

Tests

Multiple Assigner Tests

To test the multiple assigner code, we implement the previous multiple assigner example.

The multiple assigner example requires three classes. To avoid naming conflicts with the single assigner example, `_ma` is appended to the class names. As before, there is no implementation presence for the Service class since it contains no information other than referential attributes.

```

<<association module tests>>=
const DepartmentId = enum {
    womens_shoes,
    housewares,
    mens_clothing,
    mens_shoes,
};

const Department_ma = RelClass(
    .department_ma,

```

```

struct {
    id: DepartmentId,
    manager: []const u8,
},
&.{ &.{ .id } },
&.{ },
.&{ },
5,
&.{ 
    .{ .womens_shoes, .{ .id = .womens_shoes, .manager = "Joan" } },
    .{ .housewares, .{ .id = .housewares, .manager = "John" } },
    .{ .mens_clothing, .{ .id = .mens_clothing, .manager = "Jeeves" } },
},
);
};

const Customer_ma = RelClass(
    .customer_ma,
    struct {
        id: ClassGeneratedId = generated_id_default,
        awaiting_clerk: bool = false,
},
&.{ &.{ .id } },
&.{ },
.&{ },
10,
&.{ // Start with no initial instances. A customer is created before entering
    // a department.
},
);
;

const Clerk_ma = RelClass(
    .clerk_ma,
    struct {
        id: ClassGeneratedId = generated_id_default,
        awaiting_customer: bool = false,
},
&.{ &.{ .id } },
&.{ },
.&{ },
5,
&.{ 
    // We must have at least one clerk per department as demanded by R4.
    // One lonely clerk per department suffices.
    .{ .clerk1, .{} },
    .{ .clerk2, .{} },
    .{ .clerk3, .{} },
},
);
;
);

```

The addition of the Department class requires R4 and R5 to partition the Customer and Clerk sets.

<<association module tests>>

```

// Clerk +-> "has" -R4- "works in"-> Department
const r4_spec = AssociationSpec{
    .DomainRelClass = Clerk_ma,
    .codomain_ref = .@"+",
    .codomain_phrase = .has,
    .assoc_tag = .r4,
    .domain_phrase = .@"works in",
    .domain_ref = .@"1",
    .CodomainRelClass = Department_ma,
};

const R4 = Association(
    &r4_spec,
    &.{{
        // Assign the clerks to the various departments.
        .{ .clerk1, .womens_shoes },
        .{ .clerk2, .housewares },
        .{ .clerk3, .mens_clothing },
    }},
);

// Customer *-> "is shopped by" -R5- "shops in"-> Department
const r5_spec = AssociationSpec{
    .DomainRelClass = Customer_ma,
    .codomain_ref = .@"*",
    .codomain_phrase = .@"is shopped by",
    .assoc_tag = .r5,
    .domain_phrase = .@"shops in",
    .domain_ref = .@"1",
    .CodomainRelClass = Department_ma,
};

const R5 = Association(
    &r5_spec,
    &.{},
);

// Customer ?-> "serves" -R3C- "is served by"->? Clerk
const r3c_spec = AssociationSpec{
    .DomainRelClass = Customer_ma,
    .codomain_ref = .@"?",
    .codomain_phrase = .serves,
    .assoc_tag = .r3c,
    .domain_phrase = .@"is served by",
    .domain_ref = .@"?",
    .CodomainRelClass = Clerk_ma,
    .assigner_state_model = &r3c_state_model_spec,
    .PartitionClass = Department_ma,
};

const R3c = Association(
    &r3c_spec,
);

```

```

    &.{},
);

```

The transition table for the multiple assigner is the same as for the single assigner. It is repeated here for convenience.

	customer_ready	clerk_ready	assigned
waitingForCustomer	waitForClerk	ignore	ignore
waitingForClerk	ignore	assigning	ignore
assigning	never	never	waitingForCustomer

Table 16. Customer / Clerk Multiple Assigner Transition Matrix

The state model specification is also the same, but we make the necessary changes to accommodate the different association names.

```

<<association module tests>>=
const r3c_state_model_spec: StateModelSpec = .{
  .state_names = &.{ "waitingForCustomer", "waitingForClerk", "assigning" },
  .event_names = &.{ "customer_ready", "clerk_ready", "assigned" },
  .transitions = &.{ {
    "waitingForCustomer", &.{ {
      "customer_ready", "waitingForClerk" },
    },
  },
  .{
    "waitingForClerk", &.{ {
      "clerk_ready", "assigning" },
    },
  },
  .{
    "assigning", &.{ {
      "assigned", "waitingForCustomer" },
      .{ "customer_ready", "never" },
      .{ "clerk_ready", "never" },
    },
  },
},
.default_initial_state = "waitingForCustomer",
.StateActivities = r3c_assigner_activities,
.unspecified_transition = "ignore",
.terminal_states = &.{},
};

```

The action functions for the **R3C** assigner are direct transliteration of the structured description from the graphic. In this case, there is additional work to track the department wherein the Customer / Clerk interaction takes place.

```

<<association module tests>>=
const r3c_assigner_activities = struct {
    const log = std.log;
    const customer_ma: Customer_ma = .{};
    const clerk_ma: Clerk_ma = .{};
    const r4: R4 = .{};
    const r5: R5 = .{};
    const r3c: R3c = .{};
    const InstanceRef = R3c.InstanceRef;

    pub fn waitingForCustomer(
        self: InstanceRef,
    ) void {
        log.debug(
            "in 'waitingForCustomer' state: customer_ma =\n{any}\nclerk_ma =\n{any}",
            .{
                customer_ma.selectAll().relValueOf(),
                clerk_ma.selectAll().relValueOf(),
            },
        );
    }

    const department = r3c.selectPartitionInstance(self);
    if (department.traverseAssociation(r5, null, .customer_ma)
        .restrictEqual(.awaiting_clerk, true)
        .notEmpty())
    {
        self.continueWith(.customer_ready, .{});
    }
}

pub fn waitingForClerk(
    self: InstanceRef,
) void {
    log.debug(
        "in 'waitingForClerk' state: customer_ma =\n{any}\nclerk_ma =\n{any}",
        .{
            customer_ma.selectAll().relValueOf(),
            clerk_ma.selectAll().relValueOf(),
        },
    );
}

const department = r3c.selectPartitionInstance(self);
if (department.traverseAssociation(r4, null, .clerk_ma)
    .restrictEqual(.awaiting_customer, true)
    .notEmpty())
{
    self.continueWith(.clerk_ready, .{});
}

pub fn assigning(
    self: InstanceRef,

```

```

) void {
    log.debug(
        "in 'assigning' state: customer_ma =\n{any}\nclerk_ma =\n{any}",
        .{
            customer_ma.selectAll().relValueOf(),
            clerk_ma.selectAll().relValueOf(),
        },
    );

    const department = r3c.selectPartitionInstance(self);
    const ready_customer =
        department.traverseAssociation(r5, null, .customer_ma)
        .restrictEqual(.awaiting_clerk, true)
        .limit(1);
    const ready_clerk =
        department.traverseAssociation(r4, null, .clerk_ma)
        .restrictEqual(.awaiting_customer, true)
        .limit(1);

    if (ready_customer.notEmpty() and ready_clerk.notEmpty()) {
        ready_customer.updateAttribute(.awaiting_clerk, false);
        ready_clerk.updateAttribute(.awaiting_customer, false);
        r3c.relate(
            ready_customer.instanceRefFromInstanceSet(),
            .@"is served by",
            ready_clerk.instanceRefFromInstanceSet(),
            null,
        );
    }

    // In practice, customer and clerk would both have state models
    // and the assigner would signal them at this point to indicate
    // the assignment. For testing, the extended interactions are omitted.
} else {
    unreachable; // See comment for the single assigner example.
}

self.continueWith(.assigned, .{});
}
};

}

```

For the test case itself, the scenario used has two customer shopping in the same department and another customer shopping in a different department. This show both contention between the customers shopping in the same department and still allowing the other customer to shop as long as the department is different.

```

<<association module tests>>=
test MultiAssigner {
    testing.log_level = .info;

    const department_ma = Department_ma.init(0);
    const customer_ma = Customer_ma.init(1);
    const clerk_ma = Clerk_ma.init(2);

```

```

const r3c = R3c.init(3);
const r4: R4 = .{};
const r5: R5 = .{};

// R4 must be consistent as a result of its initial associations
try testing.expect(r4.isConsistent());

// Bring a Clerk on duty for each department.
const clerk_1 = clerk_ma.selectInitialInstance(.clerk1);
clerk_1.updateAttribute(.awaiting_customer, true);
const clerk_2 = clerk_ma.selectInitialInstance(.clerk2);
clerk_2.updateAttribute(.awaiting_customer, true);
const clerk_3 = clerk_ma.selectInitialInstance(.clerk3);
clerk_3.updateAttribute(.awaiting_customer, true);

// To demonstrate contention, all the customers arrive at once.
// Create Customers as they enter Departments and relate them by R5
// to the department where they begin shopping.

// The first customer enters the women's shoes department.
const customer_1 = customer_ma.create(.{});
const ws_dept = department_ma.selectInitialInstance(.womens_shoes);
r5.relate(customer_1, .@"shops in", ws_dept, null);
try testing.expect(r5.isConsistent());
customer_1.updateAttribute(.awaiting_clerk, true);
const r3c_ws = try r3c.selectByIdentifier(0, .{.id = .womens_shoes});
_= r3c.receiveEvent(r3c_ws.makeEvent(.customer_ready, .{}));

// Since there is a ready clerk, an assignment is made.
try testing.expect(r3c.isConsistent());
// Test that R3C is properly constrained such that the
// assigned customer and clerk are both related to the same
// department.
const customer_1_dept = customer_1.traverseAssociation(r5, null, .department_ma);
const customer_1_clerk_dept = customer_1
    .traverseAssociation(r3c, null, .clerk_ma)
    .traverseAssociation(r4, null, .department_ma);
try testing.expectEqual(
    customer_1_dept.readAttribute(.id),
    customer_1_clerk_dept.readAttribute(.id),
);
;

// The second customer arrives and also enters the women's shoes department.
// In this case, we don't expect any assignment since our lonely clerk is
// servicing customer 1.
const customer_2 = customer_ma.create(.{});
r5.relate(customer_2, .@"shops in", ws_dept, null);
try testing.expect(r5.isConsistent());
customer_2.updateAttribute(.awaiting_clerk, true);
_= r3c.receiveEvent(r3c_ws.makeEvent(.customer_ready, .{}));

// The third customer arrives and enters the housewares department.

```

```

const customer_3 = customer_ma.create({});  

const hw_dept = department_ma.selectInitialInstance(.housewares);  

r5.relate(customer_3, @"shops in", hw_dept, null);  

try testing.expect(r5.isConsistent());  

customer_3.updateAttribute(.awaiting_clerk, true);  

const r3c_hw = try r3c.selectByIdentifier(0, {.id = .housewares});  

_ = r3c.receiveEvent(r3c_hw.makeEvent(.customer_ready, {}));  
  

// The third customer arrived in a different department and so we expect  

// that customer to have been assigned to the lonely clerk in the  

// housewares department, i.e. clerk_2.  

try testing.expect(r3c.isConsistent());  

const hw_customer = clerk_2.traverseAssociation(r3c, null, .customer_ma);  

try testing.expectEqual(@as(usize, 1), hw_customer.cardinality());  

try testing.expectEqual(customer_3.readAttribute(.id), hw_customer.readAttribute(.id));  
  

// Customer 1 finishes shopping and leaves the department.  

// Clerk 1 can now serve another Customer.  

_ = r5.unrelateInstance(customer_1, @"shops in");  

_ = r3c.unrelateInstance(customer_1, @"is served by");  

customer_1.destroy();  

try testing.expect(r5.isConsistent());  

try testing.expect(r3c.isConsistent());  

clerk_1.updateAttribute(.awaiting_customer, true);  

_ = r3c.receiveEvent(r3c_ws.makeEvent(.clerk_ready, {}));  
  

// Customer 3 finishes shopping and leaves the department.  

_ = r5.unrelateInstance(customer_3, @"shops in");  

_ = r3c.unrelateInstance(customer_3, @"is served by");  

customer_3.destroy();  

try testing.expect(r5.isConsistent());  

try testing.expect(r3c.isConsistent());  

clerk_2.updateAttribute(.awaiting_customer, true);  

_ = r3c.receiveEvent(r3c_hw.makeEvent(.clerk_ready, {}));  
  

// Customer 2 finishes shopping and leaves the department.  

_ = r5.unrelateInstance(customer_2, @"shops in");  

_ = r3c.unrelateInstance(customer_2, @"is served by");  

customer_2.destroy();  

try testing.expect(r5.isConsistent());  

try testing.expect(r3c.isConsistent());  

clerk_1.updateAttribute(.awaiting_customer, true);  

_ = r3c.receiveEvent(r3c_ws.makeEvent(.clerk_ready, {}));  
  

// At this point, there are no more customers and all  

// the clerks are available to serve when new customers arrive.  

try testing.expectEqual(@as(usize, 0), customer_ma.cardinality());  

const available_clerks = clerk_ma.selectEqual(.awaiting_customer, true);  

try testing.expectEqual(@as(usize, 3), available_clerks.cardinality());  

}

```

Sometimes, partitioning class instances are created or destroyed. This means the corresponding

multiple assigner class instances must also be created or destroyed.

```
<<association module tests>>=
test "dynamic assigners" {
    const department_ma: Department_ma = .{};
    const clerk_ma: Clerk_ma = .{};
    const r3c: R3c = .{};
    const r4: R4 = .{};
    const r5: R5 = .{};

    const ms_dept = department_ma.create(.{ .id = .mens_shoes, .manager = "Fred" });
    const ms_clerk = clerk_ma.create(.{});
    r4.relate(ms_clerk, .@"works in", ms_dept, null);
    try testing.expect(r4.isConsistent());
    try testing.expect(r5.isConsistent());
    const ms_assigner = r3c.create(.{.id = .mens_shoes});
    try testing.expectEqual(@as(usize, 4), R3c.AssignerModel.assigner_class.cardinality());

    const unrelated_dept = r4.unrelateInstance(ms_clerk, .@"works in");
    ms_clerk.destroy();
    unrelated_dept.destroy();
    try testing.expect(r4.isConsistent());
    ms_assigner.destroy();
    try testing.expectEqual(@as(usize, 3), R3c.AssignerModel.assigner_class.cardinality());
}
```

Assigner Interface

Since assigners are attached to Associations, we bring out the required functions to deal with the assigner state models for those Associations that have an attached assigner.

```
<<association declarations>>=
pub const has_assigner_model = association_spec.assigner_state_model != null;
pub const has_single_assigner = has_assigner_model and association_spec.PartitionClass == null;
pub const has_multi_assigner = has_assigner_model and association_spec.PartitionClass != null;

pub const AssignerModel = if (has_single_assigner)
    Assigner(association_spec.assoc_tag, association_spec.assigner_state_model.?)
else if (has_multi_assigner)
    MultiAssigner(
        association_spec.assoc_tag,
        association_spec.assigner_state_model.?,
        association_spec.PartitionClass.?,
    )
else
    @compileError(
        @tagName(association_spec.assoc_tag) ++
        ": association has no state model");
```

```

pub const assigner_model = if (has_assigner_model)
    AssignerModel{}
else
    @compileError(
        @tagName(association_spec.assoc_tag) ++
        ": association has no state model");

pub fn initAssigner() void {
    if (has_multi_assigner) _ = AssignerModel.init();
}

pub const Event = if (has_assigner_model)
    AssignerModel.Event
else
    @compileError(
        @tagName(association_spec.assoc_tag) ++
        ": association has no state model");

pub const EventId = if (has_assigner_model)
    AssignerModel.EventId
else
    @compileError(
        @tagName(association_spec.assoc_tag) ++
        ": association has no state model");

pub fn receiveEvent(
    _: Self,
    event: Event,
) bool {
    return if (has_assigner_model)
        assigner_model.receiveEvent(event)
    else
        @compileError(
            @tagName(association_spec.assoc_tag) ++
            ": association has no state model");
}

pub const InstanceSet = if (has_assigner_model)
    AssignerModel.InstanceSet
else
    @compileError(
        @tagName(association_spec.assoc_tag) ++
        ": association has no state model");

pub const InstanceRef = if (has_assigner_model)
    AssignerModel.InstanceRef
else
    @compileError(
        @tagName(association_spec.assoc_tag) ++
        ": association has no state model");

pub fn create()

```

```

_: Self,
tuple: AssignerModel.AssignerClass.TupleBase,
) InstanceRef {
    return if (has_multi_assigner)
        AssignerModel.assigner_class.create(tuple)
    else
        @compileError(
            @tagName(association_spec.assoc_tag) ++
            ": association has no state model");
}

pub fn select(
    _: Self,
) InstanceRef {
    return if (has_single_assigner)
        assigner_model.select()
    else
        @compileError(
            @tagName(association_spec.assoc_tag) ++
            ": association has no state model");
}

pub fn selectByIdentifier(
    _: Self,
    comptime ident_num: usize,
    id_attr_values: AssignerModel.AssignerClass.IdentifierAttributes(ident_num),
) RelVarError!InstanceRef {
    return if (has_multi_assigner)
        try AssignerModel.assigner_class.selectByIdentifier(ident_num, id_attr_values)
    else
        @compileError(
            @tagName(association_spec.assoc_tag) ++
            ": association is a single assigner or has no state model");
}

pub fn selectPartitionInstance(
    _: Self,
    inst: InstanceRef,
) AssignerModel.PartitionClass.InstanceRef {
    return if (has_multi_assigner)
        AssignerModel.selectPartitionInstance(inst)
    else
        @compileError(
            @tagName(association_spec.assoc_tag) ++
            ": association is a single assigner or has no state model");
}

pub fn selectAll() InstanceSet {
    return if (has_multi_assigner)
        AssignerModel.assigner_class.selectAll()
    else

```

```

    @compileError(
        @tagName(association_spec.assoc_tag) ++
        ": association is a single assigner or has no state model");
}

```

Assigner Code Layout

```

<<assigner.zig>>=
///! The `assigner.zig` file implements the concept of an assigner association.
///! An assigner association has an attached state model the synchronize the
///! creation of association instances where the association relationship is
///! competitive. The competition is usually concerned with the allocation of
///! resources. This code builds upon both the `RelClass` type function and
///! the `StateModel` type function.
<<edit warning>>
<<copyright info>>

const std = @import("std");
const builtin = @import("builtin");

const enums = std.enums;
const meta = std.meta;
const fmt = std.fmt;
const StructField = std.builtin.Type.StructField;
const panic = std.debug.panic;
const assert = std.debug.assert;

const RelVarError = @import("relvar.zig").RelVarError;

const class = @import("class.zig");
const RelClass = class.RelClass;
const isRelClassType = class.isRelClassType;

const state_model = @import("state_model.zig");
const StateModelSpec = state_model.StateModelSpec;
const StateModel = state_model.StateModel;

const reltuple = @import("reltuple.zig");
const RelTuple = reltuple.RelTuple;

pub const version = "1.0.0-a1";

<<assigner module declarations>>

<<assigner module public functions>>

<<assigner module private functions>>

<<assigner module tests>>

```

Generalization Behavior

A polymorphic event can arise only in the context of a generalization relationship. Events can be designated as polymorphic and when such an event is signaled to a superclass instance, it is mapped at run time to an event of the subclass instance that is currently related to superclass instance. The net effect is to allow the subclasses of a generalization all to respond to the same events and yet the events are signaled to the superclass instances of the generalization. This alleviates the burden on the signalling class to determine the subclass instance to which the superclass instance is currently related.

Event polymorphism can be viewed as an optimization performed by the event dispatch mechanism. Technically, mapping an event to a subclass could be performed as part of a state activity function. However, such code is repetitive and rather fragile in the wake of adding or deleting subclasses in a generalization. This is a case where the system can “know” the right thing to do and remove a significant burden from the model level processing.

In its most common usage, polymorphic events are quite straight forward. Events are designated as polymorphic by the superclass and they are used as transitioning events by the state models of the subclasses. In its most general usage however, we must account for both repeated generalization (*i.e.* where a subclass serves as the superclass for a further generalization) and compound generalization (*i.e.* where a class is the superclass for multiple generalization relationships). The full set of rules and implications of polymorphic events can be rather daunting.

- A polymorphic event has no affect on the behavior of the superclass that defines it. Associating polymorphic events with a superclass does *not* imply that the superclass has no state behavior of its own. A superclass may have both a state model and define polymorphic events.
- A given superclass may be the superclass of multiple generalizations. In this case, generating an event to an instance of such a superclass will cause an event to be generated to all generalizations for which the class is a superclass. In this way, signalling a single event may result in multiple events being dispatched.
- The state model for a subclass may respond to transitioning events that are not part of the polymorphic event set associated with the generalization. Such events may be signalled directly to instances of the subclass or they may be signalled by a subclass instance itself.
- A class that is a subclass may also be a superclass of another generalization, *i.e.* a subclass may be subject to repeated generalization. Such a mid-level class may designate additional polymorphic events associated with the generalization relationship for which it is the superclass. Also a mid-level class may delegate polymorphic events associated with its subclass role to any hierarchy for which it serves as a superclass.
- A mid-level class may have a state model that consumes an forwarded polymorphic event. In that case, the event is not forwarded down any other generalization for which the mid-level class is a superclass.
- All leaf classes, *i.e.* subclasses which are *not* the superclass of another generalization, must consume as transitioning events all polymorphic events delegated to them. It is sufficient to ignore an event or deem an event as undefined (never), but polymorphic events ultimately are mapped to transitioning events when the bottom of the generalization hierarchy is reached.

Note also that generalizations are *not* subject to the so called “diamond” construct where a given class is a subclass of multiple generalizations which themselves have a common superclass ancestor. Blessedly, referential integrity simply does not allow such a beast to be constructed.

Rules for Polymorphic Event propagation.

We can make the previous set of requirements more precise by defining some terms.

1. Events are either Effective or Delegated.
 - Delegated events are those that have been forwarded along on a generalization hierarchy to a subclass of a generalization.
 - Ultimate superclasses (*i.e.* those that do not serve as a domain for any generalization) have no forwarded events.
 - Effective events are all other events for any role a class plays in an association or generalization. Effective events are those typically defined on a state model.
2. Polymorphic events can be defined on a Class that serves in the role of superclass (codomain) of a Generalization.
 - A Polymorphic event must be distinct from any Effective events or Deleted events that are defined or forwarded to the superclass.
3. A class forwards downward to all subclasses of all generalizations for which it serves as a superclass (codomain) a set of Delegated events.. The Delegated events forwarded down are the union of any Polymorphic events and any forwarded events from a superclass.
4. Any Delegated event that is consumed by a state model attached to the Class becomes an Effective event and is *not* delegated any further along the generalization.
 - The set of Delegated events for ultimate subclasses (*i.e.* those that do not serve as a codomain for any generalization) must be empty, *i.e.* all events forwarded down the generalization must be consumed by leaf subclasses.

Bringing in the Sheaves



This chapter is currently under active development and significant changes are anticipated in the next release.

Early in Part II of this book, the [core model concepts](#) were presented. Previous chapters expanded on the core concepts using concrete designs and code to realize these concepts. We started with an unadorned relational algebra library which forms the basis of the ReSEE mechanisms for handling application program data. The intervening chapters continued to add software technology to the code model concepts with implementation of state models, refining the relational algebra, and adding execution behavior to both Relational Classes and the relationships between them.

The approach continues to be from the bottom up but how the data, dynamics, and processing facets operate together must now be considered.

In the [execution model concepts](#) discussion, the term *domain* is introduced. We have also used that term informally on occasions. It is now time to be more specific about what requirements of the execution model are delegated to domain and exactly what role a domain serves in the execution model.



Once again the definition of the term, *domain*, must be clarified.. When discussing binary relations we used the term domain as one of the sets of relational tuples participating in the relation. A binary relation is a mapping between two sets called the domain and codomain. Now we intend to use the term domain in the sense of a domain of discourse. Specifically, our notion of domain gathers together the set of entities upon which our model is based. In terms of our model concepts, domains are composed of Relational Classes, Associations, and Generalizations. It is usually clear from context which type of domain is being discussed and we will clarify the usage when necessary.

The precise way that a system is decomposed into domains is a creative output of the system analysis. Some domains of an application are clear. The primary purpose^[1] of the application is the subject matter for the application domain. Application domains are specific to the primary purpose and there are no expectations that a subject matter decomposition of some particular software system application can be reused to produce other systems with a different purpose. Other domains exist to support requirements delegated to them by the application domain. These domains are called *service domains* and usually describe subject matter that are independent of a particular application. For example, a control system may be required to produce alarm conditions to its human operators when it detects circumstances that require intervention. Alarming can be treated as a world of its own, defining policies about severity or acknowledgement times that are completely independent of whatever control over the environment the system is producing.

A *domain* is an autonomous, real, hypothetical or abstract world inhabited by a set of conceptual entities that behave according to characteristic rules and policies [\[mb-xuml\]](#), p.30.

— Mellor and Balcer, Executable UML: A Foundation for Model-Driven Architecture

That definition is quite dense as it tries to cover all the semantic uses of the domain concept. The key concept is that a domain defines its own world view and behaves according to specific rules and policies. The job of an analyst is to understand the problem at hand well enough to capture its solution into domains consisting of specific the entities, rules and policies. Our job here is to show how a domain can be turned into a running program that directly uses those captured entities, rules and policies.

Domain Requirements

For the ReSEE execution environment, a domain plays a coordinating role between the entities inside the domain and those entities outside the domain. That role can be described as:

- Since the domain is the unit of encapsulation, a domain serves as a container for the classes, associations, and generalizations it holds.
- The primary mechanism for sequencing execution is having state machines signal events to other state machines. It is the responsibility of the domain to provide the mechanisms required to sequence execution by events.
- The set of classes and relationships of a domain form a constrained data space where the multiplicity and conditionality of the relationships limit the possible set membership arrangement of class instances. The domain is responsible to ensure that execution which modifies the data facet of the model leaves its constraints invariant.
- The domain provides the means for the domain entities to interact with other domains or with the system environment. A domain in isolation is highly cohesive but has no coupling to anything else. Such an isolated arrangement would imply the domain would be unable to produce any side effects outside of itself. Yet for a reactive system, producing side effect into the system environment is the primary goal. Therefore, some coupling of a domain is necessary and it is the domain that manages that coupling. This is an important area and Part III of the book is devoted to this topic.

Defining a Domain

```
<<domain module declarations>>=
pub fn Domain(
    comptime domain_tag: @Type(.enum_literal),
    comptime Classes: []const type,
    comptime Associations: []const type,
    comptime Generalizations: []const type,
    comptime opt_event_queue_count: ?u8,
) type {
    comptime validateClasses(Classes);
    comptime validateAssociations(Associations);
    comptime validateGeneralizations(Generalizations);

    return struct {
        const Self = @This();

        <<domain variables>>
        <<domain declarations>>
        <<domain public functions>>
        <<domain private functions>>
    };
}
```

```
<<domain module private functions>>=
fn validateClasses(
    comptime Classes: []const type,
) void {
```

```

if (Classes.len == 0) @compileError("a domain must have at least one class");

inline for (Classes) |Class| {
    if (comptime !isRelClassType(Class))
        @compileError("domain class, '" ++
                     @typeName(Class) ++
                     "', is not a RelClass type");
}
}

```

```

<<domain module private functions>>=
fn validateAssociations(
    comptime Associations: []const type,
) void {
    inline for (Associations) |Assoc| {
        if (comptime !isAssociationType(Assoc))
            @compileError("domain association, '" ++
                          @typeName(Assoc) ++
                          "', is not an Association type");
    }
}

```

```

<<domain module private functions>>=
fn validateGeneralizations(
    comptime Generalizations: []const type,
) void {
    inline for (Generalizations) |Gen| {
        if (comptime !isGeneralizationType(Gen))
            @compileError("domain generalization, '" ++
                          @typeName(Gen) ++
                          "', is not an Generalization type");
    }
}

```

```

<<domain variables>>=
pub const domain_id = domain_tag;
pub var domain_index: u8 = 0;

```

```

<<domain declarations>>=
pub const log = std.log.scoped(domain_id);

```

Gathering Information about Classes

The basis for a domain ability to sequence execution and ensure data integrity requires storing information about the collection of entity that constitute the domain. To that end, we define a three primary *catalogs* which simply register the class, association, and generalization entities that define the domain.

Class Catalog

The `Classes` argument to the `Domain` type function gives the set of relational class types that are to be encapsulated in the domain. To identify a class, we create an enumeration based on the `class_id` field of a Relational Class. One complication arises from those associations having an assigner. Assigners have a backing `RelClass`, so it is necessary to scan the `Associations` of the domain to find any `RelClass` used by an assigner.

Note that a `Domain` must contain at least one `RelClass`. Empty domains are meaningless. However, it is possible to have a `Domain` consisting of only classes with no associations or generalizations.

```
<<domain declarations>>=
/// An enumerated type that has tags for each relational class included
/// in the domain.
pub const ClassId = TagClasses();
pub const any_class_id = math.maxInt(u8);
```

```
/// The `TagClasses` function generates an enumeration type that has
[source,zig]
/// a tag for each relational class given by the `Classes` argument.
<<domain private functions>>=
fn TagClasses() type {
    comptime var class_id_fields: []const EnumField = &[_]EnumField{};
    comptime var class_index = 0;
    inline for (Classes) |Class| {
        class_id_fields = class_id_fields ++ &[_]EnumField{
            .{
                .name = @tagName(Class.class_id),
                .value = class_index,
            },
        };
        class_index += 1;
    }

    inline for (Associations) |Assoc| {
        if (Assoc.has_assigner_model) {
            const AssignerRelClass = @TypeOf(Assoc.AssignModel.relclass);
            class_id_fields = class_id_fields ++ &[_]EnumField{
                .{
                    .name = @tagName(AssignerRelClass.class_id),
                    .value = class_index,
                },
            };
            class_index += 1;
        }
    }

    return @Type(.{
        .@"_enum" = .{
            .tag_type = u8, // strict upper limit on the number of classes in a domain
            .fields = class_id_fields,
            .decls = &.{},
        },
    });
}
```

```

        .is_exhaustive = true,
    },
});
}

```

We define a namespace `struct` to hold the class catalog and a set of methods to access the catalog information.

```

<<domain declarations>>=
/// A namespace `struct` that holds the information about the class makeup
/// of the domain.
pub const class_info = struct {
    <<class info declarations>>
};

```

```

<<class info declarations>>=
/// The type of an entry in the class catalog.
const ClassCatalogEntry = MakeClassCatalogEntry();

```

```

<<class info declarations>>=
/// The `MakeClassCatalogEntry` type function creates a tagged union of
/// the classes in a domain. The union is tagged by `ClassId` and each
/// element of the union is the `RelClass` type for the corresponding
/// class.
fn MakeClassCatalogEntry() type {
    var entry_fields: []const UnionField = &[_]UnionField{};
    inline for (comptime enums.values(ClassId), Classes) |class_id, Class| {
        entry_fields = entry_fields ++ &[_]UnionField{
            .{
                .name = @tagName(class_id),
                .type = Class,
                .alignment = @alignOf(Class),
            },
        };
    }

    inline for (Associations) |Assoc| {
        if (Assoc.has_assigner_model) {
            const AssignerRelClass = @TypeOf(Assoc.AssignModel.relclass);
            entry_fields = entry_fields ++ &[_]UnionField{
                .{
                    .name = @tagName(AssignerRelClass.class_id),
                    .type = AssignerRelClass,
                    .alignment = @alignOf(AssignerRelClass),
                },
            };
        }
    }

    return @Type(.{

```

```

        .@"union" = .{
            .layout = .auto,
            .tag_type = ClassId,
            .fields = entry_fields,
            .decls = &.{},
        },
    });
}

```

```

<<class info declarations>>=
/// An instantiation of the class catalog for the domain.
/// The catalog is an array of `ClassCatalogEntry` typed values.
const class_catalog = makeClassCatalog();

```

```

<<class info declarations>>=
/// The `makeClassCatalog` function creates an array of `ClassCatalogEntry` values,
/// one for each `RelClass` defined for the domain. The array is indexed by
/// `@intFromEnum(class_id)` where `class_id` is of type `ClassId`.
fn makeClassCatalog() enums.EnumArray(ClassId, ClassCatalogEntry) {
    comptime var catalog: enums.EnumArray(ClassId, ClassCatalogEntry) = .initUndefined();
    comptime var iter = catalog.iterator();
    inline while (comptime iter.next()) |entry| {
        entry.value.* = @unionInit(ClassCatalogEntry, @tagName(entry.key), .{});
    }

    return catalog;
}

```

The remaining methods of `class_info` perform operations to "lift" `RelClass` introspect to the domain level by using the `class_catalog`.

```

<<class info declarations>>=
pub fn init() void {
    // We justify the following @constCast() on the basis that there is no
    // const iterator function for an EnumArray, and we do not modify the
    // contents of the array in this operation. The class_catalog is built
    // at compile time and not ever modified.
    comptime var iter = @constCast(&class_catalog).iterator();
    inline while (comptime iter.next()) |entry| {
        const Class = @FieldType(ClassCatalogEntry, @tagName(entry.key));
        _ = Class.init(@intFromEnum(entry.key));
        Class.executor = executor;
    }
}

```

```

<<class info declarations>>=
pub fn getCatalogEntry(
    class_id: ClassId,
) ClassCatalogEntry {

```

```
    return class_catalog.get(class_id);
}
```

```
<<class info declarations>>=
pub fn GetClass(
    comptime class_id: ClassId,
) type {
    return @FieldType(ClassCatalogEntry, @tagName(class_id));
}
```

```
<<class info declarations>>=
pub fn getClassVar(
    comptime class_id: ClassId,
) class_info.GetClass(class_id) {
    const catalog_entry = comptime class_catalog.get(class_id);
    return @field(catalog_entry, @tagName(class_id));
}
```

```
<<class info declarations>>=
pub fn Tuple(
    comptime class_id: ClassId,
) type {
    return GetClass(class_id).RelTuple;
}
```

```
<<class info declarations>>=
pub fn AttributeId(
    comptime class_id: ClassId,
) type {
    return GetClass(class_id).AttributeId;
}
```

```
<<class info declarations>>=
pub fn AttributeType(
    comptime class_id: ClassId,
    comptime attr_id: AttributeId(class_id),
) type {
    return GetClass(class_id).AttributeType(attr_id);
}
```

```
<<class info declarations>>=
pub fn IdentifierAttributes(
    comptime class_id: ClassId,
    comptime ident_num: usize,
) type {
    return GetClass(class_id).IdentifierAttributes(ident_num);
}
```

```
<<class info declarations>>=
pub fn InstanceSet(
    comptime class_id: ClassId,
) type {
    return GetClass(class_id).InstanceSet;
}
```

```
<<class info declarations>>=
pub fn InstanceRef(
    comptime class_id: ClassId,
) type {
    return GetClass(class_id).InstanceRef;
}
```

```
<<class info declarations>>=
pub fn Event(
    comptime class_id: ClassId,
) type {
    return GetClass(class_id).Event;
}
```

Gathering Information about Associations

Following the same pattern as for `Classes` we collect information about the `Associations` and build an `association_catalog` to hold it and an `association_info` namespace to supply methods on the catalog. Note that in the case of associations, the `Associations` slice argument may be empty.

```
<<domain declarations>>=
/// An enumerated type that has tags for each association included in the domain.
pub const AssociationId = if (Associations.len != 0) TagAssociations() else void;
```

```
<<domain private functions>>=
fn TagAssociations() type {
    comptime var assoc_id_fields: []const EnumField = &[_]EnumField{};
    inline for (Associations, 0..) |Assoc, assoc_index| {
        assoc_id_fields = assoc_id_fields ++ &[_]EnumField{
            .{
                .name = @tagName(Assoc.assoc_id),
                .value = assoc_index,
            },
        };
    }

    return @Type(.{
        .@"enum" = .{
            .tag_type = math.IntFittingRange(0, assoc_id_fields.len),
            .fields = assoc_id_fields,
            .decls = &.{},
        },
    });
}
```

```

        .is_exhaustive = true,
    },
});
}

```

```

<<domain declarations>>=
pub const association_info = struct {
    <<association info declarations>>
};

```

```

<<association info declarations>>=
const AssocCatalogEntry = if (AssociationId != void) MakeAssocCatalogEntry() else void;

```

```

<<association info declarations>>=
fn MakeAssocCatalogEntry() type {
    if (Associations.len != 0) {
        var entry_fields: []const UnionField = &[_]UnionField{};
        inline for (comptime enums.values(AssociationId), Associations) |assoc_id, Assoc| {
            entry_fields = entry_fields ++ &[_]UnionField{
                .{
                    .name = @tagName(assoc_id),
                    .type = Assoc,
                    .alignment = @alignOf(Assoc),
                },
            };
        }

        return @Type(.{
            .@("union") = .{
                .layout = .auto,
                .tag_type = AssociationId,
                .fields = entry_fields,
                .decls = &.{},
            },
        });
    } else return void;
}

```

```

<<association info declarations>>=
pub const association_catalog = if (AssociationId != void) makeAssociationCatalog() else void{};

```

```

<<association info declarations>>=
fn makeAssociationCatalog() enums.EnumArray(AssociationId, AssocCatalogEntry) {
    comptime var catalog: enums.EnumArray(AssociationId, AssocCatalogEntry) =
.initUndefined();
    comptime var iter = catalog.iterator();
    inline while (comptime iter.next()) |entry| {

```

```

        entry.value.* = @unionInit(AssocCatalogEntry, @tagName(entry.key), .{});
    }

    return catalog;
}

```

```

<<association info declarations>>=
pub fn init() void {
    if (Associations.len != 0) {
        // We justify the following @constCast() on the basis that there is no
        // const iterator function for an EnumArray, and we do not modify the
        // contents of the array in this operation. The association_catalog is built
        // at compile time and not ever modified.
        comptime var iter = @constCast(&association_catalog).iterator();
        inline while (comptime iter.next()) |entry| {
            const Assoc = @FieldType(AssocCatalogEntry, @tagName(entry.key));
            _ = Assoc.init(@intFromEnum(entry.key));
        }
    }
}

```

```

<<association info declarations>>=
pub fn GetAssociation(
    comptime assoc_id: AssociationId,
) type {
    return @FieldType(AssocCatalogEntry, @tagName(assoc_id));
}

```

```

<<association info declarations>>=
pub fn DomainOf(
    comptime assoc_id: AssociationId,
) type {
    return GetAssociation(assoc_id).DomainClass;
}

```

```

<<association info declarations>>=
pub fn CodomainOf(
    comptime assoc_id: AssociationId,
) type {
    return GetAssociation(assoc_id).CodomainClass;
}

```

```

<<association info declarations>>=
pub fn DomainInstanceSet(
    comptime assoc_id: AssociationId,
) type {
    return GetAssociation(assoc_id).DomainInstanceSet;
}

```

```
<<association info declarations>>=
pub fn CodomainInstanceSet(
    comptime assoc_id: AssociationId,
) type {
    return GetAssociation(assoc_id).CodomainInstanceSet;
}
```

```
<<association info declarations>>=
pub fn DomainInstanceRef(
    comptime assoc_id: AssociationId,
) type {
    return GetAssociation(assoc_id).DomainInstanceRef;
}
```

```
<<association info declarations>>=
pub fn CodomainInstanceRef(
    comptime assoc_id: AssociationId,
) type {
    return GetAssociation(assoc_id).CodomainInstanceRef;
}
```

Gathering Information about Generalizations

Finally, we follow the same pattern for Generalizations. Like Associations, the Generalizations argument may be an empty slice.

```
<<domain declarations>>=
pub const GeneralizationId = if (Generalizations.len != 0) TagGeneralizations() else void;
```

```
<<domain private functions>>=
fn TagGeneralizations() type {
    comptime var gen_id_fields: []const EnumField = &[_]EnumField{};
    inline for (Generalizations, 0..) |Generalization, gen_index| {
        gen_id_fields = gen_id_fields ++ &[_]EnumField{
            .{
                .name = @tagName(Generalization.gen_id),
                .value = gen_index,
            },
        };
    }

    return @Type(.{
        .@enum" = .{
            .tag_type = math.IntFittingRange(0, gen_id_fields.len),
            .fields = gen_id_fields,
            .decls = &.{},
            .is_exhaustive = true,
        },
    },
}
```

```
});  
}
```

```
<<domain declarations>>=  
pub const generalization_info = struct {  
    <<generalization info declarations>>  
};
```

```
<<generalization info declarations>>=  
const GenCatalogEntry = if (GeneralizationId != void) MakeGenCatalogEntry() else void;  
pub const generalization_catalog = if (GeneralizationId != void)  
makeGeneralizationCatalog() else void{};
```

```
<<generalization info declarations>>=  
fn MakeGenCatalogEntry() type {  
    if (Generalizations.len != 0) {  
        var entry_fields: []const UnionField = &[_]UnionField{};  
        inline for (comptime enums.values(GeneralizationId), Generalizations) |gen_id, Gen|  
{  
            entry_fields = entry_fields ++ &[_]UnionField{  
                .{  
                    .name = @tagName(gen_id),  
                    .type = Gen,  
                    .alignment = @alignOf(Gen),  
                },  
            };  
        }  
  
        return @Type(.{  
            .@("union") = .{  
                .layout = .auto,  
                .tag_type = GeneralizationId,  
                .fields = entry_fields,  
                .decls = &.{},  
            },  
        });  
    } else return void;  
}
```

```
<<generalization info declarations>>=  
fn makeGeneralizationCatalog() enums.EnumArray(GeneralizationId, GenCatalogEntry) {  
    comptime var catalog: enums.EnumArray(GeneralizationId, GenCatalogEntry) =  
.initUndefined();  
    comptime var iter = catalog.iterator();  
    inline while (comptime iter.next()) |entry| {  
        entry.value.* = @unionInit(GenCatalogEntry, @tagName(entry.key), .{});  
    }  
  
    return catalog;
```

```
}
```

```
<<generalization info declarations>>=
pub fn init() void {
    if (Generalizations.len != 0) {
        // We justify the following @constCast() on the basis that there is no
        // const iterator function for an EnumArray, and we do not modify the
        // contents of the array in this operation. The generalization_catalog is built
        // at compile time and not ever modified.
        comptime var iter = @constCast(&generalization_catalog).iterator();
        inline while (comptime iter.next()) |entry| {
            const Generalization = @FieldType(GenCatalogEntry, @tagName(entry.key));
            _ = Generalization.init(@intFromEnum(entry.key));
        }
    }
}
```

```
<<generalization info declarations>>=
pub fn GetGeneralization(
    comptime gen_id: GeneralizationId,
) type {
    const catalog_entry = generalization_catalog[@intFromEnum(gen_id)];
    return @FieldType(GenCatalogEntry, @tagName(meta.activeTag(catalog_entry)));
}
```

```
<<generalization info declarations>>=
pub fn SubclassId(
    comptime gen_id: GeneralizationId,
) type {
    return GetGeneralization(gen_id).SubclassId;
}
```

```
<<generalization info declarations>>=
pub fn DomainInstanceSet(
    comptime gen_id: GeneralizationId,
    comptime subclass_id: SubclassId(gen_id),
) type {
    return GetGeneralization(gen_id).SubclassInstanceSet(subclass_id);
}
```

```
<<generalization info declarations>>=
pub fn CodomainInstanceSet(
    comptime gen_id: GeneralizationId,
) type {
    return GetGeneralization(gen_id).SuperclassInstanceSet;
}
```

```

<<generalization info declarations>>=
pub fn DomainInstanceRef(
    comptime gen_id: GeneralizationId,
    comptime subclass_id: SubclassId(gen_id),
) type {
    return GetGeneralization(gen_id).SubclassInstanceRef(subclass_id);
}

```

```

<<generalization info declarations>>=
pub fn CodomainInstanceRef(
    comptime gen_id: GeneralizationId,
) type {
    return GetGeneralization(gen_id).SuperclassInstanceRef;
}

```

Gathering Information about Class Roles in Associations

For a domain to fulfill its requirement to ensure data integrity, we need information about the roles that each class plays in any associations where it is a participant. This design uses a strategy to attempt to minimize the amount of integrity checking that must be done on each transaction of the data model. Whenever, any classes are created or destroyed, or any associations are related or unrelated, or any generalization are related or unrelated, the association involved must be checked for consistency. Only these operations in the domain can affect consistency and rather than check every association and generalization on every domain transaction, we intend to keep track of only those classes, associations, and generalization that are modified during the transaction. In practice, most computations during domain execution do not perform operations that could affect data consistency.

The strategy is to record those associations and generalizations that could be modified while the transaction is ongoing and examine at the end only the modified associations and generalizations for consistency. To accomplish this strategy, we need to know which classes participate in which associations and generalizations and what role they play in those relationships.

For classes, that means building a mapping for each class to the role it plays, either as the domain (set) or codomain of all association.

```

<<domain declarations>>=
/// Map classes to the associations where they serve as the domain (as
/// in domain set) of the association. Answers the question: for a given
/// class, in which associations does it act in the role of a domain set.
pub const class_assoc_domain_map = MakeClassAssocDomainMap();

```

```

<<domain private functions>>=
fn MakeClassAssocDomainMap() enums.EnumArray(ClassId, enums.EnumSet(AssociationId)) {
    comptime var map : enums.EnumArray(ClassId, enums.EnumSet(AssociationId)) =
        .initFill(.initEmpty());

    inline for (comptime enums.values(ClassId)) |class_id| {
        const Class = class_info.GetClass(class_id);
        inline for (comptime enums.values(AssociationId)) |assoc_id| {
            const Assoc = association_info.GetAssociation(assoc_id);

```

```

        if (@TypeOf(Assoc.domain) == Class) {
            const domain_roles = map.getPtr(class_id);
            domain_roles.insert(assoc_id);
        }
    }
}

return map;
}

```

```

<<domain declarations>>=
/// Map classes to the associations where they serve as the codomain of
/// the association. Answers the question: for a given class, in which
/// association does it act in the role of a codomain set.
pub const class_assoc_codomain_map = MakeClassAssocCodomainMap();

```

```

<<domain private functions>>=
fn MakeClassAssocCodomainMap() enums.EnumArray(ClassId, enums.EnumSet(AssociationId)) {
    comptime var map : enums.EnumArray(ClassId, enums.EnumSet(AssociationId)) =
        .initFill(.initEmpty());

    inline for (comptime enums.values(ClassId)) |class_id| {
        const Class = class_info.GetClass(class_id);
        inline for (comptime enums.values(AssociationId)) |assoc_id| {
            const Assoc = association_info.GetAssociation(assoc_id);
            if (@TypeOf(Assoc.codomain) == Class) {
                const codomain_roles = map.getPtr(class_id);
                codomain_roles.insert(assoc_id);
            }
        }
    }
    return map;
}

```

```

<<domain declarations>>=
/// Map classes to any associations in which they participate.
/// Answers the question: for a given class, in which association does it
/// participate. This is just the union of the `class_assoc_domain_map`
/// and `class_assoc_codomain_map` entries for the given class
pub const class_assoc_map = MakeClassAssocMap();

```

```

<<domain private functions>>=
fn MakeClassAssocMap() enums.EnumArray(ClassId, enums.EnumSet(AssociationId)) {
    comptime var map : enums.EnumArray(ClassId, enums.EnumSet(AssociationId)) =
        .initFill(.initEmpty());

    inline for (comptime enums.values(ClassId)) |class_id| {
        const domains = class_assoc_domain_map.get(class_id);
        const codomains = class_assoc_codomain_map.get(class_id);
        map.set(class_id, domains.unionWith(codomains));
    }
}

```

```

    }
    return map;
}

```

Gathering Information about Class Roles in Generalizations

Similar to the mappings created for associations, we create the same mappings for the generalizations. Note that since a generalization is a disjoint union, accumulate the information from the multiple Subclass of a generalization.

```

<<domain declarations>>=
/// Map classes to the generalizations where they serve as the codomain
/// (superclass) of the generalization. Answers the question: for a given
/// class, in which generalizations does it act in the role of a superclass.
pub const class_gen_codomain_map = MakeClassGenCodomainMap();

```

```

<<domain private functions>>=
fn MakeClassGenCodomainMap() enums.EnumArray(ClassId, enums.EnumSet(GeneralizationId)) {
    comptime var map : enums.EnumArray(ClassId, enums.EnumSet(GeneralizationId)) =
        .initFill(.initEmpty());

    inline for (comptime enums.values(ClassId)) |class_id| {
        const Class = class_info.GetClass(class_id);
        inline for (comptime enums.values(GeneralizationId)) |gen_id| {
            const Generalization = generalization_info.GetGeneralization(gen_id);
            if (Generalization.Superclass == Class) {
                const domain_roles = map.getPtr(class_id);
                domain_roles.insert(gen_id);
            }
        }
    }
    return map;
}

```

```

<<domain declarations>>=
/// Map classes to the generalizations where they serve as a domain
/// (subclass) of the generalization. Answers the question: for a given
/// class, in which generalizations does it act in the role of a superclass.
pub const class_gen_domain_map = MakeClassGenDomainMap();

```

```

<<domain private functions>>=
fn MakeClassGenDomainMap() enums.EnumArray(ClassId, enums.EnumSet(GeneralizationId)) {
    comptime var map : enums.EnumArray(ClassId, enums.EnumSet(GeneralizationId)) =
        .initFill(.initEmpty());

    inline for (comptime enums.values(ClassId)) |class_id| {
        const Class = class_info.GetClass(class_id);
        inline for (comptime enums.values(GeneralizationId)) |gen_id| {
            const Generalization = generalization_info.GetGeneralization(gen_id);

```

```

    inline for (Generalization.Subclasses) |Subclass| {
        if (Subclass == Class) {
            const domain_roles = map.getPtr(class_id);
            domain_roles.insert(gen_id);
        }
    }
}
return map;
}

```

```

<<domain declarations>>=
// Map classes to any generalization in which they participate.
// Answers the question: for a given class, in which generalization does it participate.
// This is just the union of the `class_gen_domain_map` and `class_gen_codomain_map`
entries
// for the given class
pub const class_gen_map = MakeClassGenMap();

```

```

<<domain private functions>>=
fn MakeClassGenMap() enums.EnumArray(ClassId, enums.EnumSet(GeneralizationId)) {
    comptime var map : enums.EnumArray(ClassId, enums.EnumSet(GeneralizationId)) =
        .initFill(.initEmpty());

    inline for (comptime enums.values(ClassId)) |class_id| {
        const domains = class_gen_domain_map.get(class_id);
        const codomains = class_gen_codomain_map.get(class_id);
        map.set(class_id, domains.unionWith(codomains));
    }
}
return map;
}

```

Execution Sequencing

```

<<domain module declarations>>=
pub const DomainExecution = struct {
    vtable: *const VTable,
    pub const VTable = struct {
        signal: *const fn (
            src_inst: ClassInstanceRef,
            target_inst: ClassInstanceRef,
            target_event: *const anyopaque,
        ) void,
    };
    pub fn signal(
        exec: DomainExecution,
        src_inst: ClassInstanceRef,
    );
}

```

```

        target_inst: ClassInstanceRef,
        target_event: *const anyopaque,
    ) void {
        exec.vtable.signal(src_inst, target_inst, target_event);
    }
};

```

```

<<domain declarations>>=
pub const ExecutionService = struct {
    pub fn execution_service() DomainExecution {
        return .{
            .vtable = &.{.
                .signal = signal,
                .signal_delayed = signal_delayed,
                .cancel_delayed = cancel_delayed,
            },
        };
    }

    pub fn signal(
        _: ClassInstanceRef,
        target_inst: ClassInstanceRef,
        target_event: *const anyopaque,
    ) void {
        const class_index = target_inst.extract(.class_index);
        const target_class_id: ClassId = @enumFromInt(class_index);
        const domain_event = switch (target_class_id) {
            inline else => |tid| eblk: {
                const TargetEvent = class_info.Event(tid);
                const event_ptr: *const TargetEvent = @ptrCast(@alignCast(target_event));
                break :eblk makeDomainEvent(tid, event_ptr.*);
            },
        };
        domain_event_queue.insert(domain_event) catch
            @panic("domain event queue error: full");
    }

    pub fn signal_delayed(
        _: ClassInstanceRef,
        _: ClassInstanceRef,
        _: *const anyopaque,
    ) void {
        @panic("signal_delayed: not implemented");
    }

    pub fn cancel_delayed(
        _: ClassInstanceRef,
        _: ClassInstanceRef,
        _: *const anyopaque,
    ) void {
        @panic("cancel_delayed: not implemented");
    }
};

```

```
};
```

```
<<domain variables>>=
const executor = ExecutionService.execution_service();
```

```
<<class variables>>=
pub var executor: DomainExecution = undefined;
```

```
<<domain declarations>>=
pub const DomainEvent = MakeDomainEvent();
```

```
<<domain private functions>>=
// tagged union of all the event types for the classes in the domain that have
// state models.
fn MakeDomainEvent() type {
    var event_fields: []const UnionField = &[_]UnionField{};
    inline for (comptime enums.values(ClassId)) |class_id| {
        const Class = class_info.GetClass(class_id);
        if (Class.has_state_model) {
            event_fields = event_fields ++ &[_]UnionField{
                .{
                    .name = @tagName(class_id),
                    .type = Class.ClassModel.Event,
                    .alignment = @alignOf(Class.ClassModel.Event),
                },
            };
        }
    }
    return @Type(.{
        .@union" = .{
            .layout = .auto,
            .tag_type = ClassId,
            .fields = event_fields,
            .decls = &.{},
        },
    });
}
```

```
<<domain public functions>>=
pub fn makeDomainEvent(
    comptime class_id: ClassId,
    target_event: anytype, // Event value for a specific class
) DomainEvent {
    return @unionInit(DomainEvent, @tagName(class_id), target_event);
}
```

```

<<domain variables>>=
const domain_event_queue = struct {
    // Make a type to hold the requested count that is the next power of two up from
    // the count value. N.B. this scheme uses wrapping arithmetic and consumes one
    // slot to detect overflow.
    const StorageIndex = // type of the nearest power of two that hold the requested count
        if (opt_event_queue_count) |event_queue_count|
            meta.Int(.unsigned, math.log2_int_ceil(u8, event_queue_count + 1)) // +1 for
wasted slot
        else
            u2; // default is 2 bits == 4 queue elements, max capacity = 3

    var queue_storage: [math.maxInt(StorageIndex)]DomainEvent = undefined;
    var head: StorageIndex = 0;
    var tail: StorageIndex = 0;

    pub const QueueError = error {
        empty,
        full,
    };

    pub fn insert(
        event: DomainEvent,
    ) QueueError!void {
        const next_tail = tail +% 1;
        if (next_tail == head) return QueueError.full;
        queue_storage[tail] = event;
        tail = next_tail;
    }

    pub fn remove() QueueError!DomainEvent {
        if (is_empty()) return QueueError.empty;
        const event = queue_storage[head];
        head +=% 1;
        return event;
    }

    pub fn is_empty() bool {
        return head == tail;
    }

    pub fn is_not_empty() bool {
        return !is_empty();
    }
};

```

```

<<domain variables>>=
var in_thread_of_control: bool = false;

```

```

<<domain public functions>>=
pub fn runThreadOfControl(

```

```

event: DomainEvent,
) void {
    assert(!in_thread_of_control);

    in_thread_of_control = true;
    defer in_thread_of_control = false;

    dispatchDomainEvent(event);
    while(domain_event_queue.remove() |queued_event| {
        dispatchDomainEvent(queued_event);
    } else |_| {}
}

```

```

<<domain private functions>>=
fn dispatchDomainEvent(
    event: DomainEvent,
) void {
    switch (event) {
        inline else => |class_event, class_id| {
            const TargetClass = class_info.GetClass(class_id);
            const target_model: TargetClass.ClassModel = .{};
            const terminal = target_model.receiveEvent(class_event);
            if (terminal) {
                switch (class_event) {
                    inline else => |event_tuple| {
                        const target_inst_ref = event_tuple.@"0";
                        // HERE -- unrelated the instance from all relationships
                        target_inst_ref.destroy();
                    },
                }
            },
        },
    }
}

```

Tracing Domain Execution

Each domain provides an interface to enable the relational classes contained in the domain to report significant *model-level operations* that occur during execution. This tracing facility has many uses. For testing purposes, execution tracing allows obtaining a chronological view of significant logical happenings in terms of model concepts. The importance of tracing cannot be overstated. In an event driven execution model that uses state machines, traditional breakpoint debugging is not easily accomplished. Events drive the thread of execution control across a directed graph and, *a priori*, it is difficult to know where a breakpoint needs to be set.

Naturally, tracing a domain's own execution incurs a computational cost. We would like a design that minimizes that cost. Anticipated usage during system development is that tracing is an integral part of the building, testing, and debugging effort. Anticipated usage during system deployment is that tracing is used selectively and most classes of a domain will not be traced at all. However, the particular tracing needed when a domain is reused is generally not possible to predict.

This design allows trace detection and any tracing consequences to be specified at run time. There are a lot of moving parts. The following figure shows a block diagram of the class tracing design.

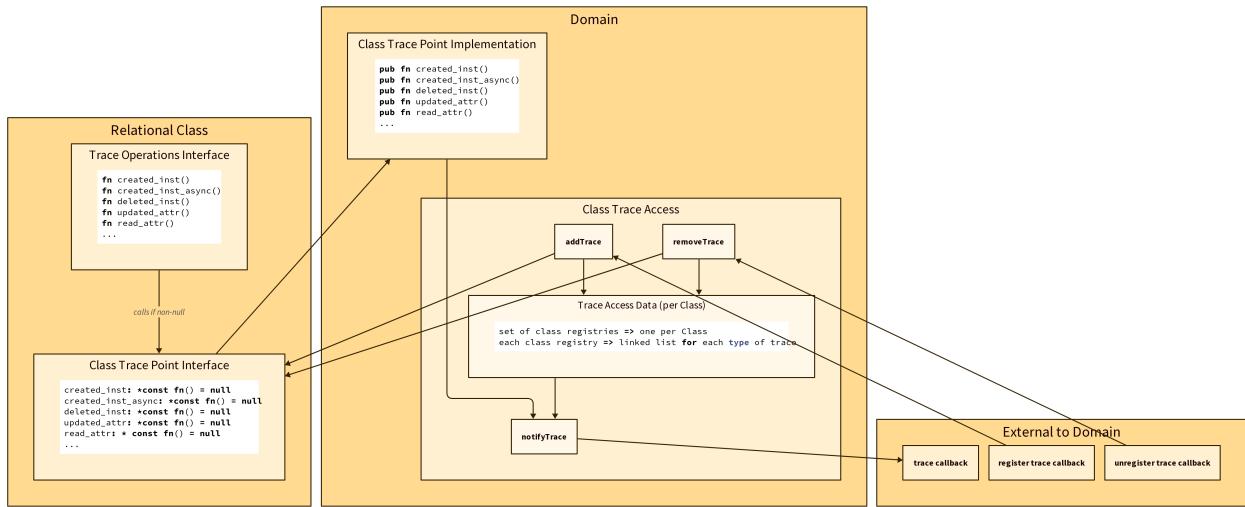


Figure 58. Block Diagram of Class Tracing Design

In the previous figure starting from the left with the Class block, every Relational Class has a Trace Operations Interface that is used by the class operations implementation to report the specific model-level operations that have been executed. For example, when a class instance is created, the code in the class that creates the instance invokes the `created_inst` function in the Trace Operations Interface to indicate that an instance creation happened. The interface is structured as a simple namespace `struct` containing one function for each possible traced operation. The Trace Operations Interface functions perform two essential operations.

1. Invoke the corresponding Class Trace Point Interface function (more on this follows) if it is non-null. The Class Trace Point Interface uses function pointers to direct the trace notification to the containing Domain. Optional function pointers are used to reduce the computational burden on those class operations that are not traced to a simple test for a `null` function pointer.
2. Convert any `class InstanceRef` argument of a Trace Point Interface function to a `ClassInstanceRef`. When the reference to a class instance leaves the boundary of the class, a class number is added to the reference. This allows the domain to determine from which class the trace came so it can handle traces from any class. This pattern of augmenting the identifying information is used in other places when an instance reference is transferred to a scope where it is no longer a unique identifier.

The Class Trace Point Interface is a namespace `struct` of optional function pointers. All the fields are initialized to `null` and as tracing is requested, the `null` function pointer values are replaced. The function pointers in the Class Trace Point Interface are resolved to functions of the Class Trace Point Implementation in the containing Domain.

Moving right to the Domain block, the function pointers in the Class Trace Point Interface are updated to point to the functions contained in the Class Trace Point Implementation. There are two essential operations performed by the interface implementation functions.

1. Convert any `ClassInstanceRef` values to `DomainInstanceRef` values by adding a domain number to the reference. Since these references are transferred out of the domain, the domain numbering allows the system level code to determine from which domain the trace came.

2. Invoke the `notifyTrace` function to forward the trace information to any registered observers.

Each Domain presents a Class Trace Access Interface that allows entities external to the Domain to register and unregister for traces on specific classes and trace types. There are two primary trace access data structures:

1. An `EnumArray` indexed by `ClassId` where each element holds a registry for the particular class.
2. The class registry which is a `struct` with a field for each type of trace. The registry fields contain a linked list where trace observers are recorded.

The `notifyTrace` function traverses the list associated with a particular class and type of trace sequentially invoking any callback functions that are found.

When the first trace of a given type is registered on a class, the `null` pointer in the Class Trace Point Interface is set to the appropriate function from the Class Trace Point Implementation. The `addTrace` function detects this condition. Note that the `addTrace` function adds observers to the registry list by *prepend*ing the observer. This means that `notifyTrace` executes trace callbacks in reverse order in which they were added.

When the last trace is unregistered on a class for a particular trace type, the pointer in the Class Trace Point Interface is set back to `null`. The `removeTrace` detects this condition.

Finally, entities external to the Domain may invoke `addTrace` and `removeTrace` to register for the tracing notifications. The execution control and trace data is supplied to a trace callback function. We do not specify any particulars of the external entities that may use the tracing facilities. However, the Domain module provides a trace logging feature that can trace the class operations for any or all of the classes in a given domain. The trace logging outputs informative messages to the scoped logging namespace of the Domain.

Tracing Class Actions

We start by enumerating all the actions of a class that can be traced.

```
<<domain module declarations>>=
pub const ClassTraceId = enum {
    created_inst,
    deleted_inst,
    updated_attr,
    read_attr,
    created_inst_async,
    transitioned,
    undefined_transition,
    exited_state,
    continue_event,
    signaled_event,
    signaled_delayed_event,
    canceled_delayed_event,
};
```

Note that the enumeration names are mainly in the past tense. This is because the trace is notified only after the class operation has happened.

Relational Class Trace Operations Interface

The following functions are used by class code to indicate to the domain that the execution corresponding to trace has been executed. The implementation of the functions in the `class_trace_point` forwards the tracing information to the containing domain. Integral to the design, is that the class converts its internal `InstanceRef` into a `ClassInstanceRef`.

Note we use literate program chunks to insert this declaration into the class module.

```
<<class declarations>>=
/// The `trace_operations` structure is a namespace structure containing
/// the functions used by Relational Class operations to indicate
/// trace conditions. There is one function for each class trace type.
/// The functions in the namespace follow the same pattern. The optional
/// function pointer is unwrapped and any `InstanceRef` arguments are
/// converted into `ClassInstanceRef` values. All `ClassInstanceRef`
/// values along with the other arguments are passed as arguments to
/// the invocation of the unwrapped function pointer.
pub const trace_operations = struct {
    fn created_inst(
        inst_ref: InstanceRef,
    ) void {
        if (trace_point.created_inst) |created_inst_trace|
            created_inst_trace(inst_ref.classInstanceRef());
    }
    fn deleted_inst(
        inst_ref: InstanceRef,
    ) void {
        if (trace_point.deleted_inst) |deleted_inst_trace|
            deleted_inst_trace(inst_ref.classInstanceRef());
    }
    fn updated_attr(
        inst_ref: InstanceRef,
        attr_index: u8,
        value: *const anyopaque,
    ) void {
        if (trace_point.updated_attr) |updated_attr_trace|
            updated_attr_trace(
                inst_ref.classInstanceRef(),
                attr_index,
                value,
            );
    }
    fn read_attr(
        inst_ref: InstanceRef,
        attr_index: u8,
        value: *const anyopaque,
    ) void {
        if (trace_point.read_attr) |read_attr_trace|
            read_attr_trace(
                inst_ref.classInstanceRef(),
                attr_index,
            );
    }
}
```

```

        value,
    );
}

pub fn created_inst_async(
    inst_ref: InstanceRef,
) void {
    if (trace_point.created_inst_async) |created_inst_async_trace|
        created_inst_async_trace(inst_ref.classInstanceRef());
}

pub fn transitioned(
    inst_ref: InstanceRef,
    source_state: u8,
    event: u8,
    target_state: i8,
) void {
    if (trace_point.transitioned) |transitioned_trace|
        transitioned_trace(
            inst_ref.classInstanceRef(),
            source_state,
            event,
            target_state,
        );
}

pub fn undefined_transition(
    inst_ref: InstanceRef,
    source_state: u8,
    event: u8,
) void {
    if (trace_point.undefined_transition) |undefined_transition_trace|
        undefined_transition_trace(
            inst_ref.classInstanceRef(),
            event,
            source_state,
        );
}

pub fn exited_state(
    inst_ref: InstanceRef,
    state: u8,
) void {
    if (trace_point.exited_state) |exited_state_trace|
        exited_state_trace(
            inst_ref.classInstanceRef(),
            state,
        );
}

pub fn continue_event(
    inst_ref: InstanceRef,
    event: u8,
) void {
    if (trace_point.continue_event) |continued_state_trace|
        continued_state_trace(
            inst_ref.classInstanceRef(),

```

```

        event,
    );
}

pub fn signaled_event(
    source_inst: InstanceRef,
    target_inst: anytype,
    event: u8,
) void {
    if (trace_point.signaled_event) |signaled_event_trace|
        signaled_event_trace(
            source_inst.classInstanceRef(),
            target_inst.classInstanceRef(),
            event,
        );
}

pub fn signaled_delayed_event(
    source_inst: InstanceRef,
    target_inst: anytype,
    event: u8,
    delay: u32,
) void {
    if (trace_point.signaled_delayed_event) |signaled_delayed_event_trace|
        signaled_delayed_event_trace(
            source_inst.classInstanceRef(),
            target_inst.classInstanceRef(),
            event,
            delay,
        );
}

pub fn canceled_delayed_event(
    source_inst: InstanceRef,
    target_inst: anytype,
    event: u8,
) void {
    if (trace_point.canceled_delayed_event) |canceled_delayed_event_trace|
        canceled_delayed_event_trace(
            source_inst.classInstanceRef(),
            target_inst.classInstanceRef(),
            event,
        );
}
};


```

Class Trace Point Interface

A Domain declares a namespace `struct` to serve as the interface for handling the detection of trace points. The structure is simply a jump table of function pointers. The referenced functions are of the type which corresponds to those of the Trace Operations Interface, with the substitution of an `InstanceRef` value with a `ClassInstanceRef` value.

```
<<domain module declarations>>=
```

```

/// The `class_trace_point` structure is a namespace containing optional
/// pointers to functions that indicate trace actions to a containing
/// domain. The pointers are default initialized to `null` and are
/// replaced with a non-null value when the containing domain determines
/// that an observer has registered a trace for the given class.
pub const class_trace_point = struct {
    created_inst: ?*const fn (inst: ClassInstanceRef) void = null,
    deleted_inst: ?*const fn (inst: ClassInstanceRef) void = null,
    updated_attr: ?*const fn (
        inst: ClassInstanceRef,
        attr_index: u8,
        value: *const anyopaque,
    ) void = null,
    read_attr: ?*const fn (
        inst: ClassInstanceRef,
        attr_index: u8,
        value: *const anyopaque,
    ) void = null,
    created_inst_async: ?*const fn (inst: ClassInstanceRef) void = null,
    transitioned: ?*const fn (
        inst: ClassInstanceRef,
        source_state: u8,
        event: u8,
        target_state: i8,
    ) void = null,
    undefined_transition: ?*const fn (
        inst: ClassInstanceRef,
        source_state: u8,
        event: u8,
    ) void = null,
    exited_state: ?*const fn (inst: ClassInstanceRef, state: u8) void = null,
    continue_event: ?*const fn (inst: ClassInstanceRef, event: u8) void = null,
    signaled_event: ?*const fn (
        source_inst: ClassInstanceRef,
        target_inst: ClassInstanceRef,
        event: u8,
    ) void = null,
    signaled_delayed_event: ?*const fn (
        source_inst: ClassInstanceRef,
        target_inst: ClassInstanceRef,
        event: u8,
        delay: u32,
    ) void = null,
    canceled_delayed_event: ?*const fn (
        source_inst: ClassInstanceRef,
        target_inst: ClassInstanceRef,
        event: u8,
    ) void = null,
};


```

Using literate programming chunks, we insert a variable into the RelClass to hold the Trace Point Interface.

```
<<class variables>>=
pub var trace_point: class_trace_point = .{};
```

Class Trace Point Implementation

The counterpart of the Class Trace Point Interface is the Class Trace Point Implementation. The implementation is provided by a namespace `struct` that contains a set of function whose signature corresponds to those of the Class Trace Point Interface.

```
<<domain private functions>>=
/// The `domainInstanceRefFromClassInstanceRef` function creates a
/// `DomainInstanceRef` value from a `ClassInstanceRef` value. These
/// values are suitable for uniquely identifying a class instance in
/// the context of the entire system.
fn domainInstanceRefFromClassInstanceRef(
    inst: ClassInstanceRef,
) DomainInstanceRef {
    return DomainInstanceRef.create(.{
        .domain_index = domain_index,
        .class_index = inst.extract(.class_index),
        .tuple_id = inst.extract(.tuple_id),
        .gen_id = inst.extract(.gen_id),
    });
}
```

```
<<domain declarations>>=
/// The `class_trace_point_impl` structure is a namespace containing
/// functions that are invoked from the Class Trace Point Interface
/// to indicate a class condition that may be traced. These functions
/// follow a pattern. The identifier of the `RelClass` in the containing
/// Domain is obtained from a `ClassInstanceRef` value. There is always
/// at least one `ClassInstanceRef` typed argument since we are tracing
/// `RelClass` actions. Any `ClassInstanceRef` values are converted
/// to `DomainInstanceRef` values by adding the `domain_index`. The
/// currently registered observers of the trace type for the given class
/// are notified. The arguments corresponding to the registered trace
/// callback are packaged as a tuple.
pub const class_trace_point_impl = struct {
    pub fn created_inst(
        inst: ClassInstanceRef,
    ) void {
        const class_id: ClassId = @enumFromInt(inst.extract(.class_index));
        const domain_inst = domainInstanceRefFromClassInstanceRef(inst);
        class_trace_access.created_inst.notifyTrace(class_id, .{domain_inst});
    }
    pub fn deleted_inst(
        inst: ClassInstanceRef,
    ) void {
        const class_id: ClassId = @enumFromInt(inst.extract(.class_index));
        const domain_inst = domainInstanceRefFromClassInstanceRef(inst);
```

```

        class_trace_access.deleted_inst.notifyTrace(class_id, .{domain_inst});
    }

    pub fn updated_attr(
        inst: ClassInstanceRef,
        attr_index: u8,
        value: *const anyopaque,
    ) void {
        const class_id: ClassId = @enumFromInt(inst.extract(.class_index));
        const domain_inst = domainInstanceRefFromClassInstanceRef(inst);
        class_trace_access.updated_attr.notifyTrace(class_id, .{ domain_inst, attr_index,
value });
    }

    pub fn read_attr(
        inst: ClassInstanceRef,
        attr_index: u8,
        value: *const anyopaque,
    ) void {
        const class_id: ClassId = @enumFromInt(inst.extract(.class_index));
        const domain_inst = domainInstanceRefFromClassInstanceRef(inst);
        class_trace_access.read_attr.notifyTrace(class_id, .{ domain_inst, attr_index,
value });
    }

    pub fn created_inst_async(
        inst: ClassInstanceRef,
    ) void {
        const class_id: ClassId = @enumFromInt(inst.extract(.class_index));
        const domain_inst = domainInstanceRefFromClassInstanceRef(inst);
        class_trace_access.created_inst_async.notifyTrace(class_id, .{domain_inst});
    }

    pub fn transitioned(
        inst: ClassInstanceRef,
        source_state: u8,
        event: u8,
        target_state: i8,
    ) void {
        const class_id: ClassId = @enumFromInt(inst.extract(.class_index));
        const domain_inst = domainInstanceRefFromClassInstanceRef(inst);
        class_trace_access.transitioned.notifyTrace(class_id, .{
            domain_inst,
            source_state,
            event,
            target_state,
        });
    }

    pub fn undefined_transition(
        inst: ClassInstanceRef,
        source_state: u8,
        event: u8,
    ) void {
        const class_id: ClassId = @enumFromInt(inst.extract(.class_index));
        const domain_inst = domainInstanceRefFromClassInstanceRef(inst);
        class_trace_access.undefined_transition.notifyTrace(class_id, .{
    }

```

```

        domain_inst,
        source_state,
        event,
    });
}

pub fn exited_state(
    inst: ClassInstanceRef,
    state: u8,
) void {
    const class_id: ClassId = @enumFromInt(inst.extract(.class_index));
    const domain_inst = domainInstanceRefFromClassInstanceRef(inst);
    class_trace_access.exited_state.notifyTrace(class_id, .{ domain_inst, state });
}

pub fn continue_event(
    inst: ClassInstanceRef,
    event: u8,
) void {
    const class_id: ClassId = @enumFromInt(inst.extract(.class_index));
    const domain_inst = domainInstanceRefFromClassInstanceRef(inst);
    class_trace_access.continue_event.notifyTrace(class_id, .{ domain_inst, event });
}

pub fn signaled_event(
    source_inst: ClassInstanceRef,
    target_inst: ClassInstanceRef,
    event: u8,
) void {
    const class_id: ClassId = @enumFromInt(source_inst.extract(.class_index));
    const domain_source_inst = domainInstanceRefFromClassInstanceRef(source_inst);
    const domain_target_inst = domainInstanceRefFromClassInstanceRef(target_inst);
    class_trace_access.signaled_event.notifyTrace(class_id, .{
        domain_source_inst,
        domain_target_inst,
        event,
    });
}

pub fn signaled_delayed_event(
    source_inst: ClassInstanceRef,
    target_inst: ClassInstanceRef,
    event: u8,
    delay: u32,
) void {
    const class_id: ClassId = @enumFromInt(source_inst.extract(.class_index));
    const domain_source_inst = domainInstanceRefFromClassInstanceRef(source_inst);
    const domain_target_inst = domainInstanceRefFromClassInstanceRef(target_inst);
    class_trace_access.signaled_delayed_event.notifyTrace(class_id, .{
        domain_source_inst,
        domain_target_inst,
        event,
        delay,
    });
}

```

```

    }

    pub fn canceled_delayed_event(
        source_inst: ClassInstanceRef,
        target_inst: ClassInstanceRef,
        event: u8,
    ) void {
        const class_id: ClassId = @enumFromInt(source_inst.extract(.class_index));
        const domain_source_inst = domainInstanceRefFromClassInstanceRef(source_inst);
        const domain_target_inst = domainInstanceRefFromClassInstanceRef(target_inst);
        class_trace_access.canceled_delayed_event.notifyTrace(class_id, .{
            domain_source_inst,
            domain_target_inst,
            event,
        });
    }
};

}

```

Defining Class Callbacks

A Domain has data structures and an interface to support entities from outside of the domain to have access to the execution trace information. For any entity outside of a Domain, tracing information arrives in the form of a callback function whose arguments carry the trace details. In this section, we define what those callback function interfaces.

```

<<domain module declarations>>=
/// A `DomainInstanceRef` is a `RelTuple` that contains a class instance
/// identifier that is unique across the entire system. It contains information
/// to identify not only an instance, but also to which class the instance
/// belongs and to which domain the class belongs.
pub const DomainInstanceRef = RelTuple(struct {
    domain_index: u8,
    class_index: u8,
    tuple_id: RelationCapacity,
    gen_id: GenerationNumber,
});

```

The following declaration defines the callback function types that must be supplied to a domain to obtain tracing information. Note that each type of trace has a distinct trace callback function type.

N.B. that each trace function callback returns a `bool` value. If the return value is `true`, then callback notifications further on in the registry list are invoked. If the return value is `false`, then callback notifications are terminated and any remaining callbacks for the trace type are *not* invoked. This gives some measure of control to the callback function over the continuation processing without having to handle error unions. An error union return in this context would cause widespread consideration of how to handle the errors. For tracing, any errors encountered are not allowed to control continued execution of the domain.

```

<<domain module declarations>>=
pub const CreatedInstTraceFn = *const fn (inst: DomainInstanceRef) bool;
pub const DeletedInstTraceFn = *const fn (inst: DomainInstanceRef) bool;

```

```

pub const UpdatedAttrTraceFn = *const fn (
    inst: DomainInstanceRef,
    attr_index: u8,
    value: *const anyopaque,
) bool;
pub const ReadAttrTraceFn = *const fn (
    inst: DomainInstanceRef,
    attr_index: u8,
    value: *const anyopaque,
) bool;
pub const CreatedInstAsyncTraceFn = *const fn (inst: DomainInstanceRef) bool;
pub const TransitionedTraceFn = *const fn (
    inst: DomainInstanceRef,
    source_state: u8,
    event: u8,
    target_state: i8,
) bool;
pub const UndefinedTransitionTraceFn = *const fn (
    inst: DomainInstanceRef,
    source_state: u8,
    event: u8,
) bool;
pub const ExitedStateTraceFn = *const fn (inst: DomainInstanceRef, state: u8) bool;
pub const ContinueEventTraceFn = *const fn (inst: DomainInstanceRef, event: u8) bool;
pub const SignaledEventTraceFn = *const fn (
    source_inst: DomainInstanceRef,
    target_inst: DomainInstanceRef,
    event: u8,
) bool;
pub const SignaledDelayedEventTraceFn = *const fn (
    source_inst: DomainInstanceRef,
    target_inst: DomainInstanceRef,
    event: u8,
    delay: u32,
) bool;
pub const CanceledDelayedEventTraceFn = *const fn (
    source_inst: DomainInstanceRef,
    target_inst: DomainInstanceRef,
    event: u8,
) bool;

```

Class Trace Registry

For each RelClass, the Domain maintains a registry for the observers of the actions of the class.

```

<<domain module declarations>>=
/// The `ClassTraceRegistry` structure contains one field for each type of
/// trace. The registry for a given trace type consists of a linked list
/// along with operations on the list.
pub const ClassTraceRegistry = struct {
    const TraceList = std.DoublyLinkedList;

```

```

created_inst: CreatedInst = .{},
deleted_inst: DeletedInst = .{},
updated_attr: UpdatedAttr = .{},
read_attr: ReadAttr = .{},
created_inst_async: CreatedInstAsync = .{},
transitioned: Transitioned = .{},
undefined_transition: UndefinedTransition = .{},
exited_state: ExitedState = .{},
continue_event: ContinueEvent = .{},
signaled_event: SighaledEvent = .{},
signaled_delayed_event: SignaledDelayedEvent = .{},
canceled_delayed_event: CanceledDelayedEvent = .{},

<<class registry declarations>>
};

```

For each type of class trace, there are type-specific declarations and functions to manage the trace registration. We use a `comptime` type function to tailor the trace type registries to the callback functions of each trace type.

```

<<class registry declarations>>=
/// The `MakeRegistry` returns a data type suitable for use in defining
/// a registry of observers for the trace callback function given by the
/// `TraceFn` argument.
fn MakeRegistry(
    comptime TraceFn: type,
) type {
    // Most of these trait checks are never triggered since the callback function
    // types are known and specified correctly. We leave them in place to
    // support future additions to the types of traces supported.
    const trace_fn_info = @typeInfo(TraceFn);
    if (trace_fn_info != .pointer)
        @compileError("trace function argument must be a pointer");

    const ptr = &trace_fn_info.pointer;
    if (!ptr.is_const)
        @compileError("trace function argument must be a const pointer");
    if (ptr.size != .one)
        @compileError("trace function argument must be a singular pointer");

    const child = @typeInfo(ptr.child);
    if (child != .@"fn")
        @compileError("trace function argument must point to a function");
    if (child.@"fn".is_generic)
        @compileError("trace functions may not generic");
    if (child.@"fn".is_var_args)
        @compileError("trace functions may not be variable argument functions");
    if (child.@"fn".return_type) |ret| {
        if (@typeInfo(ret) != .bool)
            @compileError(
                "trace functions must return bool, got: '" ++

```

```

        @typeName(ret) ++
        "()");
    } else {
        @compileError("trace functions must have a void return type");
    }

    return struct {
        traces: TraceList = .{},
        const Self = @This();
        <<make registry declarations>>
    };
}

```

The core of the registry is a doubly linked list. Since the lists from `std.DoublyLinkedList` are intrusive, it is necessary to declare a type for the list entries and that type must include a linked list `Node`. The functions provided are simple operations to add, remove, and walk the list. Two items to note:

1. Both `addEntry` and `removeEntry` detect the state change of empty to non-empty and non-empty to empty, respectively. This state change is used by other code to determine when the function pointers of the Class Trace Point Interface require updating.
2. The `notifyEntries` function implements the processing to terminate the walk of the list should one of the callback functions return a `false` value.

```

<<make registry declarations>>=
/// _N.B._ that the memory pointed to by the `entry` argument to
/// `addEntry` and `removeEntry` is _not_ managed by the registry code.
/// The registry code does _not_ take ownership of the memory. Callers
/// of `addEntry` and `removeEntry` must ensure that the lifetime of
/// a registry `Entry` value spans the time from when `addEntry` is invoked
/// until `removeEntry` is invoked. This is a long winded way to say that
/// passing pointers to local variables allocated on the stack is almost
/// always the wrong approach to allocating memory for the trace entries.
/// See the `TraceBinding` type function for one way to handle
/// memory allocation for trace entries.

pub const Entry = struct {
    node: TraceList.Node = .{},
    trace_fn: TraceFn,
};

pub fn addEntry(
    self: *Self,
    entry: *Entry,
) bool {
    const was_empty = self.traces.first == null;
    self.traces.prepend(&entry.node);
    return was_empty;
}

pub fn removeEntry(

```

```

    self: *Self,
    entry: *Entry,
) bool {
    self.traces.remove(&entry.node);
    entry.node = .{ }; // make sure things work if the entry is added later.
    return self.traces.first == null;
}

pub fn notifyEntries(
    self: *Self,
    args: meta.ArgsTuple(@typeInfo(TraceFn).pointer.child),
) void {
    var iter = self.traces.first;
    while (iter) |element| : (iter = element.next) {
        const entry: *Entry = @alignCast(@fieldParentPtr("node", element));
        const success = @call(.auto, entry.trace_fn, args);
        if (!success) break;
    }
}

```

To finish off the `ClassTraceRegistry` declarations, we generate the data types for each of the trace type registries.

```

<<class registry declarations>>=
pub const CreatedInst = MakeRegistry(CreatedInstTraceFn);
pub const DeletedInst = MakeRegistry(DeletedInstTraceFn);
pub const UpdatedAttr = MakeRegistry(UpdatedAttrTraceFn);
pub const ReadAttr = MakeRegistry(ReadAttrTraceFn);
pub const CreatedInstAsync = MakeRegistry(CreatedInstAsyncTraceFn);
pub const Transitioned = MakeRegistry(TransitionedTraceFn);
pub const UndefinedTransition = MakeRegistry(UndefinedTransitionTraceFn);
pub const ExitedState = MakeRegistry(ExitedStateTraceFn);
pub const ContinueEvent = MakeRegistry(ContinueEventTraceFn);
pub const SignaledEvent = MakeRegistry(SignaledEventTraceFn);
pub const SignaledDelayedEvent = MakeRegistry(SignaledDelayedEventTraceFn);
pub const CanceledDelayedEvent = MakeRegistry(CanceledDelayedEventTraceFn);

```

Access to Class Traces

Each domain make available an interface allowing access to adding or removing trace observers. The design is similar to the `class_trace_point_impl` namespace in that there is a separate declaration for each trace type. Note this design very much depends on the naming convention that uses the same trace type name in every structure that has fields or declarations that are dependent on the type of the class trace. This makes managing the mapping between the related fields and declarations easier since the name mapping is essentially the identity mapping.

```

<<domain declarations>>=
/// The `class_trace_access` namespace serves as a container for the
/// `ClassTraceRegistry` values for each `RelClass` in the `Domain`.
pub const class_trace_access = struct {
    // Array of trace registries indexed by ClassId.

```

```

// When a trace is notified, this list is searched for all observers
// watching the notified trace id.
var registry: enums.EnumArray(ClassId, ClassTraceRegistry) = .initFill(.{});

<<trace access declarations>>
};

```

```

<<trace access declarations>>=
/// The `updateTracePointInterface` function modifies the function
/// pointer in the class trace point interface for the class given by,
/// `class_id`. The `trace_field_name` argument gives the field name in
/// the trace point interface that is to be modified. The modification
/// is controlled by the `op` argument. If `op` is given as `.link`,
/// then the trace point interface function pointer is updated to point
/// to the appropriate function in the trace point implementation.
/// If `op` is given as `.unlink`, then `null` is stored in the trace
/// pointer interface function pointer field.
fn updateTracePointInterface(
    op: enum {link, unlink},
    class_id: ClassId,
    comptime trace_field_name: []const u8,
) void {
    const catalog_entry = class_info.getCatalogEntry(class_id);
    switch (catalog_entry) {
        inline else => |class| {
            const Class = @TypeOf(class);
            @field(Class.trace_point, trace_field_name) = switch (op) {
                .link => @field(class_trace_point_impl, trace_field_name),
                .unlink => null,
            };
        },
    }
}

```

Since there is substantial repetition in the trace access code, we use a type function to generate the required code, parameterized appropriately by the type of trace.

```

<<trace access declarations>>=
/// The `MakeTraceAccess` type function returns a data type suitable for
/// declaring storage for access to the trace information for a particular
/// trace type. The `Registry` argument gives the type of an observer
/// registry for particular type of trace. The `trace_field_name`'
/// is a string that gives the field name in the `ClassTraceRegistry`'
/// of the domain whose data type is given by `Registry`.
fn MakeTraceAccess(
    comptime Registry: type,
    comptime trace_field_name: []const u8,
) type {
    return struct {
        pub fn addTrace(
            class_id: ClassId,

```

```

        entry: *Registry.Entry,
    ) void {
        const class_registry = registry.getPtr(class_id);
        const trace_registry = &@field(class_registry, trace_field_name);
        const was_empty = trace_registry.addEntry(entry);
        if (was_empty) updateTracePointInterface(.link, class_id, trace_field_name);
    }

    pub fn removeTrace(
        class_id: ClassId,
        entry: *Registry.Entry,
    ) void {
        const class_registry = registry.getPtr(class_id);
        const trace_registry = &@field(class_registry, trace_field_name);
        const is_empty = trace_registry.removeEntry(entry);
        if (is_empty) updateTracePointInterface(.unlink, class_id, trace_field_name);
    }

    fn notifyTrace(
        class_id: ClassId,
        args: TraceArgs(Registry.Entry),
    ) void {
        const class_registry = registry.getPtr(class_id);
        const trace_registry = &@field(class_registry, trace_field_name);
        trace_registry.notifyEntries(args);
    }

    fn TraceArgs(
        comptime Entry: type,
    ) type {
        return comptime meta.ArgsTuple(
            @typeInfo(@FieldType(Entry, "trace_fn")).pointer.child,
        );
    }
};

}

```

Finally, we can make the required declarations to handle access to each type of trace.

```

<<trace access declarations>>=
pub const created_inst = MakeTraceAccess(
    ClassTraceRegistry.CreatedInst,
    "created_inst",
);
pub const deleted_inst = MakeTraceAccess(
    ClassTraceRegistry.DeletedInst,
    "deleted_inst",
);
pub const updated_attr = MakeTraceAccess(
    ClassTraceRegistry.UpdatedAttr,
    "updated_attr",
);

```

```

pub const read_attr = MakeTraceAccess(
    ClassTraceRegistry.ReadAttr,
    "read_attr",
);
pub const created_inst_async = MakeTraceAccess(
    ClassTraceRegistry.CreatedInstAsync,
    "created_inst_async",
);
pub const transitioned = MakeTraceAccess(
    ClassTraceRegistry.Transitioned,
    "transitioned",
);
pub const undefined_transition = MakeTraceAccess(
    ClassTraceRegistry.UndefinedTransition,
    "undefined_transition",
);
pub const exited_state = MakeTraceAccess(
    ClassTraceRegistry.ExitedState,
    "exited_state",
);
pub const continue_event = MakeTraceAccess(
    ClassTraceRegistry.ContinueEvent,
    "continue_event",
);
pub const signaled_event = MakeTraceAccess(
    ClassTraceRegistry.SignaledEvent,
    "signaled_event",
);
pub const signaled_delayed_event = MakeTraceAccess(
    ClassTraceRegistry.SignaledDelayedEvent,
    "signaled_delayed_event",
);
pub const canceled_delayed_event = MakeTraceAccess(
    ClassTraceRegistry.CanceledDelayedEvent,
    "canceled_delayed_event",
);

```

Domain Logging Service

As a convenience, the Domain module provides type functions that can be used to provide log output for all classes in a domain and for each type of trace.

The first task for logging a domain is to define callback functions for all the different types of traces. Note that in this case the code directly logs to the domain's scoped log function. Other types of output are possible by creating a different trace function type function where the callbacks write the trace information to another place. However, remember that the callbacks execute synchronously to the traced action and blocking calls in the trace callback blocks the execution flow in the domain.

```

<<domain module declarations>>=
/// The `TraceLogging` type function returns a namespace structure
/// that contains a trace callback for each type of class trace. The
/// `TracedDomain` argument is the data type of the domain as returned

```

```

/// from the `Domain` type function. The generated callback functions
/// will reference class properties from those Relation Classes contained
/// in `TracedDomain`.
pub fn TraceLogging(
    comptime TracedDomain: type,
) type {
    return struct {
        fn created_inst(
            inst: DomainInstanceRef,
        ) bool {
            TracedDomain.log.info("created: {s}.{d}:{d}", .{
                @tagName(@as(TracedDomain.ClassId, @enumFromInt(inst.extract(.class_index)))),
                inst.extract(.tuple_id),
                inst.extract(.gen_id),
            });
            return true;
        }
        fn created_inst_async(
            inst: DomainInstanceRef,
        ) bool {
            TracedDomain.log.info("created async: {s}.{d}:{d}", .{
                @tagName(@as(TracedDomain.ClassId, @enumFromInt(inst.extract(.class_index)))),
                inst.extract(.tuple_id),
                inst.extract(.gen_id),
            });
            return true;
        }
        fn deleted_inst(
            inst: DomainInstanceRef,
        ) bool {
            TracedDomain.log.info("deleted: {s}.{d}:{d}", .{
                @tagName(@as(TracedDomain.ClassId, @enumFromInt(inst.extract(.class_index)))),
                inst.extract(.tuple_id),
                inst.extract(.gen_id),
            });
            return true;
        }
        fn updated_attr(
            inst: DomainInstanceRef,
            attr_index: u8,
            value: *const anyopaque,
        ) bool {
            const id: TracedDomain.ClassId = @enumFromInt(inst.extract(.class_index));
            const entry = TracedDomain.class_info.getCatalogEntry(id);
            switch (entry) {
                inline else => |class, class_id| {
                    const Class = @TypeOf(class);
                    const attr_id: Class.AttributeId = @enumFromInt(attr_index);
                    switch (attr_id) {

```

```

        inline else => |aid| {
            const attr_value: *const Class.AttributeType(aid) =
                @ptrCast(@alignCast(value));
            TracedDomain.log.info("updated attribute: {s}.{d}:{d}.{s} <-
{any}", .{
                @tagName(class_id),
                inst.extract(.tuple_id),
                inst.extract(.gen_id),
                @tagName(aid),
                attr_value.*,
            });
        },
    },
},
return true;
}
fn read_attr(
    inst: DomainInstanceRef,
    attr_index: u8,
    value: *const anyopaque,
) bool {
    const id: TracedDomain.ClassId = @enumFromInt(inst.extract(.class_index));
    const entry = TracedDomain.class_info.getCatalogEntry(id);
    switch (entry) {
        inline else => |class, class_id| {
            const Class = @TypeOf(class);
            const attr_id: Class.AttributeId = @enumFromInt(attr_index);
            switch (attr_id) {
                inline else => |aid| {
                    const attr_value: *const Class.AttributeType(aid) =
                        @ptrCast(@alignCast(value));
                    TracedDomain.log.info("read attribute: {s}.{d}:{d}.{s} ->
{any}", .{
                        @tagName(class_id),
                        inst.extract(.tuple_id),
                        inst.extract(.gen_id),
                        @tagName(aid),
                        attr_value.*,
                    });
                },
            },
        },
    }
}
return true;
}
fn exited_state(
    inst: DomainInstanceRef,
    state: u8,
) bool {
    const id: TracedDomain.ClassId = @enumFromInt(inst.extract(.class_index));
    const entry = TracedDomain.class_info.getCatalogEntry(id);
}

```

```

        switch (entry) {
            inline else => |class, class_id| {
                const Class = @TypeOf(class);
                TracedDomain.log.info("exited_state: {s}.{d}:{d}/{s}", .{
                    @tagName(class_id),
                    inst.extract(.tuple_id),
                    inst.extract(.gen_id),
                    @tagName(@as(Class.StateId, @enumFromInt(state))),
                });
            },
        }
        return true;
    }

    fn transitioned(
        inst: DomainInstanceRef,
        source_state: u8,
        event: u8,
        target_state: i8,
    ) bool {
        const id: TracedDomain.ClassId = @enumFromInt(inst.extract(.class_index));
        const entry = TracedDomain.class_info.getCatalogEntry(id);
        switch (entry) {
            inline else => |class, class_id| {
                const Class = @TypeOf(class);
                TracedDomain.log.info("transitioned: {s}.{d}:{d}/{s} - {s} -> {s}", .{
                    @tagName(class_id),
                    inst.extract(.tuple_id),
                    inst.extract(.gen_id),
                    @tagName(@as(Class.StateId, @enumFromInt(source_state))),
                    @tagName(@as(Class.EventId, @enumFromInt(event))),
                    @tagName(@as(Class.TargetStateId, @enumFromInt(target_state))),
                });
            },
        }
        return true;
    }

    fn undefined_transition(
        inst: DomainInstanceRef,
        source_state: u8,
        event: u8,
    ) bool {
        const id: TracedDomain.ClassId = @enumFromInt(inst.extract(.class_index));
        const entry = TracedDomain.class_info.getCatalogEntry(id);
        switch (entry) {
            inline else => |class, class_id| {
                const Class = @TypeOf(class);
                TracedDomain.log.info("undefined transition: {s}.{d}:{d}/{s} - {s} -> never", .{
                    @tagName(class_id),
                    inst.extract(.tuple_id),
                    inst.extract(.gen_id),
                    @tagName(@as(Class.StateId, @enumFromInt(source_state))),
                });
            },
        }
    }
}

```

```

        @tagName(@as(Class.EventId, @enumFromInt(event))),
    });
},
return true;
}
fn continue_event(
    inst: DomainInstanceRef,
    event: u8,
) bool {
    const id: TracedDomain.ClassId = @enumFromInt(inst.extract(.class_index));
    const entry = TracedDomain.class_info.getCatalogEntry(id);
    switch (entry) {
        inline else => |class, class_id| {
            const Class = @TypeOf(class);
            TracedDomain.log.info("continue event: {s}.{d}:{d}/{s}", .{
                @tagName(class_id),
                inst.extract(.tuple_id),
                inst.extract(.gen_id),
                @tagName(@as(Class.EventId, @enumFromInt(event))),
            });
        },
    }
    return true;
}
fn signaled_event(
    source_inst: DomainInstanceRef,
    target_inst: DomainInstanceRef,
    event: u8,
) bool {
    const source_id: TracedDomain.ClassId = @enumFromInt(source_inst.extract(.class_index));
    const target_id: TracedDomain.ClassId = @enumFromInt(target_inst.extract(.class_index));
    const target_entry = TracedDomain.class_info.getCatalogEntry(target_id);
    switch (source_id) {
        inline else => |src_class_id| {
            switch (target_entry) {
                inline else => |target_class, target_class_id| {
                    const TargetClass = @TypeOf(target_class);
                    TracedDomain.log.info("signaled event: {s}.{d}:{d} - {s} -> {s}.{d}:{d}", .{
                        @tagName(src_class_id),
                        source_inst.extract(.tuple_id),
                        source_inst.extract(.gen_id),
                        @tagName(@as(TargetClass.EventId, @enumFromInt(event))),
                        @tagName(target_class_id),
                        target_inst.extract(.tuple_id),
                        target_inst.extract(.gen_id),
                    });
                },
            },
        },
    }
}

```

```

        },
    }
    return true;
}
fn signaled_delayed_event(
    source_inst: DomainInstanceRef,
    target_inst: DomainInstanceRef,
    event: u8,
    delay: u32,
) bool {
    const source_id: TracedDomain.ClassId = @enumFromInt(source_inst.extract(.class_index));
    const target_id: TracedDomain.ClassId = @enumFromInt(target_inst.extract(.class_index));
    const target_entry = TracedDomain.class_info.getCatalogEntry(target_id);
    switch (source_id) {
        inline else => |src_class_id| {
            switch (target_entry) {
                inline else => |target_class, target_class_id| {
                    const TargetClass = @TypeOf(target_class);
                    TracedDomain.log.info("signaled delayed event: {s}.{d}:{d} - {s} -> {s}.{d}:{d} @ {d}", .{
                        @tagName(src_class_id),
                        source_inst.extract(.tuple_id),
                        source_inst.extract(.gen_id),
                        @tagName(@as(TargetClass.EventId, @enumFromInt(event))),
                        @tagName(target_class_id),
                        target_inst.extract(.tuple_id),
                        target_inst.extract(.gen_id),
                        delay,
                    });
                },
            },
        },
    },
    return true;
}
fn canceled_delayed_event(
    source_inst: DomainInstanceRef,
    target_inst: DomainInstanceRef,
    event: u8,
) bool {
    const source_id: TracedDomain.ClassId = @enumFromInt(source_inst.extract(.class_index));
    const target_id: TracedDomain.ClassId = @enumFromInt(target_inst.extract(.class_index));
    const target_entry = TracedDomain.class_info.getCatalogEntry(target_id);
    switch (source_id) {
        inline else => |src_class_id| {
            switch (target_entry) {
                inline else => |target_class, target_class_id| {
                    const TargetClass = @TypeOf(target_class);
                    TracedDomain.log.info("canceled delayed event: {s}.{d}:{d} - {s} -> {s}.{d}:{d} @ {d}", .{
                        @tagName(src_class_id),
                        source_inst.extract(.tuple_id),
                        source_inst.extract(.gen_id),
                        @tagName(@as(TargetClass.EventId, @enumFromInt(event))),
                        @tagName(target_class_id),
                        target_inst.extract(.tuple_id),
                        target_inst.extract(.gen_id),
                        delay,
                    });
                },
            },
        },
    },
    return true;
}

```

```
TracedDomain.log.info("canceled delayed event: {s}.{d}:{d} - {s} -> {s}.{d}:{d}", .{
    @tagName(src_class_id),
    source_inst.extract(.tuple_id),
    source_inst.extract(.gen_id),
    @tagName(@as(TargetClass.EventId, @enumFromInt(event))),
    @tagName(target_class_id),
    target_inst.extract(.tuple_id),
    target_inst.extract(.gen_id),
});
},
}
},
},
return true;
}
};
}
```

The second part of the trace logging setup is to bind the logging callback functions to trace registry entries. This step allocates the required memory for the entries and fills in the callback function pointer. Some complication arises since, generally, callback functions have different signature and thus the pointers are different data types.

```
<<domain module declarations>>=
/// The `TraceBinding` type function returns a data type that is suitable
/// for creating variables that store trace registry entries and binds
/// each entry to a callback function. The `TraceDomain` argument
/// is a domain type as returned from the `Domain` type function.
/// The `trace_callbacks` argument is a namespace `struct` that declare
/// a callback function for type of trace. The `TraceLogging` function
/// is a function that returns a suitable type and contains callbacks
/// for logging trace data.
pub fn TraceBinding(
    comptime TracedDomain: type, // domain to trace
    comptime trace_callbacks: type, // namespace struct with callback functions.
) type {
    return struct {
        const Self = @This();

        // Trace entries for a single class.
        const TraceEntries = struct {
            created_inst: ClassTraceRegistry.CreatedInst.Entry,
            deleted_inst: ClassTraceRegistry.DeletedInst.Entry,
            updated_attr: ClassTraceRegistry.UpdatedAttr.Entry,
            read_attr: ClassTraceRegistry.ReadAttr.Entry,
            created_inst_async: ClassTraceRegistry.CreatedInstAsync.Entry,
            transitioned: ClassTraceRegistry.Transitioned.Entry,
            undefined_transition: ClassTraceRegistry.UndefinedTransition.Entry,
            exited_state: ClassTraceRegistry.ExitedState.Entry,
            continue_event: ClassTraceRegistry.ContinueEvent.Entry,
            signaled_event: ClassTraceRegistry.SigaledEvent.Entry,
        };
    };
}
```

```

        signaled_delayed_event: ClassTraceRegistry.SignaledDelayedEvent.Entry,
        canceled_delayed_event: ClassTraceRegistry.CanceledDelayedEvent.Entry,
    });

    // Array of trace entries, one per class.
    pub const DomainTraceEntries = enums.EnumArray(TracedDomain.ClassId, TraceEntries);
    trace_entry_storage: DomainTraceEntries = .initFill(.{
        .created_inst = .{ .trace_fn = &trace_callbacks.created_inst },
        .deleted_inst = .{ .trace_fn = &trace_callbacks.deleted_inst },
        .updated_attr = .{ .trace_fn = &trace_callbacks.updated_attr },
        .read_attr = .{ .trace_fn = &trace_callbacks.read_attr },
        .created_inst_async = .{ .trace_fn = &trace_callbacks.created_inst_async },
        .transitioned = .{ .trace_fn = &trace_callbacks.transitioned },
        .undefined_transition = .{ .trace_fn = &trace_callbacks.undefined_transition },
        .exited_state = .{ .trace_fn = &trace_callbacks.exited_state },
        .continue_event = .{ .trace_fn = &trace_callbacks.continue_event },
        .signaled_event = .{ .trace_fn = &trace_callbacks.signaled_event },
        .signaled_delayed_event = .{ .trace_fn = &trace_callbacks
            .signaled_delayed_event },
        .canceled_delayed_event = .{ .trace_fn = &trace_callbacks
            .canceled_delayed_event },
    }),
};

// Iterator on the `trace_entry_storage` array.
pub fn iterator(
    self: *Self,
) DomainTraceEntries.Iterator {
    return self.trace_entry_storage.iterator();
}

pub fn getPtr(
    self: *Self,
    class_id: TracedDomain.ClassId,
) DomainTraceEntries.Entry {
    return self.trace_entry_storage.getPtr(class_id);
}
};
}

```

The final step to create a logging for a domain is to combine the first two parts and create functions that access the bound trace entries to add and remove trace callbacks.

```

<<domain module declarations>>=
/// The `DomainClassTraceLogger` type function returns a data type suitable
/// to use for declaring variables that access class trace data and log
/// it to a scoped domain log function. The `TracedDomain` argument is
/// the data type of the domain whose class trace data is to be logged
/// as returned from the `Domain` type function.
pub fn DomainClassTraceLogger(
    comptime TracedDomain: type,
) type {
    return struct {

```

```

const Self = @This();
const trace_access = TracedDomain.class_trace_access;

const logging_callbacks = TraceLogging(TracedDomain);
trace_entries: TraceBinding(TracedDomain, logging_callbacks) = .{},

/// The `logAllTraces` function adds a trace entry for all class
/// trace types.
pub fn logAllTraces(
    self: *Self,
) void {
    var entry_iter = self.trace_entries.iterator();
    while (entry_iter.next()) |entry| {
        const class_id = entry.key;
        var entries = entry.value;

        trace_access.created_inst.addTrace(class_id, &entries.created_inst);
        trace_access.deleted_inst.addTrace(class_id, &entries.deleted_inst);
        trace_access.updated_attr.addTrace(class_id, &entries.updated_attr);
        trace_access.read_attr.addTrace(class_id, &entries.read_attr);
        trace_access.created_inst_async.addTrace(class_id, &entries
.created_inst_async);
        trace_access.transitioned.addTrace(class_id, &entries.transitioned);
        trace_access.undefined_transition.addTrace(class_id, &entries
.undefined_transition);
        trace_access.exited_state.addTrace(class_id, &entries.exited_state);
        trace_access.continue_event.addTrace(class_id, &entries.continue_event);
        trace_access.signaled_event.addTrace(class_id, &entries.signaled_event);
        trace_access.signaled_delayed_event.addTrace(class_id, &entries
.signaled_delayed_event);
        trace_access.canceled_delayed_event.addTrace(class_id, &entries
.canceled_delayed_event);
    }
}

pub fn unlogAllTraces(
    self: *Self,
) void {
    var entry_iter = self.trace_entries.iterator();
    while (entry_iter.next()) |entry| {
        const class_id = entry.key;
        var entries = entry.value;

        trace_access.created_inst.removeTrace(class_id, &entries.created_inst);
        trace_access.deleted_inst.removeTrace(class_id, &entries.deleted_inst);
        trace_access.updated_attr.removeTrace(class_id, &entries.updated_attr);
        trace_access.read_attr.removeTrace(class_id, &entries.read_attr);
        trace_access.created_inst_async.removeTrace(class_id, &entries
.created_inst_async);
        trace_access.transitioned.removeTrace(class_id, &entries.transitioned);
        trace_access.undefined_transition.removeTrace(class_id, &entries
.undefined_transition);
}

```

```

        trace_access.exited_state.removeTrace(class_id, &entries.exited_state);
        trace_access.continue_event.removeTrace(class_id, &entries.continue_event);
        trace_access.signaled_event.removeTrace(class_id, &entries.signaled_event);
        trace_access.signaled_delayed_event.removeTrace(class_id, &entries
.signaled_delayed_event);
        trace_access.canceled_delayed_event.removeTrace(class_id, &entries
.canceled_delayed_event);
    }
}

/// The `logStateTraces` function adds a trace entry for only
/// the class trace types that pertain to classes that have a
/// state model..
pub fn logStateTraces(
    self: *Self,
) void {
    var entry_iter = self.trace_entries.iterator();
    while (entry_iter.next()) |entry| {
        const class_id = entry.key;
        var entries = entry.value;

        trace_access.created_inst_async.addTrace(class_id, &entries
.created_inst_async);
        trace_access.transitioned.addTrace(class_id, &entries.transitioned);
        trace_access.undefined_transition.addTrace(class_id, &entries
.undefined_transition);
        trace_access.continue_event.addTrace(class_id, &entries.continue_event);
        trace_access.signaled_event.addTrace(class_id, &entries.signaled_event);
        trace_access.signaled_delayed_event.addTrace(class_id, &entries
.signaled_delayed_event);
        trace_access.canceled_delayed_event.addTrace(class_id, &entries
.canceled_delayed_event);
    }
}

pub fn unlogStateTraces(
    self: *Self,
) void {
    var entry_iter = self.trace_entries.iterator();
    while (entry_iter.next()) |entry| {
        const class_id = entry.key;
        var entries = entry.value;

        trace_access.created_inst_async.removeTrace(class_id, &entries
.created_inst_async);
        trace_access.transitioned.removeTrace(class_id, &entries.transitioned);
        trace_access.undefined_transition.removeTrace(class_id, &entries
.undefined_transition);
        trace_access.continue_event.removeTrace(class_id, &entries.continue_event);
        trace_access.signaled_event.removeTrace(class_id, &entries.signaled_event);
        trace_access.signaled_delayed_event.removeTrace(class_id, &entries
.signaled_delayed_event);
    }
}

```

```

        trace_access.canceled_delayed_event.removeTrace(class_id, &entries
.canceled_delayed_event);
    }
}

/// The `logTrace` function adds a single trace entry to the class
/// given by `class_id` for the trace type given by `trace_id`.
pub fn logTrace(
    self: *Self,
    class_id: TracedDomain.ClassId,
    comptime trace_id: ClassTraceId,
) void {
    const class_traces = self.trace_entries.getPtr(class_id);
    const trace_name = @tagName(trace_id);
    const trace_entry = &@field(class_traces, trace_name);
    @field(trace_access, trace_name).addTrace(class_id, trace_entry);
}

pub fn unlogTrace(
    self: *Self,
    class_id: TracedDomain.ClassId,
    comptime trace_id: ClassTraceId,
) void {
    const class_traces = self.trace_entries.getPtr(class_id);
    const trace_name = @tagName(trace_id);
    const trace_entry = &@field(class_traces, trace_name);
    @field(trace_access, trace_name).removeTrace(class_id, trace_entry);
}
};

}
}

```

Domain Services for Data Model Transactions

Domain Services for Data Model Transactions

Domain Services for Client Domains

Domain Services for Clients

```

<<domain public functions>>=
pub fn init() void {
    class_info.init();
    association_info.init();
    generalization_info.init();
}

```

```

<<domain public functions>>=
pub fn create(
    class_id: ClassId,

```

```

        tuple: class_info.Tuple(class_id),
) class_info.InstanceSet(class_id) {
    return class_info.getClassVar(class_id).create(tuple);
}

```

```

<<domain public functions>>=
pub fn createOrError(
    class_id: ClassId,
    tuple: class_info.Tuple(class_id),
) RelVarError!class_info.InstanceSet(class_id) {
    return class_info.getClassVar(class_id).create(tuple);
}

```

```

<<domain public functions>>=
pub fn selectByIdentifier(
    comptime class_id: ClassId,
    comptime ident_num: usize,
    id_attr_values: class_info.IdentifierAttributes(class_id, ident_num),
) RelVarError!class_info.InstanceRef(class_id) {
    return try class_info.getClassVar(class_id).selectByIdentifier(ident_num,
id_attr_values);
}

```

```

<<domain public functions>>=
pub fn selectInitialInstance(
    comptime class_id: ClassId,
    comptime initial_inst_id: @Type(.enum_literal),
) class_info.InstanceRef(class_id) {
    const class_var = class_info.getClassVar(class_id);
    const Class = @TypeOf(class_var);
    const inst_id = @as(Class.InitialInstanceId, initial_inst_id);
    return class_var.selectInitialInstance(inst_id);
}

```

```

<<domain public functions>>=
pub fn readAttribute(
    comptime class_id: ClassId,
    inst_set: class_info.InstanceRef(class_id),
    comptime attr_id: class_info.AttributeId(class_id),
) class_info.AttributeType(class_id, attr_id) {
    return inst_set.readAttribute(attr_id);
}

```

```

<<domain public functions>>=
pub fn updateAttribute(
    comptime class_id: ClassId,
    inst_set: class_info.InstanceSet(class_id),
    comptime attr_id: class_info.AttributeId(class_id),
)

```

```

    value: class_info.AttributeType(class_id, attr_id),
) void {
    inst_set.updateAttribute(attr_id, value);
}

```

```

<<domain module tests>>=
test "class catalog" {
    const domain = @import("lighting.zig");
    const Lighting = domain.Lighting;
    const blinky_led = domain.blinky_led;

    Lighting.init();

    var lighting_trace_all: DomainClassTraceLogger(Lighting) = .{};
    lighting_trace_all.logAllTraces();
    defer lighting_trace_all.unlogAllTraces();
    try testing.expectEqual(@as(usize, 2), enums.values(Lighting.ClassId).len);
    try testing.expectEqual(@as(usize, 2),
Lighting.class_info.getClassVar(.bouncy_button).cardinality());
    try testing.expectEqual(@as(usize, 2),
Lighting.class_info.getClassVar(.blinky_led).cardinality());

    const b2 = try Lighting.selectByIdentifier(.bouncy_button, 0, {.pin = 2});
    try testing.expectEqual(@as(u32, 20), Lighting.readAttribute(.bouncy_button, b2,
.confirm_interval));

    const is_bouncy_r1_domain =
Lighting.class_assoc_domain_map.get(.bouncy_button).contains(.r1);
    try testing.expect(is_bouncy_r1_domain);
    const is_blinky_r1_codomain =
Lighting.class_assoc_codomain_map.get(.blinky_led).contains(.r1);
    try testing.expect(is_blinky_r1_codomain);

    b2.updateAttribute(.relax_interval, b2.readAttribute(.relax_interval) + 50);
    try testing.expectEqual(@as(u32, 60), Lighting.readAttribute(.bouncy_button, b2,
.relax_interval));

    const l3 = blinky_led.selectInitialInstance(.l3);
    try testing.expectEqual(@as(u32, 1000), l3.readAttribute(.on_interval));

    const trigger_event = Lighting.makeDomainEvent(.bouncy_button, b2.makeEvent(.trigger,
.{}));

    // Simulate pressing the button.
    Lighting.runThreadOfControl(trigger_event);

    // Simulate delayed events to make the LED turn off and on.
    const turn_off_event = Lighting.makeDomainEvent(.blinky_led, l3.makeEvent(.turn_off,
.{}));
    const turn_on_event = Lighting.makeDomainEvent(.blinky_led, l3.makeEvent(.turn_on,
.{}));
    Lighting.runThreadOfControl(turn_off_event);

```

```

    Lighting.runThreadOfControl(turn_on_event);

    // Simulate pressing the button again.
    Lighting.runThreadOfControl(trigger_event);
}

```

Domain Code Layout

```

<<domain.zig>>=
/// The `domain.zig` file contains the declarations required to
/// define a ReSEE Domain. A `Domain` is a collection of `RelClass`,
/// `Association`, and `Generalization` components. The recommended
/// organization for a Domain is to place all the `RelClass`,
/// `Association`, and `Generalization` components that constitute the
/// domain and then invoke the `Domain` type function to bind the domain
/// components together.
<<edit warning>>
<<copyright info>>

const std = @import("std");
const builtin = @import("builtin");
const math = std.math;
const enums = std.enums;
const meta = std.meta;
const EnumField = std.builtin.Type.EnumField;
const UnionField = std.builtin.Type.UnionField;

const assert = std.debug.assert;

const testing = if (builtin.os.tag == .freestanding)
    @import("resee_testing")
else
    std.testing;

pub const version = "1.0.0-a1";

const reltuple = @import("reltuple.zig");
const RelTuple = reltuple.RelTuple;

const relvalue = @import("relvalue.zig");

const relvar = @import("relvar.zig");
const RelVarError = relvar.RelVarError;
const GenerationNumber = relvar.GenerationNumber;

const relclass = @import("class.zig");
const RelClass = relclass.RelClass;
const isRelClassType = relclass.isRelClassType;
const ClassInstanceRef = relclass.ClassInstanceRef;
const RelationCapacity = relclass.RelationCapacity;

```

```
const relassoc = @import("association.zig");
const Association = relassoc.Association;
const isAssociationType = relassoc.isAssociationType;
const AssociationSpec = relassoc.AssociationSpec;

const relgen = @import("generalization.zig");
const isGeneralizationType = relgen.isGeneralizationType;

const state_model = @import("state_model.zig");
const StateModelSpec = state_model.StateModelSpec;

<<domain module declarations>>

<<domain module public functions>>

<<domain module private functions>>

<<domain module tests>>
```

[1] Sometimes called the *business purpose*.

Button Blinky



Due to the reorganization of some chapters relative to the original version of the book, several of the examples in this section will be moved when the required facilities are implemented. For now, we simply leave the previous implementation, which is not tangled out in the Zig version.

A previous [example application](#) combines blinking an LED under the control timer expirations. The resulting program starts and stops blinking of the LED after some time elapses.

Blinky LED State Model

The diagram below is a state model diagram for blinking the LED. The graphical notation is a restricted subset of UML state diagram notation. Each yellow box represents a state. Arrows represent events and are labeled by an event name. An arrow connecting two boxes implies a state transition when event is received. The arrow pointing from the black circle to a yellow box indicates the default initial state of the model. When a state machine is created, it is placed in the initial state *without* executing the associated action. Only when the machine transitions, is the action contained in the state box executed. The actions are described by *pseudo-code* statements which express the intent of the action's computation.

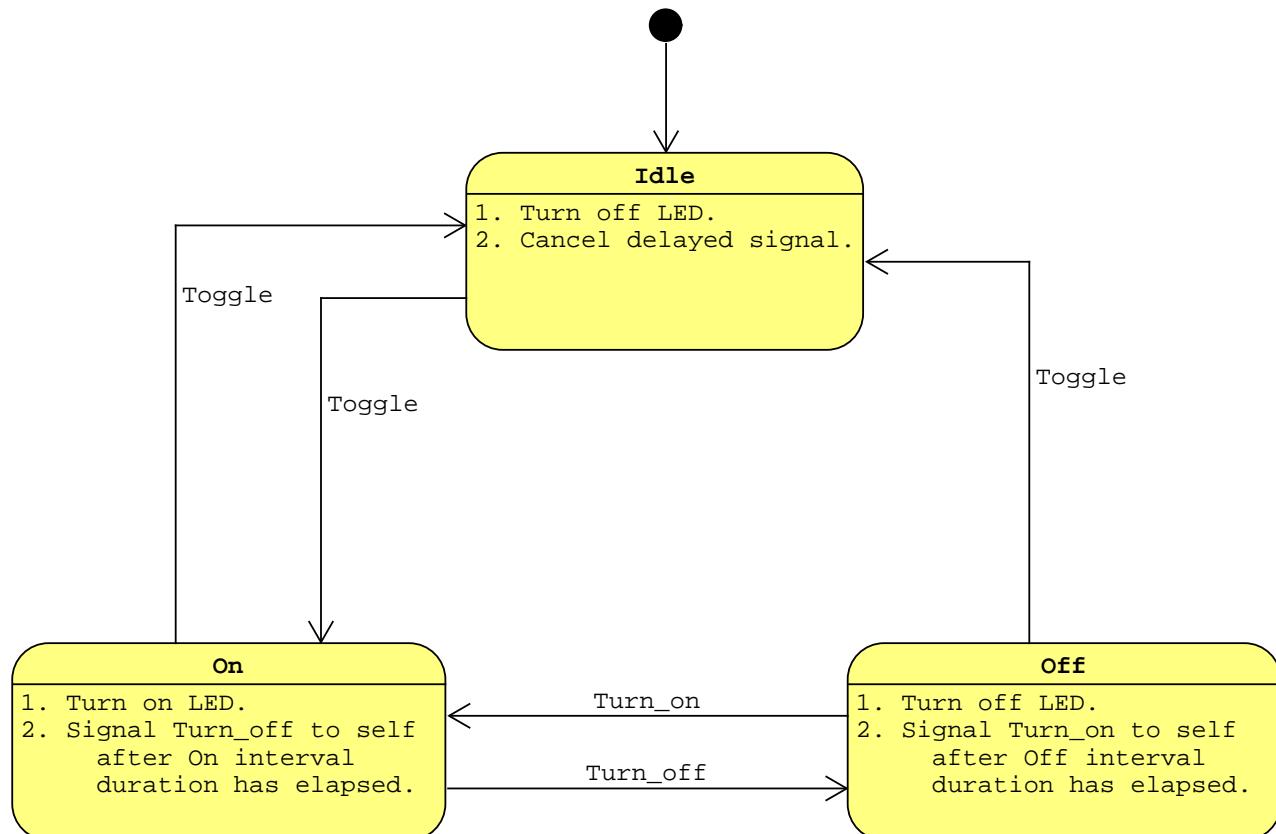


Figure 59. Blinky State Model

For an LED, the state model starts in the **IDLE** state. Since no action is executed when the machine is created, it is assumed that initialization code has turned off the LED. In this case, the output pin is

zero after reset. When the **Toggle** event is received, the transition to the **ON** state is made. The action of the **ON** state turns on the LED and arranges for the **Turn_off** event to be signaled at some time in the future. When the **Turn_off** event arrives the transition to the **OFF** state happens. The **OFF** state action turns off the LED and arranges for the **Turn_on** event to be signaled at some future time. The sum of the on and off intervals is contrived to be one second yielding a 1 Hz blink rate. This behavior continues repeating on/off, on/off, etc. as the state transitions continue to be driven by time delayed events. At some time, when the **Toggle** event is received, the machine transitions to the **IDLE** state where the action turns off the LED and cancels any delayed signal created by either the **ON** or **OFF** states.

Note, this is *not* the only way this behavior could be captured in a state model. In general, a given problem may have multiple state model solutions in much the same manner that it would have multiple directly programmed solutions. This model emphasizes the correspondence of the illumination state of the physical LED with the conceptual state of the model both of which are separate from the behavior to initiate and terminate a continuing blinking sequence. An alternative state model might have only two states and use the ability to toggle a GPIO pin value. That effectively holds part of the state of the LED pin value in the hardware.

In our formulation of finite state machines, a careful distinction is made between a *state model* and a *state machine*. A state model is used to describe the behavior shown in the previous diagram. A state machine is an instance of a state model associated with some parametric data. All state machine instances of a given state model behave the same. However, each state machine instance has its own notion of its current state and access to any parametric data associated with the particular state machine. This implies there may be multiple state machines based on the same state model, all executing independently and generally in different states, yet the state machines go through the same transitions, exhibit the same behavior, and execute the same code.

The last major component of a state model is the transition function. This function maps the current state and an event into a new target state. The new state may be any state defined by the model or one of two non-transitioning states. Here, two additional transition rules are introduced which are encoded as a non-transitioning target state.

- **ignore** as a target state means that the event is ignored. Sometimes events simply arrive too late to be considered. For example, a state model describing a device which can be powered up and down does not care if a **Power down** event arrives when it is already powered off. The event simply arrived too late. Ignoring an event can be implemented by adding states to the model, but the need is so common that ignored states would significantly clutter the diagram.
- **never** as a target state means that the event logically can't happen in the given state. If an event is received and the new target state is **never**, no transition happens and it is considered an error condition. For example, in the Blinky state model, the **Turn_on** event cannot happen when a machine is in the **ON** state. This is because the event is only generated by the **OFF** state and it was the receipt of the **Turn_on** event while in the **OFF** state that causes the transition to the **ON** state. If the **Turn_on** event is received while in the **ON** state, something is terribly wrong. Can't happen transitions are usually the result of errors in problem analysis.

Note that the non-transitioning target states of **ignore** and **never** do *not* appear on the state model graphical diagram. Only the transitions to other states in the model show up in the diagram. This is because the non-transitioning states have little impact on the reasoning about the actions of the model and so adding them would contribute to diagram clutter. For these simple examples, the state model diagrams are equally simple. However, for state models larger to more than a handful of states, it is beneficial to reasoning about the behavior to keep the diagrams simple and avoid any unnecessary clutter.

The transition function is usually encoded as a two-dimensional array of states number of rows and events number of columns. The following table show the transition matrix for the Blinky state model.

	Toggle	Turn_on	Turn_off
Idle	On	ignore ①	ignore ①
On	Idle	never	Off
Off	Idle	On	never

Table 17. Blinky Transition Matrix

① The **Turn_on** and **Turn_off** events are ignored in the **Idle** state because there is a race condition for delivery of the delayed events. It is possible for the GPIO IRQ handler to queue a background notification and then for the Timer Queue IRQ handler to queue a time expiration before the pin notification can be dispatched and acted upon. Since the Timer Queue IRQ handler can preempt the state machine action execution, the background notification from the timer queue could already be in flight when the attempt to cancel the timer happens in the **IDLE** state. Again, this is just a case of the event arriving too late.

The definition of the transition function is a direct transliteration of the above transition matrix into pre-processor macro invocations.

Blinky LED State Machine

Usually, the actions of a state model access other parametric data. This is accomplished by embedding the state machine in a `struct` which contains the required data. For the Blinky model, the state actions refer to an `on` interval and an `off` interval. These are time delays for when an event should be signaled.

Bouncy Button State Model

Momentary contact push buttons usually exhibit a phenomenon called, *bouncing*. Bouncing occurs when the metal plates of the switch come together and can cause the signal transition from inactive to active to have ripples in it. These ripples can be large enough for the input pin to detect them as multiple transitions. Combined with a spring that causes the button to release, the signal seen by the input pin is *not* the clean square wave on either its rising or falling edge. This certainly happens for our case as was demonstrated by the previous button example.

Noisy switch signals can be handled in a number of ways. Sometimes there are *glitch* filters provided on the I/O pin by the microcontroller. It is possible to use external filtering electronics to smooth the signal so that it appears like a square wave. However, frequently it falls on software to provide ``debouncing'' for push buttons.

There are many techniques to debounce a switch in software. Most techniques involve the idea of waiting some time after the initial signal change to make sure the signal value stabilizes. In this example, switch debouncing is implemented in a state model. The idea is to introduce a *confirmation* interval. After the signal transition from inactive to active is reported (*i.e.* by the pin interrupt), a timer is used to measure the confirmation interval. During the confirmation interval, the input signal is ignored. At the end of the confirmation interval, the signal is read to determine if it is active. If it is active, the button was held down long enough for the signal to stabilize. If the signal is inactive at the end of the confirmation interval, it is deemed a ``glitch'' and the original notification of the change in pin state is ignored. The time needed for the signal to stabilize is quite short, usually much less than few milliseconds. However, a long confirmation interval can be used to *slow* the recognition of the button press. This would force a user to hold down the button longer before the press would be recognized and can be used to prevent an accidental, glancing touch of a button from triggering a response.

Another useful concept is one of a *relaxation* interval. The *relaxation* interval is an amount of time to wait after a button press is confirmed but before arming the pin for another press. The relaxation interval has the effect of limiting the rate at which the button can be pressed and still be recognized.

The following state model diagram shows these ideas.

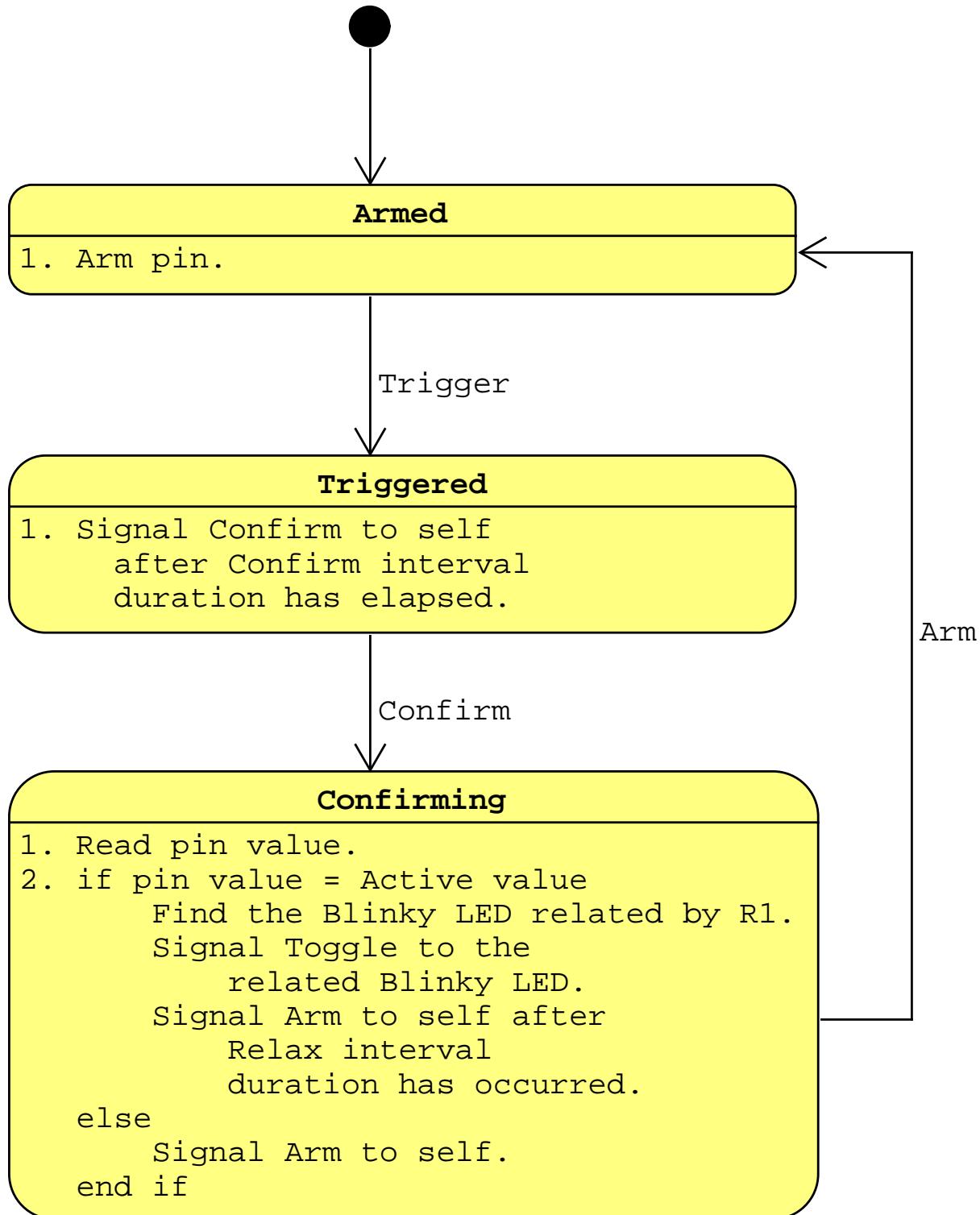


Figure 60. Bouncy State Model

Again, it is most important to supply a completed transition matrix.

	Trigger	Confirm	Arm
Armed	Triggered	never	never
Triggered	never	Confirming	never
Confirming	never	never	Armed

Table 18. Button Debounce Transition Matrix

Bouncy Button State Machine

Like the Blinky LED state machine, a structure is defined to hold the state machine dispatch mechanisms and the parametric data needed by the state actions.

An Example Domain

In this section we present the implementation of the simple model first described in [Chapter 13: The Execution Model](#). This is simple, two class, on association model that dealt with using a push button to control the blinking of an LED.

First, the bouncy button class.

```
<<lighting domain>>
const BouncyButtonHeading = struct {
    pin: u8,
    active_value: u1,
    confirm_interval: u32,
    relax_interval: u32,
};

const bouncy_button_model_spec: StateModelSpec = .{
    .state_names = &.{ "armed", "triggered", "confirming" },
    .event_names = &.{ "trigger", "confirm", "arm" },
    .transitions = &.{

        .{
            "armed", &.{

                .{ "trigger", "triggered" },
            },
        },
        .{
            "triggered", &.{

                .{ "confirm", "confirming" },
            },
        },
        .{
            "confirming", &.{

                .{ "arm", "armed" },
            },
        },
    },
    .default_initial_state = "armed",
    .StateActivities = bouncy_button_activities,
};
```

```

    .unspecified_transition = "never",
    .terminal_states = &.{},
};

const bouncy_button_activities = struct {
    const InstanceRef = BouncyButton.InstanceRef;
    const log = Lighting.log;

    pub fn armed(
        inst: InstanceRef,
    ) void {
        log.debug("in 'armed' state:\n{f}", ^{
            inst.dereference(),
        });
    }

    pub fn triggered(
        inst: InstanceRef,
    ) void {
        log.debug("in 'triggered' state:\n{f}", ^{
            inst.dereference(),
        });

        inst.continueWith(.confirm, .{}); // For now, delayed events are not available.
    }

    pub fn confirming(
        inst: InstanceRef,
    ) void {
        log.debug("in 'confirming' state:\n{f}", ^{
            inst.dereference(),
        });

        const led = inst.traverse(r1, null, .blinky_led);
        inst.signal(led, .toggle, .{});
        inst.continueWith(.arm, .{}); // For now, delayed events are not available.
    }
};

pub const BouncyButton = RelClass(
    .bouncy_button,
    BouncyButtonHeading,
    &.{ &{.pin} },
    &.{},
    &bouncy_button_model_spec,
    5,
    null,
    &.{{
        .{
            .b2,
            .{
                .pin = 2,
                .active_value = 1,
                .confirm_interval = 20,
    
```

```

        .relax_interval = 10,
    },
},
.{
    .b4,
.{
    .pin = 4,
    .active_value = 1,
    .confirm_interval = 40,
    .relax_interval = 5,
},
},
},
),
);
const bouncy_button: BouncyButton = .{};


```

Next, the blinky LED.

```

<<lighting domain>>
const BlinkyLedHeading = struct {
    pin: u8,
    active_value: u1,
    inactive_value: u1,
    on_interval: u32,
    off_interval: u32,
};
const blinky_led_model_spec: StateModelSpec = .{
    .state_names = &.{ "idle", "on", "off" },
    .event_names = &.{ "toggle", "turn_on", "turn_off" },
    .transitions = &.{.
        .{
            "idle", &.{.
                .{ "toggle", "on" },
            },
        },
        .{
            "on", &.{.
                .{ "toggle", "idle" },
                .{ "turn_off", "off" },
            },
        },
        .{
            "off", &.{.
                .{ "toggle", "idle" },
                .{ "turn_on", "on" },
            },
        },
    },
    .default_initial_state = "idle",
    .StateActivities = blinky_led_activities,
    .unspecified_transition = "never",
};


```

```

.terminal_states = &.{},
};

const blinky_led_activities = struct {
    const InstanceRef = BlinkyLed.InstanceRef;
    const log = Lighting.log;

    pub fn idle(
        inst: InstanceRef,
    ) void {
        log.debug("in 'idle' state:\n{f}", .{
            inst.dereference(),
        });
    }

    pub fn on(
        inst: InstanceRef,
    ) void {
        log.debug("in 'on' state:\n{f}", .{
            inst.dereference(),
        });
    }

    pub fn off(
        inst: InstanceRef,
    ) void {
        log.debug("in 'off' state:\n{f}", .{
            inst.dereference(),
        });
    }
};

pub const BlinkyLed = RelClass(
    .blinky_led,
    BlinkyLedHeading,
    &.{ &{.pin} },
    &{},
    &blinky_led_model_spec,
    5,
    null,
    &.{

        .{

            .l3,
            .{

                .pin = 3,
                .active_value = 1,
                .inactive_value = 0,
                .on_interval = 1000,
                .off_interval = 1000,
            },
        },
        .{

            .l5,
            .{

```

```

        .pin = 5,
        .active_value = 1,
        .inactive_value = 0,
        .on_interval = 500,
        .off_interval = 500,
    },
},
},
);
pub const blinky_led: BlinkyLed = .{};


```

The single association relates the two classes.

```

<<lighting domain>>=
// Bouncy Button 1-->"has blinking controlled by"-R1-->"controls the blinking of"-1 Blinky LED
const r1_spec = AssociationSpec{
    .DomainRelClass = BouncyButton,
    .codomain_ref = .@"1",
    .codomain_phrase = .@"has blinking controlled by",
    .assoc_tag = .r1,
    .domain_phrase = .@"controls the blinking of",
    .domain_ref = .@"1",
    .CodomainRelClass = BlinkyLed,
};

pub const R1 = Association(
    &r1_spec,
    &.{.
        .{ .b2, .l3 },
        .{ .b4, .l5 },
    },
);
pub const r1: R1 = .{};


```

And finally, the Lighting domain is defined to contain the classes and association.

```

<<lighting domain>>=
pub const Lighting = Domain(
    .lighting,
    &.{.
        BouncyButton,
        BlinkyLed,
    },
    &.{.
        R1,
    },
    &.{},
    null,
);


```

Tests

```
<<lighting domain>>
test "button_blinky" {
    testing.log_level = .info;

    Lighting.init();

    var lighting_trace_all: DomainClassTraceLogger(Lighting) = .{};
    lighting_trace_all.logAllTraces();
    defer lighting_trace_all.unlogAllTraces();

    const b2 = Lighting.selectInitialInstance(.bouncy_button, .b2);
    const trigger_event = Lighting.makeDomainEvent(.bouncy_button, b2.makeEvent(.trigger,
    .{}));

    // Simulate pressing the button.
    Lighting.runThreadOfControl(trigger_event);

    // Simulate delayed events to make the LED turn off and on.
    const l3 = Lighting.selectInitialInstance(.blinky_led, .l3);
    const turn_off_event = Lighting.makeDomainEvent(.blinky_led, l3.makeEvent(.turn_off,
    .{}));
    const turn_on_event = Lighting.makeDomainEvent(.blinky_led, l3.makeEvent(.turn_on,
    .{}));
    Lighting.runThreadOfControl(turn_off_event);
    Lighting.runThreadOfControl(turn_on_event);

    // Simulate pressing the button again.
    Lighting.runThreadOfControl(trigger_event);
}
```

Lighting Domain Code Layout

```
<<lighting.zig>>
//! The `lighting.zig` file contains an example ReSEE `Domain` definition.
//! The domain consists of two classes and a single association. This domain
//! is used as an example in the _Code to _Models_ book.
<<edit warning>>
<<copyright info>>

const std = @import("std");
const testing = std.testing;

const relclass = @import("class.zig");
const RelClass = relclass.RelClass;
const ClassInstanceRef = relclass.ClassInstanceRef;

const relassoc = @import("association.zig");
const Association = relassoc.Association;
const AssociationSpec = relassoc.AssociationSpec;
```

```

const state_model = @import("state_model.zig");
const StateModelSpec = state_model.StateModelSpec;

const domain = @import("domain.zig");
const Domain = domain.Domain;
const DomainClassTraceLogger = domain.DomainClassTraceLogger;

<<lighting domain>>

```

Background Notification Handling

The state activities of the state models contain code to make direct background requests for GPIO pin service and for Timer Queue service^[1]. But when the input pin changes state or the timer queue element has expired, a background notification is queued. The background notifications must be mapped onto state machine events. This mapping completes the ``circuit'' between foreground and background processing.

For both the GPIO and timer cases, the ability of the background notifications to carry a piece of *closure* data is used to serve as the mechanism for mapping the notifications to state machine events. The GPIO and timer queue services are general services which are not concerned about what the ultimate use is. They simply report that a pin has changed state or an interval of time has expired. It is up to the code to map those basic concepts onto the state machines which drive the execution.

The push button is the simpler situation. Each input pin has a background proxy function associated with it and the desired event number to signal is passed as the closure value. In this case the event is symbolically encoded as, `bute_Toggle`. The following code shows how the pin attached to the button is configured for input.

The mapping of delayed signal enumerator value to state machine / event pairs is implemented as an array, indexed by the delayed signal value. Each element of the mapping array is a state machine / event pair which is signaled to the target state machine as an event. The net effect is, as it appears to the state machines, is an ordinary event. The effect as seen from a system perspective is that the signaled event was delayed in time from the execution of the state action.

Note that in both of the background proxy functions, the state machines are signaled events using the `SSM_signalAndRun` function. This function both signals the event and then immediately dispatches the event as well as any other events signaled as the consequence of the delivery of the original event. For example, when the button press fails to be confirmed, the **CONFIRMING** state signals the **Arm** event to itself in order to be ready for any subsequent button presses. If that is the path taken by the code of the **CONFIRMING** state activity, then the Bouncy state model ends up in the **ARMED** state by the time `SSM_signalAndRun` returns.

Tracing execution

The following is a trace of the event dispatch for a run of the example program. The trace shows the button press to start the LED blinking and a short time later a button press to stop the blinking. The trace entries give the name of the state machine and the event it receives. After the colon, the transition from current state to new state is given.

Event Dispatch Trace

```
bouncy-machine <- Trigger: ARMED -> TRIGGERED
```

Execution sequence diagram

A sequence diagram is another useful way to reason about the actions of the state machines and to show the collaboration of the state machines with device requests. The following diagram shows the details of the interactions between the Bouncy and Blinky state machines and the interactions between the state machines and the GPIO pins and timer elements. The diagrams shows, for this case, that most of the interactions are between the state machines and the GPIO device pins and the Timer Queue. Items 6 and 22 show the only interactions between the two state machines.

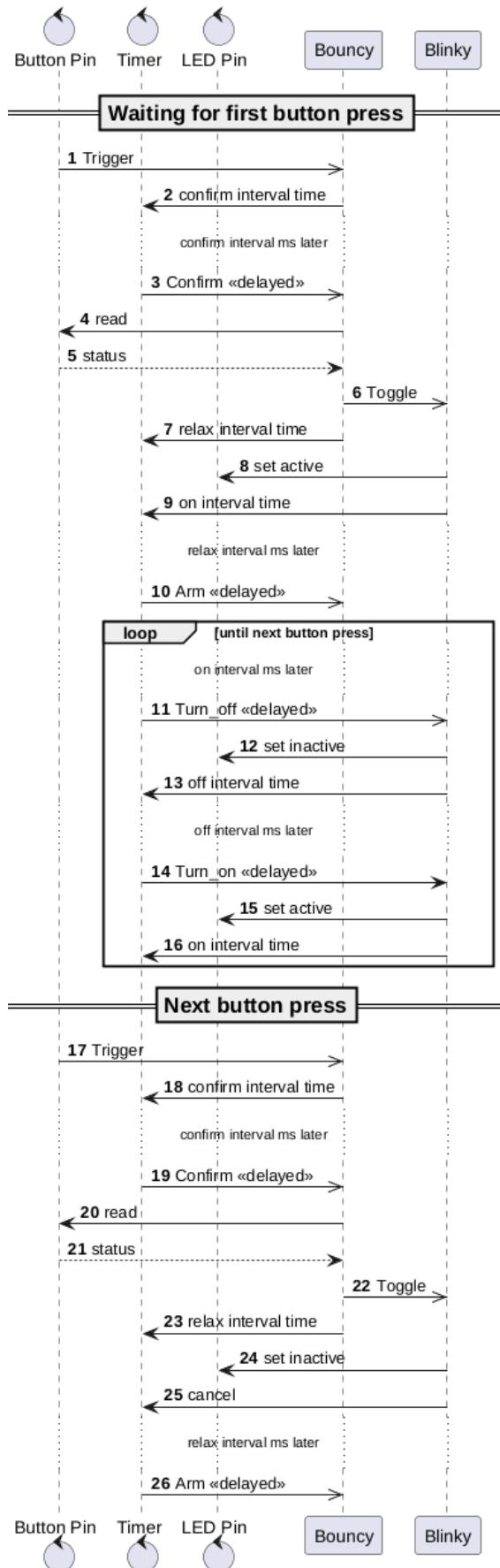


Figure 61. Sequence Diagram of Button Controlled LED Blinking

Evaluation

As mentioned at the beginning of this example, the manner in which background notifications are used to sequence background processing to include state machines was expanded. Previous examples had used direct, single level callbacks to initiate background processing in response to a background notification from an IRQ. It is fitting to examine the benefits and drawbacks of this method of controlling background execution.

Clearly, if the sole purpose of the application was to blink an LED under control of a button, the state machine formalism used is overly elaborate. There are simpler ways to accomplish manipulating some I/O pins. However, the following benefits can be seen.

- It is clear from the state model diagrams *what* computations take place without have to read through any code. The diagram does *not* indicate from where the events come, only what happens when they are received.
- There is direct association of state and action. If the state machine *transitions* into a state, the action code for that state is executed. This is true even if the state machines transitions back into the same state.
- State actions have access to parametric data which describes the characteristics of the object being controlled and provide any required data parameters.
- All transitions are accounted for. Any event defined for a state model is resolved by the transition matrix to some well defined action or non-transitioning situation.
- The same state model can be used for multiple button / LED combinations, *i.e.* the fixed cost of the state machine dispatch mechanism is amortized across all state machine instances.
- There is a clear separation of responsibility between the button and the LED state models. The button state model issues **Toggle** events to whatever state model it is associated to by its `controlled_led` data member. The LED state model is solely concerned with turning on and off the physical LED. The overall requirements to blink the LED under control of a button is achieved by *composing* the two state models where one state machine signals an event to another state machine. It is easier to reason about the behavior because of the separation than if the problem had been solved with a single state model which handled both the button debouncing and the timing of the LED illumination. A single state model solution would have involved more states and more paths through the state model.
- The interaction of the state machines can be shown in chronological order by examining the event dispatch trace. The execution trace becomes vital when a larger number of state machines interact.

The following drawbacks can be seen.

- It is not possible to read the source which implements the state models and determine the order in which processing happens. The state actions are placed in functions so it is easy to see what happens when a given state is entered. But execution sequencing is determined by generalized code operating over data structures and the same code is capable of dispatching events for any state model. The usual sequential execution model, *i.e.* one code statement is executed after another is lost in all but the state actions themselves.
- Breakpoint style debugging has limited utility. Outside of state actions, it is difficult to know which path a state machine will take. Since there may be multiple state machines executing according to the same state model, tracking instance specific execution is difficult within a breakpoint style debugging scheme.
- Specifying the state model using the pre-processor macros is laborious. For a few state models, one can simply roll up your sleeves and push through. However, if there were 30 or 40 state models, as might be found in a larger application, the pre-processor macros don't scale well to

handle the amount of specification data required.

- Much of the specification information required for a state model amounts to arbitrary encoding of semantically meaningful names into integers. That encoding is somewhat fragile, despite our use of *designated initializers* to reduce the order dependency of generating initialized array variables for the state model specification data. When the inevitable changes to a state model arrive, the encoding must be carefully preserved. This is more difficult given the manner in which the state model is specified in this example, since adding or subtracting a state or event requires changes in multiple places.

It will come as no surprise that when background execution is discussed in later parts of this book, techniques will be employed that seek to maximize the benefits while minimizing the drawbacks.

[1] This is not the best design, but simplifies the situation for the benefit of the example.

Islands in the Stream

Abstract

Part III of the book is not in place. It will discuss how subject matter decomposed domains, as presented in Part II, are wired together to form complete software systems.

Introduction

Introduction

Supporting code

Memory Mapped I/O

In this chapter we define the mechanisms used to access system hardware whose control is mapped to an address in memory. This design borrows heavily from the ideas described [here](#) and extends those ideas to handle other register field arrangements commonly found in hardware peripherals.

There are two main insights the above link describes:

1. Memory mapped registers often differ in what they return when reading versus what they accept when writing. Some registers are read-only, others are write-only, most are both read and write, and frequently different access types are mixed together in the same register. In the ARM world, it is common to have registers that are write one to clear and read as zero. This leads to the idea of describing both the read and write layouts of a register.
2. Zig provides a `packed struct` construct that can assign names to the layout of bits in a register in a determined order. Zig also provides for variable sized integer values, e.g. `u6` is a 6 bit unsigned integer. The combination of `packed struct` and variable sized integer types give a convenient mechanism to describe the layout of a hardware register. If the bit access of a register value is known at compile time, the Zig compiler often produces specific ARM Thumb bit field instructions.

Hardware designers contrive many different arrangements of bits in memory words to be used for hardware control. Broadly speaking, the arrangements fall into the following categories:

Named bit fields

These fields consist of a bit offset into a memory word and a length counting the number of bits in the field.

Indexed fields

Sometimes, hardware consists of replicated instances of the same set of controls. In this case, it is more convenient to treat the replication as an array and to access the controls via an index.

Bit registers

Bit registers are those consisting of a set of single bit fields. They are a degenerate case of indexed fields, but common enough to warrant special consideration.

Foundational Definitions

We start with some basic definitions that are used throughout. These terms are typical in the ARM ecosystem.

```
<<mmio: foundations>>=
/// Following ARM terminology, `words` are 32 bits. This type clarifies that
/// the underlying integer type is being used as a peripheral register.
pub const WordRegister = u32;
/// Following ARM terminology, `half words` are 16 bits. This type clarifies
/// that the underlying integer type is being used as a peripheral register.
pub const HalfWordRegister = u16;
/// Following ARM terminology, `bytes` are 8 bits. This type clarifies that the
/// underlying integer type is being used as a peripheral register.
pub const ByteRegister = u8;
/// Some peripheral registers are read only or write only. The
```

```

/// `NoAccessRegisterType` serves to provide a format for those registers that
/// have no meaningful fields for the direction of access.
pub const NoAccessRegisterType = packed struct(WordRegister) {
    _reserved: WordRegister,
};

```

Naming Register Fields

Much of the value of traditional CMSIS-core is simply supplying naming information about the fields in hardware registers. Frequently, those names are known at compile time and used directly. For example, with a register that contains a field named, `offset_cnt`, we would want the code to include the characters in "offset_cnt" when it is accessed in code. We don't particularly want to pass around a lot of string information that would take up memory to store (remember, there can be thousands of field names in an SOC's peripheral hardware set). Zig provides the function `std.enums.FieldEnum` that will generate an enumeration for each field in container type (e.g. `struct`). This allows us to access fields by an enumeration literal. In the case of the `offset_cnt` field, we can refer to it as `.offset_cnt`. The generated enumeration type returned by `FieldEnum` has its enumerator values assigned so that it is the logical equivalent to an index into the fields of a register. Note that all the values of the enumeration literals in this case are generated by the compiler.

The use of an enumeration literal value as a means to specify a register field is only useful if the enumeration literal is known at compile time. That is the most frequent case, however it is not the only case. Sometimes we want to define the bit field within a register in such a way that it can be accessed by a runtime value. This case shows up when register access is generalized and the access information is stored in a data structure as descriptive information. We need both a way to define a runtime bit field and a way to use that definition. First, we start with just defining a bit field that can be used at runtime to access a register.

```

<<mmio: bitfields>>=
/// An unsigned integer type that can hold a bit offset of a field in a WordRegister.
pub const BitFieldOffset = math.Log2Int(WordRegister);
/// An unsigned integer type that can hold the length in bits of a field in a WordRegister.
pub const BitFieldLength = math.Log2Int(WordRegister); // no 32-bit bitfields
/// The `DefineBitField` type function returns a type suitable to hold the
/// definition of a bit field that may be resolved at run time to manipulate
/// the field contents. Usually, field access is provided by `comptime` known
/// bit fields. However, cases do arise when it is useful to store bit field
/// information and use it at runtime to obtain the same types of field reads and
/// updates available from the functions that use compile time bit field descriptions.
fn DefineBitField(
    comptime T: type,
) type {
    return struct {
        offset: BitFieldOffset,
        length: BitFieldLength,

        const FieldId = meta.FieldEnum(T);
        const Self = @This();

        pub fn init(
            comptime field_id: FieldId,

```

```

) Self {
    const offset = @bitOffsetOf(T, @tagName(field_id));
    const length = @bitSizeOf(@FieldType(T, @tagName(field_id)));
    std.debug.assert(offset + length < 32);
    return .{
        .offset = offset,
        .length = length,
    };
}

fn fieldMask(
    self: Self,
) WordRegister {
    return (@as(WordRegister, 1) << self.length) - 1) << self.offset;
}

pub fn insertField(
    self: Self,
    reg_value: WordRegister,
    field_value: WordRegister,
) WordRegister {
    const mask = self.fieldMask();
    const shifted_value = field_value << self.offset;
    const masked_value = shifted_value & mask;
    return (reg_value & ~mask) | masked_value;
}

pub fn extractField(
    self: Self,
    reg_value: WordRegister,
) WordRegister {
    const mask = self.fieldMask();
    return (reg_value & mask) >> self.offset;
}
};

}

```

Register Meta-programming

To bring stronger static typing to memory mapped registers, we use Zig's `comptime` meta-programming to define the layout of a register. This takes the form of a `Register` function which accepts the data type of the register from the read perspective and the data type of the register from the write perspective. The returned data type can then be used as part of the register description for a peripheral device. The description of the whole peripheral device is contained in an `extern struct` construct which provides the required determined order in memory.

```

<<mmio: register>>=
/// The `Register` type function defines a data type and a set of methods used
/// to access the fields of a memory mapped peripheral device register. The
/// `Read` argument is a data type that is assumed to be a `packed struct`
/// definition for the fields of the register when it is read from memory. The

```

```

/// `Write` argument is a data type that is assumed to be a `packed struct`
/// definition for the fields of the register when it is written to memory.
/// The differences between these types can be used to control read only or
/// write only fields within the register. The returned `type` has associated
/// methods that access the fields of the register according to the `Read` and
/// `Write` type definitions. Register fields are identified by enumeration
/// literals whose names match those of the field declarations of the `Read`
/// and `Write` types.
pub fn Register(
    comptime Read: type,
    comptime Write: type,
) type {
    return packed struct {
        /// The raw memory associated with the register.
        reg_value: WordRegister,

        const Self = @This();

        /// It is convenient to hold the argument types internally.
        pub const ReadType = Read;
        pub const WriteType = Write;
        /// An enumerated type with a tag for each field in the `ReadType`.
        pub const ReadField = meta.FieldEnum(Read);
        /// An enumerated type with a tag for each field in the `WriteType`.
        pub const WriteField = meta.FieldEnum(Write);
        /// Define a bit field type for run-time specified bit fields.
        pub const BitField = DefineBitField(ReadType);

        /// The `readDirect` function reads the memory location of the register
        /// and returns its value as an uninterpreted integer number. This is a
        /// so called, _raw read_, of the register.
        pub fn readDirect(
            self: *volatile Self,
        ) WordRegister {
            return self.reg_value;
        }

        /// The `writeDirect` function writes the memory location of the
        /// register and with an uninterpreted integer number. This is a so
        /// called, _raw write_, to the register.
        pub fn writeDirect(
            self: *volatile Self,
            value: WordRegister,
        ) void {
            self.reg_value = value;
        }

        /// The `read` function reads the value of the peripheral register
        /// returning that value typed according to the `ReadType` of the
        /// register.
        pub fn read(
            self: *volatile Self,
        )
    }
}

```

```

) ReadType {
    return @bitCast(self.readDirect());
}

/// The `write` function writes the value of the peripheral register
/// with a value typed according to the `WriteType` of the register.
pub fn write(
    self: *volatile Self,
    value: WriteType,
) void {
    self.writeDirect(@bitCast(value));
}

/// The `readField` function reads the given register and returns the
/// field given by `field_id`.
pub fn readField(
    self: *volatile Self,
    /// An enumeration literal identifying the register field to return.
    comptime field_id: ReadField,
) @FieldType(ReadType, @tagName(field_id)) {
    const reg_value = self.read();
    return @field(reg_value, @tagName(field_id));
}

/// The `updateField` function performs a read - modify - write
/// sequence to change the field given by, `field_id`, to a new value
/// given by, `value`. All other register fields are unaffected.
pub fn updateField(
    self: *volatile Self,
    comptime field_id: WriteField,
    value: @FieldType(WriteType, @tagName(field_id)),
) void {
    // New value is initialized with current register contents.
    var new_value: WriteType = @bitCast(self.read());
    @field(new_value, @tagName(field_id)) = value;
    self.write(new_value);
}

/// The `writeField` function performs a write to set the field given
/// by, `field_id` to the value given by, `value`. **All other fields
/// are written as zero.**
pub fn writeField(
    self: *volatile Self,
    comptime field_id: WriteField,
    value: @FieldType(WriteType, @tagName(field_id)),
) void {
    // New value is initialized to zero.
    var new_value: WriteType = @bitCast(@as(WordRegister, 0));
    @field(new_value, @tagName(field_id)) = value;
    self.write(new_value);
}

```

```

    /// The `updateFields` function performs a read - modify - write
    /// sequence to change the values of all the fields given by the
    /// `field_values` argument. The `field_values` is assumed to be an
    /// anonymous struct literal giving the field names and values which
    /// are to be updated. This function provides a more convenient
    /// interface to update multiple fields in a single operation.
pub fn updateFields(
    self: *volatile Self,
    field_values: anytype,
) void {
    // New value starts with current register contents.
    var new_value: WriteType = @bitCast(self.read());
    inline for (comptime meta.fieldNames(@TypeOf(field_values))) |field_name| {
        @field(new_value, field_name) = @field(field_values, field_name);
    }
    self.write(new_value);
}

    /// The `writeFields` function performs a register write to change the
    /// values of all the fields given by the `field_values` argument. The
    /// `field_values` is assumed to be an anonymous struct literal giving
    /// the field names and values which are to be updated. This function
    /// provides a more convenient interface to write multiple fields in a
    /// single operation. **All fields not included in `field_values` are
    /// written as zero.**
pub fn writeFields(
    self: *volatile Self,
    field_values: anytype,
) void {
    // New value is initialized to zero.
    var new_value: WriteType = @bitCast(@as(WordRegister, 0));
    inline for (comptime meta.fieldNames(@TypeOf(field_values))) |field_name| {
        @field(new_value, field_name) = @field(field_values, field_name);
    }
    self.write(new_value);
}

    /// The `setBitField` function performs a read - modify - write
    /// sequence to write one to the register field given by `field_id`.
    /// The given `field_id` field must be a `u1` type field. All other
    /// fields in the register are unmodified.
pub fn setBitField(
    self: *volatile Self,
    comptime field_id: WriteField,
) void {
    if (comptime @FieldType(WriteType, @tagName(field_id)) != u1) {
        @compileError("field, " ++ @tagName(field_id) ++ ", is not a single bit
field");
    }
    self.updateField(field_id, 1);
}

```

```

    /// The `insertFieldValue` function performs a read - modify - write operation
    /// to update the value of the register field given by `bitfield` to the
    /// value given by `field_value`.
    pub fn insertFieldValue(
        self: *volatile Self,
        bitfield: BitField,
        field_value: WordRegister,
    ) void {
        var new_value = self.readDirect();
        new_value = bitfield.insertField(new_value, field_value);
        self.writeDirect(new_value);
    }

    /// The `extractFieldValue` function returns the value in a register that is
    /// described by the `bitfield` argument.
    pub fn extractFieldValue(
        self: *volatile Self,
        bitfield: BitField,
    ) WordRegister {
        const current_value = self.readDirect();
        return bitfield.extractField(current_value);
    }

    /// The `testBitField` function returns `true` if the register field
    /// given by, `field_id`, is one and `false` otherwise. The given
    /// `field_id` field must be a `ul` type field.
    pub fn testBitField(
        self: *volatile Self,
        comptime field_id: ReadField,
    ) bool {
        if (comptime @FieldType(ReadType, @tagName(field_id)) != u1) {
            @compileError("field, " ++ @tagName(field_id) ++ ", is not a single bit
field");
        }
        return self.readField(field_id) == 1;
    }

    /// The `clearBitField` function performs a read - modify - write
    /// sequence to write zero to the register field given by `field_id`.
    /// The given `field_id` field must be a `ul` type field. All other
    /// fields in the register are unmodified.
    pub fn clearBitField(
        self: *volatile Self,
        comptime field_id: WriteField,
    ) void {
        if (comptime @FieldType(WriteType, @tagName(field_id)) != u1) {
            @compileError("field, " ++ @tagName(field_id) ++ ", is not a single bit
field");
        }
        self.updateField(field_id, 0);
    }

```

```

    /// The `toggleBitField` function performs a read - modify - write
    /// sequence to write the bit complement of the current field value to
    /// the register field given by `field_id`. The given `field_id` field
    /// must be a `ul` type field. All other fields in the register are
    /// unmodified.
    pub fn toggleBitField(
        self: *volatile Self,
        comptime field_id: WriteField,
    ) void {
        if (comptime @FieldType(WriteType, @tagName(field_id)) != ul) {
            @compileError("field, " ++ @tagName(field_id) ++ ", is not a single bit
field");
        }
        const current = self.readField(field_id);
        self.updateField(field_id, ~current);
    }

    /// The `writeBitField` function performs a write operation to set the
    /// register field given by `field_id` to one. The given `field_id`
    /// field must be a `ul` type field. **All other fields in the register
    /// are written as zero.**
    /// See also the `BitRegister` type function for defining registers
    /// consisting entirely of single bit fields.
    pub fn writeBitField(
        self: *volatile Self,
        comptime field_id: WriteField,
    ) void {
        if (comptime @FieldType(WriteType, @tagName(field_id)) != ul) {
            @compileError("field, " ++ @tagName(field_id) ++ ", is not a single bit
field");
        }
        var value: WriteType = @bitCast(@as(WordRegister, 0));
        @field(value, @tagName(field_id)) = 1;
        self.write(value);
    }

    /// The `writeFieldValue` function performs a write operation to set the
    /// register field given by `bitfield` to one.
    /// **All other fields in the register are written as zero.**
    pub fn writeFieldValue(
        self: *volatile Self,
        bitfield: BitField,
        field_value: WordRegister,
    ) void {
        var new_value: WordRegister = 0;
        new_value = bitfield.insertField(new_value, field_value);
        self.writeDirect(new_value);
    }
};

}

```

Indexed Registers

Sometimes hardware peripheral registers are encoded in groups where each group has the same bit structure. In those cases, it is sometimes more convenient to treat the register as a array of packed struct bit fields. However, Zig does not allow arrays in packed structs. A packed struct has a backing integer and defining a structure on top of that backing integer is logically equivalent to defining distinct bit fields at compile time. To treat a packed struct of grouped bits as an array requires some run time operations. The methods of this type perform the required bit shifting and masking necessary to present an array-like interface where each group has a defined Element packed struct and multiple groups fit within the same WordRegister.

```
<<mmio: indexed>>=
/// The `IndexedRegister` type function creates a `packed struct` type along with
/// a set of methods that allow accessing fields within a register through
/// an index value. Each indexed register is composed of an integral number
/// of elements. The `Element` argument gives the type of the bit groups
/// in the register. The minimum number of bits in an `Element` is one
/// and a `WordRegister` must hold an integral number of `Element` type
/// values.
pub fn IndexedRegister(
    /// The type of each element in the register. Note that the
    /// read type and write type are the same.
    comptime Element: type,
) type {
    if (@bitSizeOf(Element) == 0)
        @compileError("Indexed register elements must have non-zero size");
    // Insist that the packing is `close`, i.e. the number of bits in each element
    // is a factor of the number of bits in a `WordRegister`. This restriction could
    // probably be relaxed if the count of elements is also carried around.
    if (@bitSizeOf(WordRegister) % @bitSizeOf(Element) != 0)
        @compileError("Indexed register element size must be a factor of the word size");

    return packed struct {
        const Self = @This();

        const BitMask = DefineBitField(Element);
        const ElementAsInt = meta.Int(.unsigned, @bitSizeOf(Element));
        const ElementIndex = math.Log2Int(WordRegister);
        const element_scale: ElementIndex = @bitSizeOf(Element);
        const element_mask = (@as(WordRegister, 1) << @bitSizeOf(Element)) - 1;

        reg_value: WordRegister,

        /// The `readDirect` function reads the memory location of the register
        /// and returns its value as an uninterpreted integer number. This is a
        /// so called, _raw read_, of the register.
        pub fn readDirect(
            self: *volatile Self,
        ) WordRegister {
            return self.reg_value;
        }
    };
}
```

```

/// The `writeDirect` function writes the memory location of the
/// register and with an uninterpreted integer number. This is a so
/// called, _raw write_, to the register.
pub fn writeDirect(
    self: *volatile Self,
    value: WordRegister,
) void {
    self.reg_value = value;
}

/// The `readIndexedElement` function returns the value of the register
/// element that is present at the `element_index` offset into the register.
pub fn readIndexedElement(
    self: *volatile Self,
    element_index: ElementIndex,
) Element {
    return extractElement(self.ReadDirect(), element_index);
}

/// The `updateIndexedElement` function performs a read - modify - write
/// operation to set the value of the register element at `element_index`
/// to the value given by the `value` argument.
pub fn updateIndexedElement(
    self: *volatile Self,
    element_index: ElementIndex,
    value: Element,
) void {
    const new_value = insertElement(self.readDirect(), element_index, value);
    self.writeDirect(new_value);
}

/// The `writeIndexedElement` function sets the value of the register element
/// at `element_index` to be the value given by the `value` argument.
/// ** All other bits in the register are written as zero**.
pub fn setIndexedElement(
    self: *volatile Self,
    element_index: ElementIndex,
    value: Element,
) void {
    const new_value = insertElement(0, element_index, value);
    self.writeDirect(new_value);
}

fn insertElement(
    base: WordRegister,
    element_index: ElementIndex,
    value: Element,
) WordRegister {
    const element_offset = element_index * element_scale;
    const value_as_bits: ElementAsInt = @bitCast(value);
    return (base & ~(element_mask << element_offset)) | // clear field to zero
        ((value_as_bits & element_mask) << element_offset); // or in new value
}

```

```

    }

    fn extractElement(
        base: WordRegister,
        element_index: ElementIndex,
    ) Element {
        const element_offset = element_index * element_scale;
        return @bitCast((base >> element_offset) & element_mask);
    }
};

}

```

Single Bit Fields

Registers composed entirely of single bit fields are common enough to warrant a specialization of the `IndexedRegister` type to support operating on single bit fields. Here, the index into the register fields is a bit offset number. These register are often used for so called, Write / Clear register where writing a 1 to a specific bit actuates the implied control and writing 0 does nothing. For example, many peripherals use such register to manipulate the sources of interrupt from the peripheral. Other uses require preserving the current bits. For example, a register which represents GPIO values requires the values of the other bits in the register be preserved when a particular bit value is changed.

```

<<mmio: bitregister>>=
/// The `BitRegister` type function creates a `packed struct` type and a set of
/// methods to access a register consisting entirely of single bit fields.
pub fn BitRegister() type {
    return packed struct {
        const Self = @This();

        const BitIndex = math.Log2Int(WordRegister);

        reg_value: WordRegister,

        pub fn readDirect(
            self: *volatile Self,
        ) WordRegister {
            return self.reg_value;
        }

        pub fn writeDirect(
            self: *volatile Self,
            value: WordRegister,
        ) void {
            self.reg_value = value;
        }

        pub fn readBitField(
            self: *volatile Self,
            bit_index: BitIndex,
        ) u1 {
            const bit_mask = bitMask(bit_index);

```

```

        const value = self.readDirect();
        return if ((value & bit_mask) == 0) 0 else 1;
    }

pub fn setBitField(
    self: *volatile Self,
    bit_index: BitIndex,
) void {
    const bit_mask = bitMask(bit_index);
    var new_value = self.readDirect();
    new_value |= bit_mask;
    self.writeDirect(new_value);
}

pub fn clearBitField(
    self: *volatile Self,
    bit_index: BitIndex,
) void {
    const bit_mask = bitMask(bit_index);
    var new_value = self.readDirect();
    new_value &= ~bit_mask;
    self.writeDirect(new_value);
}

pub fn toggleBitField(
    self: *volatile Self,
    bit_index: BitIndex,
) void {
    const bit_mask = bitMask(bit_index);
    var new_value = self.readDirect();
    new_value ^= bit_mask;
    self.writeDirect(new_value);
}

// single bit set, remaining bits 0
pub fn writeBitField(
    self: *volatile Self,
    bit_index: BitIndex,
) void {
    const bit_mask = bitMask(bit_index);
    self.writeDirect(bit_mask);
}

fn bitMask(
    bit_index: BitIndex,
) WordRegister {
    return @as(WordRegister, 1) << bit_index;
}
};

}

```

Code Layout

```
<<mmio.zig>>
<<edit warning>>
<<copyright info>>

///! The `mmio` module contains a set of declarations and functions to aid in
///! accessing memory mapped I/O registers in a fashion that is less dependent
///! upon bit masks and bit offsets. Zig provides `extern struct` and `packed
///! struct` constructs that can be used to define the layout of peripheral
///! device registers in memory and to define the layout of control fields
///! within a register.
///
///! This implementation is oriented to the ARM Cortex-M view of device
///! registers. Although there is nothing Cortex specific, 32-bit registers are
///! the prime focus. Most (but not all) peripheral registers for Cortex SOC's are 32-bit
///! quantities.
///
///! There are many different techniques that hardware designers use to design
///! peripheral control interfaces. Most common is to layout the registers of
///! the device in an proximate group in memory and to layout the controls
///! fields within a register with bit fields that have specific peripheral
///! control or status meanings. Much of this design borrows heavily from the
///! ideas described
///! [here](https://www.scattered-thoughts.net/writing/mmio-in-zig/). This
///! design extends those ideas to handle other register field arrangements.
///
///! This module contains three primary type functions that are used to
///! define the layout of memory mapped peripherals:
///
///! 1. The `Register` function defines types for register containing
///!     named bit fields.
///! 2. The `IndexedRegister` function defines types that contain a sequence
///!     of bit fields in the register and allows for indexed access to the
///!     register fields.
///! 3. The `BitRegister` function defines register types where all the fields
///!     are a single bit. This is a degenerate case of `IndexedRegister` and
///!     allows the register to be treated as an array of bits.

const std = @import("std");
const meta = std.meta;
const math = std.math;
const testing = std.testing;

<<mmio: foundations>>
<<mmio: bitfields>>
<<mmio: register>>
<<mmio: indexed>>
<<mmio: bitregister>>
```

```

test "mmio register functions" {
    const DrRegisterRead = packed struct(WordRegister) {
        data: u8,
        fedata: u1,
        pedata: u1,
        bedata: u1,
        oedata: u1,
        _reserved: u20,
    };
    const DrRegisterWrite = packed struct(WordRegister) {
        data: u8,
        _reserved: u24,
    };
    const UartDr = Register(DrRegisterRead, DrRegisterWrite);

    const LcrhRegister = packed struct(WordRegister) {
        brk: u1,
        pen: u1,
        eps: u1,
        stp: u1,
        fen: u1,
        wlen: enum(u2) {
            char_is_5_bits = 0,
            char_is_6_bits = 1,
            char_is_7_bits = 2,
            char_is_8_bits = 3,
        },
        sps: u1,
        _reserved: u24,
    };
    const UartLcrh = Register(LcrhRegister, LcrhRegister);

    const UartPeripheral = extern struct {
        dr: UartDr,
        lcrh: UartLcrh,
    };

    const uart_reg_count = 0x48 / @sizeOf(WordRegister);
    var fake_peripheral: [uart_reg_count]WordRegister = [_]WordRegister{0} **
uart_reg_count;
    fake_peripheral[1] = 3 << @bitOffsetOf(LcrhRegister, "wlen");

    const uart_0 = @as(*volatile UartPeripheral, @ptrCast(&fake_peripheral));

    const lcrh = uart_0.lcrh.readDirect();
    try testing.expectEqual(@as(WordRegister, 3 << 5), lcrh);

    uart_0.lcrh.writeDirect(42);
    try testing.expectEqual(@as(WordRegister, 42), uart_0.lcrh.readDirect());
    uart_0.lcrh.writeDirect(0); // clear things out

    var lcrh_value = uart_0.lcrh.read();
}

```

```

try testing.expectEqual(@as(u2, 0), @intFromEnum(lcrh_value.wlen));

lcrh_value.wlen = .char_is_8_bits;
uart_0.lcrh.write(lcrh_value);
try testing.expectEqual(@as(u2, 3), @intFromEnum(lcrh_value.wlen));

uart_0.lcrh.updateFields(.{
    .wlen = .char_is_6_bits,
});
const wlen_value = uart_0.lcrh.readField(.wlen);
try testing.expectEqual(.char_is_6_bits, wlen_value);

uart_0.lcrh.updateFields(.{
    .brk = 1,
    .fen = 1,
});
try testing.expectEqual(@as(u1, 1), uart_0.lcrh.readField(.brk));
try testing.expectEqual(@as(u1, 1), uart_0.lcrh.readField(.fen));

uart_0.lcrh.setBitField(.eps);
try testing.expectEqual(@as(u1, 1), uart_0.lcrh.readField(.eps));

uart_0.lcrh.clearBitField(.eps);
try testing.expectEqual(@as(u1, 0), uart_0.lcrh.readField(.eps));

uart_0.lcrh.toggleBitField(.eps);
try testing.expectEqual(@as(u1, 1), uart_0.lcrh.readField(.eps));

// start with all zero bits
uart_0.lcrh.writeDirect(0);
uart_0.lcrh.writeBitField(.eps);
try testing.expectEqual(@as(u1, 1), uart_0.lcrh.readField(.eps));

uart_0.lcrh.writeDirect(0);
uart_0.lcrh.writeBitField(.eps);
try testing.expectEqual(@as(u1, 1), uart_0.lcrh.readField(.eps));
uart_0.lcrh.writeBitField(.pen);
try testing.expectEqual(@as(u1, 1), uart_0.lcrh.readField(.pen));
uart_0.lcrh.writeBitField(.stp);
try testing.expectEqual(@as(u1, 1), uart_0.lcrh.readField(.stp));
}

```

Cortex v7e-M Core Access

In this chapter, we define a set of mechanisms to access the ARM® Cortex-M4 processor core directly from Zig. The intent is to provide the information similar to CMSIS-core but to use Zig coding conventions. CMSIS provides symbolic encoding of the large quantity of information and constants that are required to access core processor facilities. This encoding falls into three broad categories:

1. Access to Cortex v7e-M specific instructions.
2. Access to Cortex v7e-M specific registers and the bit fields in them.
3. Defining the address layout for processor peripherals.
4. Defining definitions of the layout for processor peripherals.
5. Support for handling exception priorities and the vector table.

Cortex specific instructions

Some Cortex specific instructions are not generated by compilers. These tend to be those instructions which access control registers or support specific architecture dependent operations. Here, we follow the pattern of CMSIS in the choice of instructions to provide. Note this is little more than syntactic sugar to avoid sprinkling inline assembly through the code.

```
<<cortex_7m: instructions>>=
/// The following function provide access to specific Thumb 2 instructions
/// needed for fine level control of the processor core. These instructions
/// don't perform normal data access and processing and so are not usually
/// generated by a compiler.

pub inline fn nop() void {
    asm volatile ("nop");
}

pub inline fn wfi() void {
    asm volatile ("wfi");
}

pub inline fn wfe() void {
    asm volatile ("wfe");
}

pub inline fn sev() void {
    asm volatile ("sev");
}

pub inline fn isb() void {
    asm volatile ("isb sy" ::: .{.memory = true});
}

pub inline fn dsb() void {
    asm volatile ("dsb sy" ::: .{.memory = true});
}
```

```

pub inline fn dmb() void {
    asm volatile ("dmb sy" ::: .{.memory = true});
}

pub inline fn rev(
    value: WordRegister,
) WordRegister {
    return asm volatile ("rev %[ret], %[value]"
        : [ret] "=r" (-> WordRegister),
        : [value] "r" (value),
    );
}

pub inline fn rev16(
    value: HalfWordRegister,
) HalfWordRegister {
    return asm volatile ("rev16 %[ret], %[value]"
        : [ret] "=r" (-> HalfWordRegister),
        : [value] "r" (value),
    );
}

pub inline fn revsh(
    value: i16,
) i32 {
    return asm volatile ("revsh %[ret], %[value]"
        : [ret] "=r" (-> i16),
        : [value] "r" (value),
    );
}

pub inline fn rbit(
    value: WordRegister,
) WordRegister {
    return asm volatile ("rbit %[ret], %[value]"
        : [ret] "=r" (-> WordRegister),
        : [value] "r" (value),
    );
}

pub inline fn ror(
    value: WordRegister,
    rot: WordRegister,
) WordRegister {
    return asm volatile ("ror %[ret], %[value], %[rot]"
        : [ret] "=r" (-> WordRegister),
        : [value] "r" (value),
        : [rot] "r" (rot),
    );
}

```

```

pub inline fn clrex() void {
    asm volatile("clrex" :: : .{.memory = true});
}

pub inline fn ldrexb(
    ptr: *const ByteRegister,
) ByteRegister {
    return asm volatile ("ldrexb %[ret], [%[ptr]]"
        : [ret] "=r" (-> ByteRegister),
        : [ptr] "r" (ptr),
    );
}

pub inline fn strexb(
    value: ByteRegister,
    ptr: *ByteRegister,
) void {
    asm volatile ("strexb %[value], [%[ptr]]"
        :
        : [value] "r" (value),
        [ptr] "r" (ptr),
        : {.memory = true}
    );
}

pub inline fn ldrexb(
    ptr: *const HalfWordRegister,
) HalfWordRegister {
    return asm volatile ("ldrexb %[ret], [%[ptr]]"
        : [ret] "=r" (-> HalfWordRegister),
        : [ptr] "r" (ptr),
    );
}

pub inline fn strexb(
    value: HalfWordRegister,
    ptr: *HalfWordRegister,
) void {
    asm volatile ("strexb %[value], [%[ptr]]"
        :
        : [value] "r" (value),
        [ptr] "r" (ptr),
        : {.memory = true}
    );
}

pub inline fn ldrex(
    ptr: *const WordRegister,
) WordRegister {
    return asm volatile ("ldrex %[ret], [%[ptr]]"
        : [ret] "=r" (-> WordRegister),
        : [ptr] "r" (ptr),
    );
}

```

```

    );
}

pub inline fn strex(
    value: WordRegister,
    ptr: *WordRegister,
) void {
    asm volatile ("strex %[value], [%[ptr]]"
        :
        : [value] "r" (value),
          [ptr] "r" (ptr),
        : {.memory = true}
    );
}

pub inline fn ldrbt(
    ptr: *const ByteRegister,
) ByteRegister {
    return asm volatile ("ldrbt %[ret], [%[ptr]]"
        : [ret] "=r" (-> ByteRegister),
        : [ptr] "r" (ptr),
    );
}

pub inline fn strbt(
    value: ByteRegister,
    ptr: *ByteRegister,
) void {
    asm volatile ("strbt %[value], [%[ptr]]"
        :
        : [value] "r" (value),
          [ptr] "r" (ptr),
        : {.memory = true}
    );
}

pub inline fn ldrht(
    ptr: *const HalfWordRegister,
) HalfWordRegister {
    return asm volatile ("ldrht %[ret], [%[ptr]]"
        : [ret] "=r" (-> HalfWordRegister),
        : [ptr] "r" (ptr),
    );
}

pub inline fn strht(
    value: HalfWordRegister,
    ptr: *HalfWordRegister,
) void {
    asm volatile ("strht %[value], [%[ptr]]"
        :
        : [value] "r" (value),

```

```

        : [ptr] "r" (ptr),
        : {.memory = true}
    );
}

pub inline fn ldrt(
    ptr: *const WordRegister,
) WordRegister {
    return asm volatile ("ldrt %[ret], [%[ptr]]"
        : [ret] "=r" (-> WordRegister),
        : [ptr] "r" (ptr),
    );
}

pub inline fn strt(
    value: WordRegister,
    ptr: *WordRegister,
) void {
    asm volatile ("strt %[value], [%[ptr]]"
        :
        : [value] "r" (value),
        : [ptr] "r" (ptr),
        : {.memory = true}
    );
}

// The core provides specialized instructions for setting and
// clearing the PRIMASK and FAULTMASK registers.

pub inline fn enableIrq() void {
    asm volatile ("cpsie i" :: .{.memory = true});
}

pub inline fn disableIrq() void {
    asm volatile ("cpsid i" :: .{.memory = true});
}

pub inline fn enableFaultIrq() void {
    asm volatile ("cpsie f" :: .{.memory = true});
}

pub inline fn disableFaultIrq() void {
    asm volatile ("cpsid f" :: .{.memory = true});
}

```

Core registers

In addition to the general purpose registers, the Cortex v7e-M architecture has a set of core registers. The design strategy is to define a namespace for each register and a set of functions to get or set the register. For those registers that contain bit fields, access to the individual fields is provided.

```

<<cortex_7m: core registers>>=
// MSP - Master Stack Pointer
pub const msp = struct {
    pub inline fn get() WordRegister {
        return asm volatile ("mrs %[reg], msp"
            : [reg] "=r" (-> WordRegister),
        );
    }
    pub inline fn set(
        value: WordRegister,
    ) void {
        asm volatile ("msr msp, %[value]"
            :
            : [value] "r" (value),
        );
    }
};

// PSP - Process Stack Pointer
pub const psp = struct {
    pub inline fn get() WordRegister {
        return asm volatile ("mrs %[reg], psp"
            : [reg] "=r" (-> WordRegister),
        );
    }
    pub inline fn set(
        value: WordRegister,
    ) void {
        asm volatile ("msr psp, %[value]"
            :
            : [value] "r" (value),
        );
    }
};

// Program Status register
pub const psr = struct {
    pub const apsr = packed struct(WordRegister) {
        _reserved: u27,
        q: u1,
        v: u1,
        c: u1,
        z: u1,
        n: u1,
    }

    pub inline fn get() apsr {
        return asm volatile ("mrs %[reg], apsr"
            : [reg] "=r" (-> apsr),
        );
    }

    pub inline fn set(

```

```

        value: apsr,
    ) void {
        asm volatile ("msr aspr, %[value]"
                    :
                    : [value] "=r" (value),
                    );
    }
};

pub const ge = packed struct(WordRegister) {
    _reserved1: u16,
    ge: u4,
    _reserved2: u12,

    pub inline fn get() ge {
        return asm volatile ("mrs %[reg], apsr_g"
                            : [reg] "=r" (-> ge),
                            );
    }

    pub inline fn set(
        value: ge,
    ) void {
        asm volatile ("msr aspr_g, %[value]"
                    :
                    : [value] "=r" (value),
                    );
    }
};

pub const apsr_ge = packed struct(WordRegister) {
    _reserved1: u16,
    ge: u4,
    _reserved2: u7,
    q: u1,
    v: u1,
    c: u1,
    z: u1,
    n: u1,

    pub inline fn get() apsr_ge {
        return asm volatile ("mrs %[reg], apsr_nzcvqq"
                            : [reg] "=r" (-> apsr_ge),
                            );
    }

    pub inline fn set(
        value: apsr_ge,
    ) void {
        asm volatile ("msr aspr_nzcvqq, %[value]"
                    :
                    : [value] "=r" (value),
                    );
    }
};

```

```

        );
    }
};

pub const ipsr = packed struct(WordRegister) {
    isr_number: u9,
    _reserved: u23,

    pub inline fn get() ipsr {
        return asm volatile ("mrs %[reg], ipsr"
            : [reg] "=r" (-> ipsr),
        );
    }

    pub fn format(
        value: ipsr,
        writer: *std.Io.Writer,
    ) std.Io.Writer.Error!void {
        try writer.print("ipsr: {s} ({d})", .{
            irq_names[value.isr_number],
            value.isr_number,
        });
    }
}

const reserved_name = "Reserved";
const irq_names = [_][]const u8{
    "none", // 0
    "Reset", // 1
    "NMI", // 2
    "HardFault", // 3
    "MemManage", // 4
    "BusFault", // 5
    "UsageFault", // 6
    reserved_name, // 7
    reserved_name, // 8
    reserved_name, // 9
    reserved_name, // 10
    "SVCall", // 11
    "Reserved_Debug", // 12
    reserved_name, // 13
    "PendSV", // 14
    "SysTick", // 15
    "BROWNOUT", // 16
    "WDT", // 17
    "RTC", // 18
    "VCOMP", // 19
    "IOSLAVE", // 20
    "IOSLAVEACC", // 21
    "IOMSTR0", // 22
    "IOMSTR1", // 23
    "IOMSTR2", // 24
    "IOMSTR3", // 25
}

```

```

    "IOMSTR4", // 26
    "IOMSTR5", // 27
    "BLE", // 28
    "GPIO", // 29
    "CTIMER", // 30
    "UART0", // 31
    "UART1", // 32
    "SCARD", // 33
    "ADC", // 34
    "PDM", // 35
    "MSPI0", // 36
    reserved_name, // 37
    "STIMER", // 38
    "STIMER_CMPO", // 39
    "STIMER_CMPR1", // 40
    "STIMER_CMPR2", // 41
    "STIMER_CMPR3", // 42
    "STIMER_CMPR4", // 43
    "STIMER_CMPR5", // 44
    "STIMER_CMPR6", // 45
    "STIMER_CMPR7", // 46
    "CLKGEN", // 47
};

};

pub const epsr = packed struct(WordRegister) {
    _reserved1: u10,
    ici_it1: u6,
    _reserved2: u8,
    t: u1,
    ici_it2: u2,
    _reserved3: u5,

    pub inline fn get() epsr {
        return asm volatile ("mrs %[reg], epsr"
            : [reg] "=r" (-> epsr),
        );
    }
};

pub const iepsr = packed struct(WordRegister) {
    isr_number: u9,
    _reserved1: u1,
    ici_it1: u6,
    _reserved2: u8,
    t: u1,
    ici_it2: u2,
    _reserved3: u5,

    pub inline fn get() iepsr {
        return asm volatile ("mrs %[reg], iepsr"
            : [reg] "=r" (-> iepsr),
    }
};

```

```

        );
    }
};

pub const iapsr = packed struct(WordRegister) {
    isr_number: u9,
    _reserved: u18,
    q: u1,
    v: u1,
    c: u1,
    z: u1,
    n: u1,

    pub inline fn get() iapsr {
        return asm volatile ("mrs %[reg], iapsr"
            : [reg] "=r" (-> iapsr),
        );
    }
};

pub const eapsr = packed struct(WordRegister) {
    _reserved1: u10,
    ici_it1: u6,
    _reserved2: u8,
    t: u1,
    ici_it2: u2,
    q: u1,
    v: u1,
    c: u1,
    z: u1,
    n: u1,

    pub inline fn get() eapsr {
        return asm volatile ("mrs %[reg], eapsr"
            : [reg] "=r" (-> eapsr),
        );
    }
};

pub const xpsr = packed struct(WordRegister) {
    // Iapsr part
    isr_number: u9,
    // Epsr part
    _reserved1: u1,
    ici_it1: u6,
    _reserved2: u8,
    t: u1,
    ici_it2: u2,
    // Apsr part
    q: u1,
    v: u1,
    c: u1,
}

```

```

z: u1,
n: u1,

pub inline fn get() xpsr {
    return asm volatile ("mrs %[reg], psr"
        : [reg] "=r" (-> xpsr),
    );
}

pub fn format(
    value: xpsr,
    writer: *std.Io.Writer,
) std.Io.Writer.Error!void {
    const q: u8 = if (value.q == 1) 'Q' else 'q';
    const v: u8 = if (value.v == 1) 'V' else 'v';
    const c: u8 = if (value.c == 1) 'C' else 'c';
    const z: u8 = if (value.z == 1) 'Z' else 'z';
    const n: u8 = if (value.n == 1) 'N' else 'n';
    try writer.writeAll("xpsr: apsr: ");
    try writer.print("{c}", .{q});
    try writer.print("{c}", .{v});
    try writer.print("{c}", .{c});
    try writer.print("{c}", .{z});
    try writer.print("{c}", .{n});
    try writer.print(" ipsr: {d}", .{value.isr_number});
}
};

// PRIMASK -- Priority Mask Register
pub const primask = packed struct(WordRegister) {
    mask: u1,
    _reserved: u31 = 0,

    pub inline fn get() u1 {
        const value = asm volatile ("mrs %[reg], primask"
            : [reg] "=r" (-> primask),
        );
        return value.mask;
    }

    pub inline fn set(
        mask: u1,
    ) void {
        const value: primask = .{ .mask = mask };
        asm volatile ("msr primask, %[value]"
            :
            : [value] "r" (value),
        );
    }
}

/// A critical section executes with those exceptions that have a configurable

```

```

/// priority disabled.
pub const CriticalSection = struct {
    mask: u1,

    pub fn enter() CriticalSection {
        const result = get();
        set(1);
        return .{ .mask = result };
    }

    pub fn leave(
        self: CriticalSection,
    ) void {
        set(self.mask);
    }
};

// FAULTMASK -- Fault Mask Register
pub const faultmask = packed struct(WordRegister) {
    mask: u1,
    _reserved: u31 = 0,

    pub inline fn get() u1 {
        const value = asm volatile ("mrs %[reg], faultmask"
            : [reg] "=r" (-> faultmask),
        );
        return value.mask;
    }

    pub inline fn set(
        mask: u1,
    ) void {
        const value: faultmask = .{ .mask = mask };
        asm volatile ("msr faultmask, %[value]"
            :
            : [value] "r" (value),
        );
    }
};

/// A fault section executes with all exceptions except NMI disabled.
pub const FaultSection = struct {
    mask: u1,

    pub fn enter() FaultSection {
        const result = get();
        set(1);
        return .{ .mask = result };
    }

    pub fn leave(
        self: FaultSection,
    ) void {

```

```

        set(self.mask);
    }
};

// BASEPRI -- Base Priority Mask Register
pub const basepri = packed struct(WordRegister) {
    priority: ExceptionPriority,
    _reserved: u24 = 0,

    pub inline fn get() ExceptionPriority {
        const value = asm volatile ("mrs %[reg], basepri"
            : [reg] "=r" (-> basepri),
        );
        return value.priority;
    }

    pub inline fn set(
        priority: ExceptionPriority,
    ) void {
        const value: basepri = .{ .priority = priority };
        asm volatile ("msr basepri, %[value]"
            :
            : [value] "r" (value),
        );
    }

    pub inline fn setMax(
        priority: ExceptionPriority,
    ) void {
        const value: basepri = .{ .priority = priority };
        asm volatile ("msr basepri_max, %[value]"
            :
            : [value] "r" (value),
        );
    }

    /// A priority section executes with the base priority raised to the
    /// given level. The priority is returned to the previous value upon
    /// leaving the section.
    pub const PrioritySection = struct {
        current: ExceptionPriority,

        pub fn enter(
            encoded_priority: ExceptionPriority,
        ) PrioritySection {
            const section: PrioritySection = .{ .current = get() };
            set(encoded_priority);
            return section;
        }

        pub fn leave(

```

```

        self: PrioritySection,
    ) void {
    set(self.current);
}
};

// CONTROL -- Control register
pub const control = packed struct(WordRegister) {
    const ExecPriv = enum(u1) {
        privileged = 0,
        unprivileged = 1,
    };
    const SpSelect = enum(u1) {
        msp = 0,
        psp = 1,
    };

    npriv: ExecPriv,
    spsel: SpSelect,
    fpca: u1,
    _reserved: u29,

    pub inline fn get() control {
        return asm volatile ("mrs %[reg], control"
            : [reg] "=r" (-> control),
        );
    }

    pub inline fn set(
        value: control,
    ) void {
        asm volatile ("msr control, %[value]"
            :
            : [value] "r" (value),
        );
        isb();
    }

    pub inline fn update(
        values: anytype,
    ) void {
        // New value starts with current register contents.
        var new_value = get();
        inline for (comptime meta.fieldNames(@TypeOf(values))) |field_name| {
            @field(new_value, field_name) = @field(values, field_name);
        }
        set(new_value);
    }
};

// FPSCR -- Floating Point Status and Control Register

```

```

pub const fpSCR = packed struct(WordRegister) {
    ioc: u1,
    dzc: u1,
    ofc: u1,
    ufc: u1,
    ixc: u1,
    _reserved1: u2,
    idc: u1,
    _reserved2: u14,
    rmode: enum(u2) {
        rnd_to_nearest,
        rnd_twrD_plus_inf,
        rnd_twrD_neg_inf,
        rnd_twrD_zero,
    } = .rnd_to_nearest,
    fz: u1,
    dn: u1,
    ap: u1,
    _reserved3: u1,
    v: u1,
    c: u1,
    z: u1,
    n: u1,
}

pub inline fn get() fpSCR {
    return asm volatile ("vmrs %[reg], fpSCR"
        : [reg] "=r" (-> fpSCR),
    );
}

pub inline fn set(
    value: fpSCR,
) void {
    asm volatile ("vmsr fpSCR, %[value]"
        :
        : [value] "r" (value),
    );
}

pub fn update(
    values: anytype,
) void {
    // New value starts with current register contents.
    var new_value = get();
    inline for (comptime meta.fieldNames(@TypeOf(values))) |field_name| {
        @field(new_value, field_name) = @field(values, field_name);
    }
    set(new_value);
}
};

```

Cortex Exceptions

The number of bits of exception priority and the structure of the exception frame are major factors in exception model.

```
<<cortex_7m: exceptions>>=
/// The SOC designer determines the number of priority bits that are supported.
/// The minimum is 3, and this is the most common. We use a configuration option
/// to fix the number of bits.
pub const nvic_priority_bits: u8 = config.priority_bits;
pub const system_exception_count: u16 = 16;
pub const IrqNumber = u8;

pub const ExceptionHandler = *const fn () callconv(.c) void;

/// The memory layout of an exception stack frame for a Cortex-M4 is captured by
/// the following structure definition.
/// There are two items to note here:
/// . If the FPU is not being used, there is no possibility
///   that the stack frame contains the saved floating point registers.
/// . Even if the FPU is used, the exception entry
///   may _not_ have pushed the registers.
/// There are control settings in the `FPCCR` register which determine
/// the stacking of FPU registers during exception entry.

fn ExceptionFrameReg() type {
    if (config.fpu_context == .none) {
        return extern struct { r0: u32, r1: u32, r2: u32, r3: u32, ip: u32, lr: u32, pc:
u32, xpsr: psr.xpsr };
    } else {
        // zig fmt: off
        return extern struct { r0: u32, r1: u32, r2: u32, r3: u32, ip: u32, lr: u32, pc:
u32, xpsr: psr.xpsr,
            s0: f32, s1: f32, s2: f32, s3: f32, s4: f32, s5: f32, s6: f32, s7: f32, s8:
f32,
            s9: f32, s10: f32, s11: f32, s12: f32, s13: f32, s14: f32, s15: f32, fpscr: u32
};
        // zig fmt: on
    }
}

pub const ExceptionFrame = ExceptionFrameReg();

/// Unlike CMSIS, system exception numbers are kept separate from IRQ Numbers.
/// There are only 16 system exceptions and not all have configurable priorities.
pub const SystemException = enum(u4) {
    reset = 1,
    nmi,
    hard_fault,
    mem_manage,
    bus_fault,
    usage_fault,
    sv_call = 11,
}
```

```

    debug_monitor,
    pend_sv = 14,
    sys_tick = 15,
    _, // In case one of the reserved exception numbers slips into the processing.
};

pub const ExceptionPriority = u8;
/// Priority grouping data type. Used to set / get and encode exception priorities.
pub const PriorityGroup = u3;
/// Priority values must be shifted to the upper most bits. This is the number of bits
/// that must be shifted.
pub const priority_shift: u3 = @bitSizeOf(ExceptionPriority) - nvic_priority_bits;

/// The `encodePriority` priority function returns an encoded exception priority number
/// that is suitable to use as an argument to sbc.setExceptionPriority or
nvic.setIrqPriority.

/// The `priority_group` number determines the number of priorities and sub-priorities
/// used the priority scheme configuration. The `preempt_priority` and `sub_priority`
/// arguments determine the priority configuration.

pub fn encodePriority(
    priority_group: PriorityGroup,
    preempt_priority: ExceptionPriority,
    sub_priority: ExceptionPriority,
) ExceptionPriority {
    const priority_offset = @as(ExceptionPriority, 7) - priority_group;
    const preempt_bits: PriorityGroup = @min(priority_offset, nvic_priority_bits);
    const preempt_mask: ExceptionPriority = (@as(ExceptionPriority, 1) << preempt_bits) -
1;

    const sub_priority_offset = @as(ExceptionPriority, priority_group) +
nvic_priority_bits;
    const sub_priority_bits: PriorityGroup = if (sub_priority_offset < @as
(ExceptionPriority, 7))
        0
    else
        @intCast((@as(ExceptionPriority, priority_group) + nvic_priority_bits) - @as
(ExceptionPriority, 7));
    const sub_priority_mask: ExceptionPriority = (@as(ExceptionPriority, 1) <<
sub_priority_bits) -% 1;

    return ((preempt_priority & preempt_mask) << sub_priority_bits) |
        (sub_priority & sub_priority_mask);
}

/// The `decodePriority` function returns the preemption priority and sub_priority for
/// the `priority` argument value within the `priority_group` group. This function inverts
/// the calculation of `encodePriority`. The return value is a 2-tuple of the
/// exception priority and sub-priority, in that order.

pub fn decodePriority(
    priority: ExceptionPriority,
    priority_group: PriorityGroup,
) struct { ExceptionPriority, ExceptionPriority } {

```

```

const priority_offset = @as(PriorityGroup, 7) - priority_group;
const preempt_bits: PriorityGroup = @min(priority_offset, nvic_priority_bits);
const preempt_mask: ExceptionPriority = (@as(ExceptionPriority, 1) << preempt_bits) -
1;

const sub_priority_offset = @as(ExceptionPriority, priority_group) +
nvic_priority_bits;
const sub_priority_bits: PriorityGroup = if (sub_priority_offset < @as
(ExceptionPriority, 7))
    0
else
    (priority_group % 7) + nvic_priority_bits;
const sub_priority_mask: ExceptionPriority = (@as(ExceptionPriority, 1) <<
sub_priority_bits) % 1;

return .{
    .preempt_priority = (priority >> sub_priority_bits) & preempt_mask,
    .sub_priority = priority & sub_priority_mask,
};

}

```

Cortex core peripherals

This section covers the set of core peripheral registers that are used to control system function, handle interrupts, memory management, and other functions within the core.

```

<<cortex_7m: private peripheral bus addresses>>
pub const scs_base: u32 = 0xe000e000; // System Control Space
pub const itm_base: u32 = 0xe0000000; // Instruction Trace Macrocell
pub const dwt_base: u32 = 0xe0001000; // Debug Watchpoint and Trace
pub const tpiu_base: u32 = 0xe0040000; // Trace Port Interface Unit
pub const dcb_base: u32 = 0xe000edf0; // Debug Control Block
pub const sys_tick_base: u32 = scs_base + 0x0010; // System Tick
pub const nvic_base: u32 = scs_base + 0x0100; // Nested Vectored Interrupt Controller
pub const scb_base: u32 = scs_base + 0x0d00; // System Control Block
pub const sc_n_scb_base: u32 = scs_base; // System Controls not in the SBC
pub const mpu_base: u32 = scs_base + 0xd90; // Memory Protection Unit
pub const fpu_base: u32 = scs_base + 0xf30; // Floating Point Unit
pub const rom_table_base: u32 = 0xe00ff000; // ROM Table

```

```

<<cortex_7m: peripheral access>>
/// All core peripherals have a single instance of the corresponding block.
pub const scb = @as(*volatile SystemControlBlock, @ptrToInt(scb_base));
pub const nvic = @as(*volatile NestedVectoredInterruptController, @ptrToInt(nvic_base));
pub const scnscb = @as(*volatile SystemControlsNotInSbcBlock, @ptrToInt(sc_n_scb_base));
pub const systick = @as(*volatile SysTick, @ptrToInt(sys_tick_base));
pub const dcb = @as(*volatile DebugControlBlock, @ptrToInt(dcb_base));
pub const dwt = @as(*volatile DataWatchpointTrace, @ptrToInt(dwt_base));
pub const mpu = @as(*volatile MemoryProtectionUnit, @ptrToInt(mpu_base));
pub const fpu = @as(*volatile FloatingPointUnit, @ptrToInt(fpu_base));

```

```

<<cortex_7m: peripheral access>>=
pub const SystemControlBlock = extern struct {
    cpuid: Register(CpuidReg, NoAccessRegister),
    icsr: Register(IcsrReadReg, IcsrWriteReg),
    vtor: Register(VtorReg, VtorReg),
    aircr: Register(AircrReadReg, AircrWriteReg),
    scr: Register(ScrReg, ScrReg),
    ccr: Register(CcrReg, CcrReg),
    shrp: [12]u8,
    shcsr: Register(ShcsrReg, ShcsrReg),
    cfsr: Register(CfsrReg, CfsrReg),
    hfsr: Register(HfsrReg, HfsrReg),
    dfsr: Register(DfsrReg, DfsrReg),
    mmfar: Register(WordRegister, WordRegister),
    bfar: Register(WordRegister, WordRegister),
    afsr: Register(WordRegister, WordRegister),
    _reserved1: [18]WordRegister,
    cpacr: Register(CpacrReg, CpacrReg),
    _reserved2: [93]WordRegister,
    stir: Register(StirReg, StirReg),

    /// Reset, NMI, and Hard Fault do not have configurable priorities. The
    /// initial SP is conceptually exception 0 (i.e. takes up a slot in the vector
    /// table). So, there are only 12 remaining and any exception number needs to
    /// be offset by the non-configurable exceptions to be used as an index into
    /// the configurable priority registers.
    const config_exception_offset: u4 = 4;

    // CPUID Base Register
    pub const CpuidReg = packed struct(WordRegister) {
        revision: u4,
        part_no: u12,
        constant: u4,
        variant: u4,
        implementer: u8,
    };

    // ICSR -- Interrupt Control and State Register
    pub const IcsrReadReg = packed struct(WordRegister) {
        vect_active: u9,
        _reserved1: u2,
        ret_to_base: u1,
        vect_pending: u10,
        isr_pending: u1,
        reserved_for_debug_use: u1,
        _reserved2: u1,
        _reserved3: u1, // PENDSTCLR is write only
        pend_st_set: u1,
        _reserved4: u1, // PENDSVCLR is write only
        pend_sv_set: u1,
        _reserved5: u2,
        nmi_pend_set: u1,
    };
}

```

```

};

pub const IcsrWriteReg = packed struct(WordRegister) {
    _reserved1: u9 = 0, // VECTACTIVE is read only
    _reserved2: u2 = 0,
    _reserved3: u1 = 0, // RETTTOBASE is read only
    _reserved4: u10 = 0, // VECTPENDING is read only
    _reserved5: u1 = 0, // ISR PENDING is read only
    _reserved6: u1 = 0, // Reserved for Debug use is read only
    _reserved7: u1 = 0,
    pend_st_clr: u1,
    pend_st_set: u1,
    pend_sv_clr: u1,
    pend_sv_set: u1,
    _reserved8: u2 = 0,
    nmi_pend_set: u1,
};

// VTOR -- Vector Table Offset Register
const VtorReg = packed struct(WordRegister) {
    pub const TblBase = enum(u1) {
        code = 0,
        sram = 1,
    };

    _reserved1: u7 = 0,
    tbl_off: u22,
    tbl_base: TblBase,
    _reserved2: u2 = 0,
};

// AIRCR -- Application Interrupt and Reset Control Register
pub const aircr_key: u16 = 0x05fa;
pub const aircr_key_status: u16 = 0x05fa;
pub const AircrReadReg = packed struct(WordRegister) {
    _reserved1: u8,
    pri_group: PriorityGroup,
    _reserved2: u4,
    endianness: u1,
    vect_key_stat: u16,
};

pub const AircrWriteReg = packed struct(WordRegister) {
    vect_reset: u1,
    vect_clr_active: u1,
    sys_reset_req: u1,
    _reserved1: u5 = 0,
    pri_group: PriorityGroup,
    _reserved2: u5 = 0,
    vect_key: u16,
};

// SCR -- System Control Register

```

```

pub const ScrReg = packed struct(WordRegister) {
    _reserved1: u1 = 0,
    sleep_on_exit: u1,
    sleep_deep: u1,
    _reserved2: u1 = 0,
    sev_on_pend: u1,
    _reserved3: u27 = 0,
};

// CCR -- Configuration and Control Register
pub const CcrReg = packed struct(WordRegister) {
    non_base_thrd_ena: u1,
    user_set_mpend: u1,
    _reserved1: u1 = 0,
    unalign_trp: u1,
    div_0_trp: u1,
    _reserved2: u3 = 0,
    bf_hf_nm_ign: u1,
    stk_align: u1,
    _reserved3: u22 = 0,
};

// SHCSR -- System Handler Control and State Register
pub const ShcsrReg = packed struct(WordRegister) {
    mem_fault_act: u1,
    bus_fault_act: u1,
    _reserved1: u1 = 0,
    usg_fault_act: u1,
    _reserved2: u3 = 0,
    svcall_act: u1,
    monitor_act: u1,
    _reserved3: u1 = 0,
    pendsv_act: u1,
    systick_act: u1,
    usg_fault_pended: u1,
    mem_fault_pended: u1,
    bus_fault_pended: u1,
    svcall_fault_pended: u1,
    mem_fault_ena: u1,
    bus_fault_ena: u1,
    usg_fault_ena: u1,
    _reserved4: u13 = 0,
};

// CFSR -- Configurable Fault Status Register
// composed of three sub-registers
// MMFSR - Memory Management Fault Status Register
pub const MmfsrReg = packed struct(ByteRegister) {
    iacc_viol: u1,
    dacc_viol: u1,
    _reserved1: u1 = 0,
    m_unstk_err: u1,
}

```

```

m_stk_err: u1,
m_lsp_err: u1,
_reserved2: u1 = 0,
mm_ar_valid: u1,

pub fn format(
    value: MmfsrReg,
    writer: *std.Io.Writer,
) std.Io.Writer.Error!void {
    try writer.print("mmfsr:", .{});
    const reg: ByteRegister = @bitCast(value);
    if (reg == 0) {
        try writer.writeAll(" -");
    } else {
        if (value.iacc_viol != 0) try writer.print(" iaccviol,", .{});
        if (value.dacc_viol != 0) try writer.print(" daccviol,", .{});
        if (value.m_unstk_err != 0) try writer.print(" munstkkerr,", .{});
        if (value.m_stk_err != 0) try writer.print(" mstkkerr,", .{});
        if (value.m_lsp_err != 0) try writer.print(" mlsperr,", .{});
        if (value.mm_ar_valid != 0) try writer.print(" mmarvalid,", .{});
    }
}
};

// BFSR -- Bus Fault Status Register
pub const BfsrReg = packed struct(ByteRegister) {
    ibus_err: u1,
    precis_err: u1,
    imprecis_err: u1,
    unstk_err: u1,
    stk_err: u1,
    lsp_err: u1,
    _reserved: u1 = 0,
    bf_ar_valid: u1,

    pub fn format(
        value: BfsrReg,
        writer: *std.Io.Writer,
    ) std.Io.Writer.Error!void {
        try writer.print("bfsr:", .{});
        const reg: ByteRegister = @bitCast(value);
        if (reg == 0) {
            try writer.writeAll(" -");
        } else {
            if (value.ibus_err != 0) try writer.print(" ibuserr,", .{});
            if (value.precis_err != 0) try writer.print(" preciserr,", .{});
            if (value.imprecis_err != 0) try writer.print(" impreciserr,", .{});
            if (value.unstk_err != 0) try writer.print(" unstkkerr,", .{});
            if (value.stk_err != 0) try writer.print(" stkkerr,", .{});
            if (value.lsp_err != 0) try writer.print(" lsperr,", .{});
            if (value.bf_ar_valid != 0) try writer.print(" bfarvalid,", .{});
        }
    }
}

```

```

};

// UFSR - Usage Fault Status Register
pub const UfsrReg = packed struct(HalfWordRegister) {
    undef_instr: u1,
    inv_state: u1,
    inv_pc: u1,
    no_cp: u1,
    _reserved1: u4 = 0,
    unaligned: u1,
    div_by_zero: u1,
    _reserved2: u6 = 0,

    pub fn format(
        value: UfsrReg,
        writer: *std.Io.Writer,
    ) std.Io.Writer.Error!void {
        try writer.print("ufsr:", .{});
        const reg: HalfWordRegister = @bitCast(value);
        if (reg == 0) {
            try writer.writeAll(" - ");
        } else {
            if (value.undef_instr != 0) try writer.print(" undefinstr,", .{});
            if (value.inv_state != 0) try writer.print(" invstate,", .{});
            if (value.inv_pc != 0) try writer.print(" invpc,", .{});
            if (value.no_cp != 0) try writer.print(" nocp,", .{});
            if (value.unaligned != 0) try writer.print(" unaligned,", .{});
            if (value.div_by_zero != 0) try writer.print(" divbyzero,", .{});
        }
    }
};

pub const CfsrReg = packed struct(WordRegister) {
    mmfsr: MmfsrReg,
    bfsr: BfsrReg,
    ufsr: UfsrReg,

    pub fn format(
        value: CfsrReg,
        writer: *std.Io.Writer,
    ) std.Io.Writer.Error!void {
        try writer.print("{f}\n", .{value.mmfsr});
        try writer.print("{f}\n", .{value.bfsr});
        try writer.print("{f}", .{value.ufsr});
    }
};

// HFSR -- Hard Fault Status Register
pub const HfsrReg = packed struct(WordRegister) {
    _reserved1: u1 = 0,
    vect_tbl: u1,
    _reserved2: u28 = 0,
    forced: u1,
    debug_evt: u1,
}

```

```

pub fn format(
    value: HfsrReg,
    writer: *std.Io.Writer,
) std.Io.Writer.Error!void {
    try writer.print("hfsr:", .{});
    const reg: WordRegister = @bitCast(value);
    if (reg == 0) {
        try writer.writeAll(" -");
    } else {
        if (value.vect_tbl != 0) try writer.print(" vecttbl,", .{});
        if (value.forced != 0) try writer.print(" forced,", .{});
        if (value.debug_evt != 0) try writer.print(" debugevt,", .{});
    }
}
};

// DFSR -- Debug Fault Status Register
pub const DfsrReg = packed struct(WordRegister) {
    halted: u1,
    bkpt: u1,
    dwt_trap: u1,
    vcatch: u1,
    external: u1,
    _reserved: u27 = 0,

    pub fn format(
        value: DfsrReg,
        writer: *std.Io.Writer,
    ) std.Io.Writer.Error!void {
        try writer.print("dfsr:", .{});
        const reg: WordRegister = @bitCast(value);
        if (reg == 0) {
            try writer.writeAll(" -");
        } else {
            if (value.halted != 0) try writer.print(" halted,", .{});
            if (value.bkpt != 0) try writer.print(" bkpt,", .{});
            if (value.dwt_trap != 0) try writer.print(" dwttrap,", .{});
            if (value.vcatch != 0) try writer.print(" vcatch,", .{});
            if (value.external != 0) try writer.print(" external,", .{});
        }
    }
};
};

// CPACR -- Coprocessor Access Control Register
pub const CpacrReg = packed struct(WordRegister) {
    _reserved1: u20 = 0,
    cp10: FpAccess,
    cp11: FpAccess,
    _reserved2: u8 = 0,

    pub const FpAccess = enum(u2) {

```

```

        none = 0,
        priv = 1,
        full = 3,
    };
};

// STIR -- Software Triggered Interrupt Register
pub const StirReg = packed struct(WordRegister) {
    int_id: u9,
    _reserved: u23 = 0,
};

pub fn getPriorityGrouping(
    self: *volatile SystemControlBlock,
) PriorityGroup {
    return self.aircr.readField(.pri_group);
}

pub fn setPriorityGrouping(
    self: *volatile SystemControlBlock,
    priority_group: PriorityGroup,
) void {
    var value: AircrWriteReg = @bitCast(self.aircr.read());
    value.pri_group = priority_group;
    value.vect_key = aircr_key;
    self.aircr.write(value);
}

// set system exception priority
pub fn setExceptionPriority(
    self: *volatile SystemControlBlock,
    excn: SystemException,
    priority: ExceptionPriority,
) void {
    const priority_reg_index = @intFromEnum(excn) - config_exception_offset;
    const priority_value = priority << priority_shift;
    self.shrp[priority_reg_index] = priority_value;
}

// returns encoded priority
pub fn getExceptionPriority(
    self: *volatile SystemControlBlock,
    excn: SystemException,
) ExceptionPriority {
    const priority_reg_index = @intFromEnum(excn) - config_exception_offset;
    const priority_value = self.shrp[priority_reg_index];
    return priority_value >> priority_shift;
}

/// Note in CMSIS this is reckoned as an NVIC function. The control is
/// is actually part of the SBC and so we keep it there.
pub fn systemReset()

```

```

        self: *volatile SystemControlBlock,
) noreturn {
    dsb();
    self.aircr.updateFields(.{ .sys_reset_req = 1, .vect_key = aircr_key });
    dsb();

    while (true) { // Wait while the core applies the reset.
        nop();
    }
}

pub fn enableFpu(
    self: *volatile SystemControlBlock,
) void {
    self.cpacr.writeFields(.{
        .cp10 = .full,
        .cp11 = .full,
    });
    dsb();
    isb();
}
};

test {
    testing.refAllDeclsRecursive(SystemControlBlock);
}

```

```

<<cortex_7m: peripheral access>>=
pub const NestedVectoredInterruptController = extern struct {
    iser: [8]WordRegister,
    _reserved1: [24]WordRegister,
    icer: [8]WordRegister,
    _reserved2: [24]WordRegister,
    ispr: [8]WordRegister,
    _reserved3: [24]WordRegister,
    icpr: [8]WordRegister,
    _reserved4: [24]WordRegister,
    iabr: [8]WordRegister,
    _reserved5: [56]WordRegister,
    ipr: [240]u8,
    _reserved6: [644]WordRegister,
    stir: Register(SystemControlBlock.StirReg, SystemControlBlock.StirReg),

    pub fn enableIrq(
        self: *volatile NestedVectoredInterruptController,
        irqn: IrqNumber,
    ) void {
        const index = irqn / @bitSizeOf(WordRegister);
        const offset: u5 = @intCast(irqn % @bitSizeOf(WordRegister));
        const enable = @as(WordRegister, 1) << offset;
        self.iser[index] = enable;
    }
}

```

```

pub fn getEnableIrq(
    self: *volatile NestedVectoredInterruptController,
    irqn: IrqNumber,
) bool {
    const index = irqn / @bitSizeOf(WordRegister);
    const offset: u5 = @intCast(irqn % @bitSizeOf(WordRegister));
    const position = @as(WordRegister, 1) << offset;
    const enable = self.iser[index];
    return (enable & position) != 0;
}

pub fn disableIrq(
    self: *volatile NestedVectoredInterruptController,
    irqn: IrqNumber,
) void {
    const index = irqn / @bitSizeOf(WordRegister);
    const offset: u5 = @intCast(irqn % @bitSizeOf(WordRegister));
    const disable = @as(WordRegister, 1) << offset;
    self.icer[index] = disable;
}

pub fn getPendingIrq(
    self: *volatile NestedVectoredInterruptController,
    irqn: IrqNumber,
) bool {
    const index = irqn / @bitSizeOf(WordRegister);
    const offset: u5 = @intCast(irqn % @bitSizeOf(WordRegister));
    const position = @as(WordRegister, 1) << offset;
    const pending = self.ispr[index].read();
    return (pending & position) != 0;
}

pub fn setPendingIrq(
    self: *volatile NestedVectoredInterruptController,
    irqn: IrqNumber,
) void {
    const index = irqn / @bitSizeOf(WordRegister);
    const offset: u5 = @intCast(irqn % @bitSizeOf(WordRegister));
    const pending = @as(WordRegister, 1) << offset;
    self.ispr[index] = pending;
}

pub fn clearPendingIrq(
    self: *volatile NestedVectoredInterruptController,
    irqn: IrqNumber,
) void {
    const index = irqn / @bitSizeOf(WordRegister);
    const offset: u5 = @intCast(irqn % @bitSizeOf(WordRegister));
    const pending = @as(WordRegister, 1) << offset;
    self.icpr[index] = pending;
}

```

```

pub fn getActive(
    self: *volatile NestedVectoredInterruptController,
    irqn: IrqNumber,
) bool {
    const index = irqn / @bitSizeOf(WordRegister);
    const offset: u5 = @intCast(irqn % @bitSizeOf(WordRegister));
    const position = @as(WordRegister, 1) << offset;
    const active = self.iabrv[index].read();
    return (active & position) != 0;
}

pub fn setIrqPriority(
    self: *volatile NestedVectoredInterruptController,
    irqn: IrqNumber,
    priority: ExceptionPriority,
) void {
    const priority_value = priority << priority_shift;
    self.ipr[irqn] = priority_value;
}

// returns encoded priority
pub fn getIrqPriority(
    self: *volatile NestedVectoredInterruptController,
    irqn: IrqNumber,
) u8 {
    const priority_value = self.ipr[irqn];
    return priority_value >> priority_shift;
}
};

/// An interrupt section executes with a particular interrupt disabled
/// and restores the previous state of the interrupt upon leaving the
/// section.
pub const InterruptSection = struct {
    irq: IrqNumber,
    enable: bool,

    pub fn enter(
        irqn: IrqNumber,
    ) InterruptSection {
        const section: InterruptSection = .{
            .irq = irqn,
            .enable = nvic.getEnableIrq(irqn),
        };
        nvic.disableIrq(irqn);
        return section;
    }

    pub fn leave(
        self: *const InterruptSection,
    ) void {

```

```

        if (self.enable) nvic.enableIrq(self.irq) else nvic.disableIrq(self.irq);
    }
};

test {
    testing.refAllDeclsRecursive(NestedVectoredInterruptController);
}

```

```

<<cortex_7m: peripheral access>>=
const SystemControlsNotInSbcBlock = extern struct {
    _reserved0: WordRegister,
    ictr: Register(IctrReg, IctrReg),
    actlr: Register(ActrlReg, ActrlReg),

    // ICTR -- Interrupt Controller Type Register
    const IctrReg = packed struct(WordRegister) {
        int_lines_num: u4,
        _reserved: u28 = 0,
    };

    // ACTLR -- Auxiliary Control Register
    const ActrlReg = packed struct(WordRegister) {
        dis_mcyc_int: u1,
        dis_def_wbuf: u1,
        dis_fold: u1,
        _reserved: u5 = 0,
        dis_fPCA: u1,
        dis_oofp: u1,
    };
};

test {
    testing.refAllDeclsRecursive(SystemControlsNotInSbcBlock);
}

```

```

<<cortex_7m: peripheral access>>=
pub const SysTick = extern struct {
    csr: Register(CsrReg, CsrReg),
    rvr: Register(SysTickCountReg, SysTickCountReg),
    cvr: Register(SysTickCountReg, SysTickCountReg),
    calib: Register(CalibReg, CalibReg),

    const CsrReg = packed struct(WordRegister) {
        enable: u1,
        tick_int: u1,
        clk_source: u1,
        _reserved1: u13 = 0,
        count_flag: u1,
        _reserved2: u15 = 0,
    };
};

```

```

const SysTickValueReg = packed struct(WordRegister) {
    value: u24,
    _reserved: u8 = 0,
};

const CalibReg = packed struct(WordRegister) {
    ten_ms: u24,
    _reserved: u6 = 0,
    skew: u1,
    noref: u1,
};
};

test {
    testing.refAllDeclsRecursive(SysTick);
}

```

```

<<cortex_7m: peripheral access>>=
pub const DebugControlBlock = extern struct {
    dhcsr: Register(DhcsrReadReg, DhcsrWriteReg),
    dcrsr: Register(DcrsrReg, DcrsrReg), // write only
    dcrdr: Register(WordRegister, WordRegister),
    demcr: Register(DemcrReg, DemcrReg),

    pub const dbg_key: u16 = 0xa05f;
    pub const DhcsrReadReg = packed struct(WordRegister) {
        c_debug_en: u1,
        c_halt: u1,
        c_step: u1,
        c_mask_ints: u1,
        _reserved1: u1,
        c_snap_stall: u1,
        _reserved2: u10,
        s_reg_rdy: u1,
        s_halt: u1,
        s_sleep: u1,
        s_lockup: u1,
        _reserved3: u4,
        s_retire_st: u1,
        s_reset_st: u1,
        _reserved4: u6,
    };

    pub const DhcsrWriteReg = packed struct(WordRegister) {
        c_debug_en: u1,
        c_halt: u1,
        c_step: u1,
        c_mask_ints: u1,
        _reserved1: u1 = 0,
        c_snap_stall: u1,
        _reserved2: u10 = 0,
        dbg_key: u16 = dbg_key,
    };
}

```

```

};

pub const DcrrsrReg = packed struct(WordRegister) {
    // zig fmt: off
    pub const RegSelect = enum(u7) {
        r0 = 0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, sp, lr,
        debug_return_address,
        xpsr, msp, psp, special,
        fpSCR = 33,
        s0 = 64, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, s14, s15,
        s16, s17, s18, s19, s20, s21, s22, s23, s24, s25, s26, s27, s28, s29, s30, s31
    };
    // zig fmt: on

    reg_sel: RegSelect,
    _reserved1: u9 = 0,
    reg_w_n_r: u1,
    _reserved2: u15 = 0,
};

const DemcrReg = packed struct(WordRegister) {
    vc_core_reset: u1,
    _reserved1: u3 = 0,
    vc_mm_err: u1,
    vc_noCP_err: u1,
    vc_chk_err: u1,
    vc_stat_err: u1,
    vc_bus_err: u1,
    vc_int_err: u1,
    vc_hard_err: u1,
    _reserved2: u5 = 0,
    mon_en: u1,
    mon_pend: u1,
    mon_step: u1,
    mon_req: u1,
    _reserved3: u4 = 0,
    trc_ena: u1,
    _reserved4: u7 = 0,
};
};

test {
    testing.refAllDeclsRecursive(DebugControlBlock);
}

```

```

<<cortex_7m: peripheral access>>=
pub const DataWatchpointTrace = extern struct {
    dwt_ctrl: Register(DwtCtrlRegRead, DwtCtrlRegWrite),
    dwt_cyc_cnt: WordRegister,
    // TODO: complete the remaining Data Watchpoint and Trace registers.
    // This is the minimal definition to support cycle counting.

```

```

const DwtCtrlRegRead = packed struct(WordRegister) {
    cyc_cnt_ena: u1,
    post_preset: u4,
    post_init: u4,
    cyc_tap: u1,
    sync_tap: u2,
    pc_sampl_ena: u1,
    _reserved1: u3,
    exc_trc_ena: u1,
    cpi_evt_ena: u1,
    exc_evt_ena: u1,
    sleep_evt_ena: u1,
    lsu_evt_ena: u1,
    fold_evt_ena: u1,
    cyc_evt_ena: u1,
    _reserved2: u1,
    no_prf_cnt: u1,
    no_cyc_cnt: u1,
    no_ext_trig: u1,
    no_trc_pkt: u1,
    num_comp: u4,
};

const DwtCtrlRegWrite = packed struct(WordRegister) {
    cyc_cnt_ena: u1,
    post_preset: u4,
    post_init: u4,
    cyc_tap: u1,
    sync_tap: u2,
    pc_sampl_ena: u1,
    _reserved1: u3,
    exc_trc_ena: u1,
    cpi_evt_ena: u1,
    exc_evt_ena: u1,
    sleep_evt_ena: u1,
    lsu_evt_ena: u1,
    fold_evt_ena: u1,
    cyc_evt_ena: u1,
    _reserved2: u9 = 0,
};
};

test {
    testing.refAllDeclsRecursive(DataWatchpointTrace);
}

```

```

<<cortex_7m: peripheral access>>=
pub const MemoryProtectionUnit = extern struct {
    type: Register(TypeReg, NoAccessRegister),
    ctrl: Register(CtrlReg, CtrlReg),
    rnr: Register(RnrReg, RnrReg),

```

```

rbar: Register(RbarReg, RbarReg),
rasr: Register(RasrReg, RasrReg),
rbar_a1: Register(RbarReg, RbarReg), // Alias registers
rasr_a1: Register(RasrReg, RasrReg),
rbar_a2: Register(RbarReg, RbarReg),
rasr_a2: Register(RasrReg, RasrReg),
rbar_a3: Register(RbarReg, RbarReg),
rasr_a3: Register(RasrReg, RasrReg),

// zig fmt: off
pub const RegionSize = enum(u5) {
    size_32b = 4, size_64b, size_128b, size_256b, size_512b,
    size_1kb, size_2kb, size_4kb, size_8kb, size_16kb, size_32kb,
    size_64kb, size_128kb, size_256kb, size_512kb,
    size_1mb, size_2mb, size_4mb, size_8mb, size_16mb, size_32mb,
    size_64mb, size_128mb, size_256mb, size_512mb,
    size_1gb, size_2gb, size_4gb,
};

// zig fmt: on

pub const TypeReg = packed struct(WordRegister) {
    separate: u1,
    _reserved1: u7 = 0,
    dregion: u8,
    iregion: u8,
    _reserved2: u8 = 0,
};

pub const CtrlReg = packed struct(WordRegister) {
    enable: u1,
    hf_nmi_ena: u1,
    priv_def_ena: u1,
    _reserved: u29 = 0,
};

pub const RnrReg = packed struct(WordRegister) {
    region: u4,
    _reserved: u28 = 0,
};

pub const RbarReg = packed struct(WordRegister) {
    region: u4,
    valid: u1,
    addr: u27, // The actual size of this field depends upon the region size.
};

pub const RasrReg = packed struct(WordRegister) {
    pub const AccessPriv = enum(u3) {
        none = 0,
        priv,
        uro,
        full,
    }
};

```

```

        pro = 5,
        ro,
    };

    enable: u1,
    size: RegionSize, // Region size in bytes = 2 ^ (.size + 1). Minimum value is 3.
    _reserved1: u2 = 0,
    srd: u8,
    b: u1,
    c: u1,
    s: u1,
    tex: u3,
    _reserved2: u2 = 0,
    ap: AccessPriv,
    _reserved3: u1 = 0,
    xn: u1,
    _reserved4: u3 = 0,
};

pub fn enable(
    self: *volatile MemoryProtectionUnit,
    value: CtrlReg,
) void {
    dmb();
    scb.shcsr.setBitField(.mem_fault_ena);
    self.ctrl.write(value);
    dsb();
    isb();
}

pub fn disable(
    self: *volatile MemoryProtectionUnit,
) void {
    dmb();
    scb.shcsr.clearBitField(.mem_fault_ena);
    self.ctrl.clearBitField(.enable);
    dsb();
    isb();
}

pub fn clrRegion(
    self: *volatile MemoryProtectionUnit,
    region: RegionSize,
) void {
    self.rnr.writeFields(.{ .region = region });
    self.rasr.writeDirect(0);
}

pub fn setRegion(
    self: *volatile MemoryProtectionUnit,
    rnr: RnrReg,
    rbar: RbarReg,
)

```

```

        rasr: RasrReg,
    ) void {
        self.rnr.write(rnr);
        self.rbar.write(rbar);
        self.rasr.write(rasr);
    }

    pub const RegionDesc = struct {
        addr: RbarReg,
        attrs: RasrReg,
    };

    pub fn load(
        self: *volatile MemoryProtectionUnit,
        regions: []const RegionDesc,
    ) void {
        for (regions, 0..) |region, region_number| {
            self.setRegion(@bitCast(@as(u32, region_number)), region.addr, region.attrs);
        }
    }
};

```

```

<<cortex_7m: peripheral access>>=
pub const FloatingPointUnit = extern struct {
    _reserved: WordRegister,
    fpCCR: Register(FpCCRReg, FpCCRReg),
    fpCAR: WordRegister,
    fpDSCR: Register(FpDSCRReg, FpDSCRReg),

    pub const FpCCRReg = packed struct(WordRegister) {
        lSpAct: u1,
        user: u1,
        _reserved1: u1 = 0,
        thread: u1,
        hFrDY: u1,
        mmrDY: u1,
        bFrDY: u1,
        _reserved2: u1 = 0,
        monRdy: u1,
        _reserve3: u21 = 0,
        lSpEN: u1,
        aSpEN: u1,
    };

    pub const FpDSCRReg = packed struct(WordRegister) {
        _reserve1: u22 = 0,
        rMode: u2,
        fz: u1,
        dn: u1,
        aHP: u1,
        _reserved2: u5 = 0,
    };
};

```

```

pub const FpuStackContext = enum {
    no_stack,
    always_stack,
    lazy_stack,
};

pub fn setFpuStackContext(
    self: *volatile FloatingPointUnit,
    comptime context: FpuStackContext,
) void {
    const context_fields = switch (context) {
        .no_stack => .{ .aspen = 0, .lspen = 0 },
        .always_stack => .{ .aspen = 1, .lspen = 0 },
        .lazy_stack => .{ .aspen = 1, .lspen = 1 },
    };
    self.fpccr.updateFields(context_fields);
}
};

```

Code layout

```

<<cortex_7m.zig>>=
const std = @import("std");
const math = std.math;
const meta = std.meta;
const fmt = std.fmt;
const testing = std.testing;

const mmio = @import("mmio");
const Register = mmio.Register;
const WordRegister = mmio.WordRegister;
const HalfWordRegister = mmio.HalfWordRegister;
const ByteRegister = mmio.ByteRegister;
const NoAccessRegister = mmio.NoAccess.RegisterType;

const config = @import("config"); // to obtain command line options

<<cortex_7m: instructions>>
<<cortex_7m: core registers>>
<<cortex_7m: exceptions>>
<<cortex_7m: private peripheral bus addresses>>
<<cortex_7m: peripheral access>>

```

Bipartite Buffer

A [bipartite buffer](#) is a circular queue implemented in an array with the characteristic that the allocated space is contiguous. Typically, circular queues implemented in arrays must handle the modulus arithmetic to wrap around from the end of an array to its beginning. The bipartite buffer makes contiguous allocations by splitting the array into two parts when there is insufficient space at the end of the array to satisfy a request for space.

In this section, we develop a set of functions to implement a bipartite buffer. This implementation is a direct transliteration of the ``C++'' implementation from the above reference. It has been converted to Zig and some of the pointer manipulations have been replaced by Zig slices. Otherwise, the logic and interface is the same. There are several implementations of this idea available. The rust implementation is [lock-free](#). See [also](#).

Requirements

The specific requirements for our use case are:

- Data is accessed in a FIFO manner.
- No attempt is made to insure that the code is safe under preemption. Users must take what precautions are necessary for the context in which the buffer is used.
- Producers operate the buffer as follows:
 - A portion of the buffer is reserved to obtain the required space.
 - Data can then be written to the reserved area.
 - Afterwards, the new data is made available by the `commit` operation.
- Consumers operate the buffer as follows:
 - The `getContiguousBlock` function is used to request a size to read.
 - The data is read out of the block.
 - The `decommitBlock` function then releases the space back for reuse. Note one may decommit less than the size of the obtained contiguous block, including zero.
- Producers must strictly follow the reserve, write, commit protocol.
- Consumers must strictly follow the contiguous block, read, decommit block protocol.

Bipartite Buffer Operations

```
<<BipBuffer: imports>>=
const std = @import("std");
```

```
<<BipBuffer: type function>>=
pub fn BipBuffer(
    comptime ElemType: type,
) type {
    return struct {
        const Self = @This();

        <<BipBuffer: fields>>
```

```
<<BipBuffer: public functions>>
};

}
```

```
<<BipBuffer: fields>>=
/// A slice pointing to the underlying storage.
buffer: []ElemType,
/// The index into `buffer` for the A side.
ixa: usize,
/// The size data committed to the A side.
sza: usize,
/// The index into `buffer` for the B side.
ixb: usize,
/// The size data committed to the b side.
szb: usize,
/// The index into `buffer` where a pending reservation has been made.
ix_resrv: usize,
/// The number of bytes of a pending reservation.
sz_resrv: usize,
```

```
<<BipBuffer: public functions>>=
pub fn init(
    buf: []ElemType,
) Self {
    return .{
        .buffer = buf,
        .ixa = 0,
        .sza = 0,
        .ixb = 0,
        .szb = 0,
        .ix_resrv = 0,
        .sz_resrv = 0,
    };
}
```

```
<<BipBuffer: public functions>>=
/// The `reserve` function sets aside `size` continuous bytes of space in the bip buffer.
/// The return value is a slice pointing to the underlying storage that has been reserved.
/// The length of the returned slice may be less than `size` and may be zero.
/// A returned slice length of zero implies the buffer is full.
pub fn reserve(
    self: *Self,
    size: usize,
) []ElemType {
    // We always allocate on B if B exists; this means we have two blocks and our buffer is
    // filling.
    if (self.szb != 0) {
        const b_free_space = self.ix_a - self.ix_b - self.szb;
        const available = @min(b_free_space, size);
```

```

        self.sz_resrv = available;
        self.ix_resrv = self.ixb + self.szb;
        return self.buffer[self.ix_resrv..self.ix_resrv + self.sz_resrv];
    } else {
        // Block b does not exist, so we can check if the space AFTER a is bigger than the
        // space
        // before A, and allocate the bigger one.
        const free_space_after_a = self.buffer.len - self.ix_a - self.sza;
        if (free_space_after_a >= self.ix_a) {
            const available = @min(free_space_after_a, size);

            self.sz_resrv = available;
            self.ix_resrv = self.ix_a + self.sza;
            return self.buffer[self.ix_resrv .. self.ix_resrv + self.sz_resrv];
        } else {
            const available = @min(self.ix_a, size);
            self.sz_resrv = available;
            self.ix_resrv = 0;
            return self.buffer[0..self.sz_resrv];
        }
    }
}

```

```

<<BipBuffer: public functions>>=
/// The `commit` function makes available the data written to a previously
/// reserved block. The number of bytes committed is given by the `size` argument.
/// If `size` is greater than the reserved size, an assert is triggered in
/// Debug and ReleaseSafe builds. If `size` is less than the reserved size,
/// only `size` bytes are committed and the remainder of the reserved space is
/// released back to the buffer. If `size` is zero, the reservation is released.
pub fn commit(
    self: *Self,
    size: usize,
) void {
    if (size == 0) {
        self.sz_resrv = 0;
        self.ix_resrv = 0;
        return;
    }

    std.debug.assert(size <= self.sz_resrv);
    const commit_size = @min(size, self.sz_resrv);

    if (self.sza == 0 and self.szb == 0) {
        self.ix_a = self.ix_resrv;
        self.sza = commit_size;
        self.ix_resrv = 0;
        self.sz_resrv = 0;
        return;
    }

    if (self.ix_resrv == self.sza + self.ix_a) {

```

```

        self.sza += commit_size;
    } else {
        self.szb += commit_size;
    }

    self.ix_resrv = 0;
    self.sz_resrv = 0;
}

```

```

<<BipBuffer: public functions>>=
/// The `getContiguousBlock` function returns a slice pointing to the largest
/// contiguous section of committed data in the buffer.
/// If the returned slice length is zero, then the buffer is empty.
pub fn getContiguousBlock(
    self: Self,
) []ElemType {
    return self.buffer[self.ix_a .. self.ix_a + self.sza];
}

```

```

<<BipBuffer: public functions>>=
/// The `decommitBlock` returns `size` bytes of data in the first contiguous block
/// back to the buffer. If the value of `size` is greater than the length of
/// the first contiguous block, it is silently truncated to be that length.
pub fn decommitBlock(
    self: *Self,
    size: usize,
) void {
    if (size >= self.sza) {
        self.ix_a = self.ix_b;
        self.sza = self.szb;
        self.ix_b = 0;
        self.szb = 0;
    } else {
        self.sza -= size;
        self.ix_a += size;
    }
}

```

```

<<BipBuffer: public functions>>=
/// The `clear` function discards all the data and reservations in the bip buffer.
/// The underlying storage for the buffer is not affected.
pub fn clear(
    self: *Self,
) void {
    self.ix_a = 0;
    self.sza = 0;
    self.ix_b = 0;
    self.szb = 0;
    self.ix_resrv = 0;
    self.sz_resrv = 0;
}

```

```
}
```

```
<<BiBuffer: public functions>>
/// The `getCommittedSize` function returns the number of bytes that have
/// been committed in the queue, i.e. the number of bytes that can be read out.
pub fn getCommittedSize(
    self: *const Self,
) usize {
    return self.sza + self.szb;
}
```

Testing

```
<<BiBuffer: tests>>
const testing = std.testing;

test {
    testing.refAllDecls(BiBuffer(u8));
}
```

```
<<BiBuffer: tests>>
test BiBuffer {
    // This test shows how to initialize a bi buffer and perform a
    // simple reserve/commit and getContiguousBlock/decommitBlock
    // sequence.
    var buffer: [100]u8 = undefined;
    var bip = BiBuffer(u8).init(&buffer);

    const res1 = bip.reserve(50);
    try testing.expectEqual(@as(usize, 50), res1.len);
    @memset(res1, @as(u8, 'A'));
    bip.commit(50);

    const blk1 = bip.getContiguousBlock();
    try testing.expectEqual(@as(usize, 50), blk1.len);
    try testing.expectEqual(@as(u8, 'A'), blk1[0]);
    bip.decommitBlock(50);
}
```

```
<<BiBuffer: tests>>
test "buffer boundaries" {
    // Check conditions on the boundaries of the buffer.
    var buffer: [100]u8 = undefined;
    var bip = BiBuffer(u8).init(&buffer);

    // Reserve 100 bytes.
    var res = bip.reserve(100);
    try testing.expectEqual(@as(usize, 100), res.len);
```

```

// Write and commit 50 bytes.
@memset(res[0..50], @as(u8, 'A'));
bip.commit(50);

// Check that we can see 50 bytes.
const block = bip.getContiguousBlock();
try testing.expectEqual(@as(usize, 50), block.len);
try testing.expectEqual(@as(u8, 'A'), block[0]);
bip.decommitBlock(50);

// After decommitting the previous 50 we can now get 100.
res = bip.reserve(100);
}

```

```

<<BipBuffer: tests>>=
test "overflow" {
    var buffer: [100]u8 = undefined;
    var bip = BipBuffer(u8).init(&buffer);

    const blk1 = bip.reserve(101);
    try testing.expectEqual(@as(usize, 100), blk1.len);
}

```

Bipartite Buffer Code Layout

```

<<BipBuffer.zig>>=
<<edit warning>>
<<copyright info>>

///! The original copyright notice on the code is:
///! Copyright (c) 2003 Simon Cooke, All Rights Reserved
///! Licensed royalty-free for commercial and non-commercial
///! use, without warranty or guarantee of suitability for any purpose.
///! All that I ask is that you send me an email
///! telling me that you're using my code. It'll make me
///! feel warm and fuzzy inside. spectecjr@gmail.com

///! This module contains an implementation of a Bipartite Buffer as described
///! [here](https://www.codeproject.com/Articles/3479/The-Bip-Buffer-The-Circular-Buffer-
///! with-a-Twist).
///! This implementation is re-coded from the original C++ implementation and follows
///! the interfaces and code closely (even to the short field names). Some of the original
///! commentary in the code is also retained. Most of the changes in the transliteration
///! are the result of using Zig slices instead of multi-item pointers.

<<BipBuffer: imports>>

<<BipBuffer: type function>>

<<BipBuffer: tests>>

```

Formatted Output of Relation Tuples and Values

When constructing Relation Value expressions from the algebraic operators, it is not unusual to use multiple operations to obtain the desired result. In general, the algebraic operators create new relation values with different headings. It is all but essential to be able to visualize the intermediate steps and the final result to verify the expression produced what is intended. This section shows a mechanism to obtain formatted output for Relation Tuples and Relation Values.

To work within the Zig standard library scheme for formatted output that is contained in the standard library, a data type may contain a `format` function that is invoked instead of the default implementation. This gives us a chance to present a tabular rendering as the output. As we have emphasized before, because there is no inherent order of tuples or attributes, the tabular representation of a Relation Tuple or Value is not unique. Any permutation of the tuple or attribute order of a tabular representation is an equivalent representation of a Relation Value. However, the visualization of a relation value or tuple is still extraordinarily useful. It's just important not to give any unwarranted importance to the order of the formatted output. The implementation reserves the right to change it.

The implementation given here does not have any options for how the attribute values are output. The output is generated as a rectangular table with cells containing the attribute name, data type and value. Scalar values are formatted by their default implementation. The main complication of this implementation is caused by having tuple-valued and relation-valued attributes. Tuple and relation values are treated as “sub-tables” in the tabular layout. This gives us a form of nesting and complicates the layout calculations.

Design Concepts

We have defined both relation tuples and relation values as consisting of a **heading** and a **body**. That distinction is used here to divide the formatting into two parts. Fundamentally, the implementation must create a mapping of the heading and body onto a rectangular arrangement of rows and columns that reflects any nested structure of the output. A cell is placed at the intersection of each row and column. To accommodate differing sizes of the output values, cells may span rows or columns as necessary. The cell may hold either an attribute name, attribute data type, attribute value, or a sub-table which represents a tuple-valued or relation-valued attribute.

For a simple tuple value, the output might appear as (ignoring comments on the right here and following):

```
+-----+ <-+
|RelTuple          |   |
+---+---+-----+   |
|len  |weight |name    |   | Heading
+---+---+-----+   |
|usize|f32    |[]const u8|   |
+---+---+-----+ <-+
|32    |5.0e+00|foobar  |   | Body
+---+---+-----+ <-+
```

If the previous tuple is wrapped to create a tuple-valued attribute named, `properties`, then the output might appear as:

```
+-----+ <-----+-----+
```

```

|RelTuple           |
+-----+-----+
|name      |properties   |
+-----+-----+ <--+
|[]const u8|RelTuple    |
|          +---+---+ | Sub-heading
|          |len  |weight | | for tuple-valued
|          +---+---+ | attribute
|          |usize|f32   |
+-----+-----+ <-----+
|foobar    |32    |5.0e+00| | Body
+-----+-----+ <--+
^     ^     ^
|     |     |
+-----+-----+
  Scalar    Nested tuple
  value      value

```

The grouping of the tuple-valued attribute is managed by considering its type information as a sub-heading. In this example, the `properties` attribute has a `RelTuple` type, which is contained in a sub-heading.

The body of the relation tuple is considered to have two value elements. The `foobar` value is a simple scalar value of the `name` attribute. The other value is treated as a nested value corresponding to the two attributes of `properties`.

Relation values follow the same pattern. Consider the previous relation tuple example as a relation value. We expand the population to include three items named, `foo`, and three named `bar`. The output might appear as:

```

+-----+-----+ <--+
|RelValue           |
+-----+-----+
|len  |weight |name      | | Heading
+-----+-----+-----+
|usize|f32   |[]const u8|
+-----+-----+ <--+
|0    |5.0e+00|foo      |
+-----+-----+-----+
|1    |6.0e+00|foo      |
+-----+-----+-----+
|2    |7.0e+00|foo      | | Body
+-----+-----+-----+
|0    |5.0e+00|bar      |
+-----+-----+-----+
|1    |6.0e+00|bar      |
+-----+-----+ <--+

```

Whereas the relation tuple has a single row to show its single set of attribute values, a relation value has a row for each tuple. In this example, the three attribute values of the tuples are simple scalar values.

If the previous relation value is grouped to create a relation-valued attribute named, `properties`, then the output might appear as:

<-----+ RelValue +-----+-----+ name properties +-----+-----+ <---+ Heading		
[]const u8 RelValue		
	+-----+ len weight +-----+	Sub-heading
	+-----+ usize f32 +-----+	
foo	0 5.0e+00 +-----+ 1 6.0e+00 +-----+ Tuple 2 7.0e+00 +-----+ value Body	
bar	0 5.0e+00 +-----+ Tuple 1 6.0e+00 value +-----+ ^ ^ ^ +-----+ Scalar Nested relation value value	

When a relation-valued attribute is formatted, it is treated as a nested tuple value. This is just like the treatment for nested relation tuples except the relation value can have multiple tuples.

From the previous discussion, we can formulate some rules for how the formatted output of relation tuples and values appear.

- The outer-most table consists of a heading and a body.
- A heading consists of three rows:
 - The string `RelTuple` or `RelValue` to indicate the value type. The first cell of this row spans the remaining columns.
 - The name of the attributes. Each attribute name is placed in a separate cell.
 - The type of the attributes. Each attribute data type is place din a separate cell.
- Each heading row is degree columns wide.
- If the type of an attribute is either `RelTuple` or `RelValue`, then the heading is extended down and to the right to hold the attribute names / types following the same rules as any other heading.
- A tuple value contributes one row to the body.
- A relation value contributes one row for each tuple in the relation value. An empty relation value still consumes a row, but its cell is filled with tilde (`\~`) characters.

- Each cell contains a string and supplies space for the left and top border.
- The bottom and right borders for the table are considered separate from cell borders.
- The width of all cells in a column are sized to the widest string in any of the cells in the column.
- All cell content is left justified in the cell and any additional space needed to make the cell the correct size is padded on the right.

Representing the Formatted Table

Since we are formatting relation tuples and values, we need access to those modules.

```
<<reformat module declarations>>=
const reltuple = @import("reltuple.zig");
const RelTuple = reltuple.RelTuple;
const isRelTupleType = reltuple.isRelTupleType;

const relvalue = @import("relvalue.zig");
const RelValue = relvalue.RelValue;
const isRelValueType = relvalue.isRelValueType;
const LayoutCapacity = relvalue.RelationCapacity;
```

The design uses the following constructs:

- A `RelFormatter` structure is used by the `format` functions of the `RelTuple` and `RelValue` types.
- A `Cell` represents a logical cell in the tabular representation.
- A `RelationTable` holds the layout of the value and is the container for the cells that make up the table. This data type does the heavy lifting to determine both the layout of the data in a tabular fashion, but also the formatting of the values into a string.

RelFormatter

We start with the `RelFormatter`. The `RelFormatter` uses an allocator to obtain the memory required for the operation. Since the lifetime of a `RelFormatter` is well bounded, The allocated memory is returned by the `init` method. The `format` functions for relation tuples and relation values use the `relvalue_allocator` and return the memory after formatting.

```
<<reformat module declarations>>=
/// The base data structure used to format relation tuples and values.
pub const RelFormatter = struct {
    allocator: Allocator,
    table: RelationTable = undefined,
    buffer: ?[]u8, // formatted result

    <<table formatter methods>>
};
```

```
<<table formatter methods>>=
pub fn init(
    allocator: Allocator,
) !RelFormatter {
    const self = RelFormatter{
        .allocator = allocator,
        .buffer = null,
    };

    return self;
}
```

```
<<table formatter methods>>=
pub fn_deinit(
    self: *RelFormatter,
) void {
    self.table.deinit(self.allocator);
    if (self.buffer) |buffer| {
        self.allocator.free(buffer);
    }
    self.* = undefined;
}
```

Formatting a relation tuple or a relation value have separate functions that are provided for the format functions of relation tuple and relation values. This makes it simpler to decide what type of value is being formatted.

```
<<table formatter methods>>=
pub fn formatRelTuple(
    self: *RelFormatter,
    value: anytype,
) ![]const u8 {
    self.table = try RelationTable.layoutRelTuple(self.allocator, value);
    const formatted = try self.formatTable(value);
    return formatted[0..formatted.len - 1]; // discard the trailing newline
}
```

```
<<table formatter methods>>=
pub fn formatRelValue(
    self: *RelFormatter,
    value: anytype,
) ![]const u8 {
    // Note that DUM and DEE are just special cased.
    // This save a lot of effort required to handle the boundary
```

```

// cases these relation values represent. They don't show up very often.
if (value.degree() == 0) { // DUM or DEE
    return if (value.cardinality() == 0) // DUM
        \n+-----+
        \|RelValue|
        \n+-----+
        \|~~~~~|
        \n+-----+
else // DEE
    \n+-----+
    \|RelValue|
    \n+-----+
    \|~~~~~|
    \n+-----+
    \|~~~~~|
    \n+-----+
;
}
self.table = try RelationTable.layoutRelValue(self.allocator, value);
const formatted = try self.formatTable(value);
return formatted[0..formatted.len - 1]; // discard the trailing newline
}

```

Once the tabular representation is determined, formatting the result is not specific to relation tuples or relation values. This function returns a slice containing the printable string.

```

<<table formatter methods>>=
fn formatTable(
    self: *RelFormatter,
    value: anytype,
) ![]u8 {
    const table_width = self.table.width;
    const table_height = self.table.height;
    const grid_count = table_width * table_height;
    var grid_places = try self.allocator.alloc(u8, grid_count);
    self.buffer = grid_places;

    @memset(grid_places, ' ');
    // Slice up the buffer of grid character cells into "height" slices of "width" chunks.
    // This makes indexing easier.
    var grid = try self.allocator.alloc([]u8, table_height);
    defer self.allocator.free(grid);

    var grid_cursor: usize = 0;
    for (0..table_height) |row_index| {
        grid[row_index] = grid_places[grid_cursor..grid_cursor + table_width];
        grid_cursor += table_width;
    }

    try self.table.drawTable(value, grid);
}

```

```
    return grid_places;
}
```

Cell

The `Cell` structure describes each logical cell in tabular representation. The contents of a `Cell` are mapped onto printable characters during the format operations.

```
<<reformat module declarations>>=
const Cell = struct {
    // An enumeration indicating the type of content of the cell.
    const CellType = enum {
        unused,
        banner,
        type,
        name,
        value,
    };

    // Cells can span columns and rows or just be self contained.
    const SpanType = enum {
        no_span,
        horizontal_span,
        vertical_span,
    };

    type: CellType = .unused,
    span: SpanType = .no_span,
    // The X, Y coordinates of where the cell is located in the output string.
    row_coord: LayoutCapacity = 0,
    col_coord: LayoutCapacity = 0,
    // The width of the cell's printable content. An extra character is added
    // to this value during the formatting to account for the left border character.
    width: LayoutCapacity = 0,
    // A cell is always 2 characters high. One for the upper border and one for
    // the cell value.
    height: LayoutCapacity = 2,
    <<cell methods>>
};
```

```
<<cell methods>>=
/// Draw simple text string in the cell.
fn draw(
    self: Cell,
    grid: [][]u8,
    text: []const u8,
```

```

) void {
    self.drawTopBorder(grid);
    self.drawLeftBorder(grid);

    if (text.len == 0) {
        return;
    }

    const text_row = self.row_coord + 1; // skip top border
    const text_col = self.col_coord + 1; // skip left border
    const text_len = @min(self.width - 1, text.len); // -1 for the left border
    @memcpy(grid[text_row][text_col .. text_col + text_len], text);
}

```

When a value needs to be drawn onto the grid, we use the normal formatted output function.

```

<<cell methods>>=
/// Draw a formatted value in the cell.
fn drawValue(
    self: Cell,
    grid: [][]u8,
    value: anytype,
) void {
    self.drawTopBorder(grid);
    self.drawLeftBorder(grid);

    const text_row = self.row_coord + 1;
    const text_col = self.col_coord + 1;
    const text_len = self.width - 1;
    const specifier: []const u8 =
        if (comptime trait.isZigString(@TypeOf(value)))
            "{s}"
        else
            "{any}";

    _ = fmt.bufPrint(
        grid[text_row][text_col..text_col + text_len],
        specifier,
        .{value},
    ) catch unreachable;
}

```

The third variation on drawing is just to fill the cell with a specific character.

```

<<cell methods>>=
/// Fill a cell with a single character.
fn drawFill(

```

```

    self: Cell,
    grid: [][]u8,
    fill: u8,
) void {
    self.drawTopBorder(grid);
    self.drawLeftBorder(grid);

    const text_row = self.row_coord + 1;
    const text_col = self.col_coord + 1;
    const text_len = self.width - 1;
    @memset(grid[text_row][text_col..text_col + text_len], fill);
}

```

The appearance of the top border for a cell depends upon the ``span'' of the cell.

```

<<cell methods>>=
fn drawTopBorder(
    self: Cell,
    grid: [][]u8,
) void {
    switch (self.span) {
        // .no_span cells are delimited by a '+' character, e.g. "-----".
        .no_span => {
            grid[self.row_coord][self.col_coord] = '+';
            const dash_col = self.col_coord + 1;
            const dash_width = self.width - 1;
            @memset(grid[self.row_coord][dash_col .. dash_col + dash_width], '-');
        },
        // .horizontal_span cells remove the '+' character, e.g. "-----".
        .horizontal_span => {
            @memset(grid[self.row_coord][self.col_coord .. self.col_coord + self.width], '-');
        },
        // .vertical_span cells use a '|' to show a vertical continuation.
        .vertical_span => {
            grid[self.row_coord][self.col_coord] = '|';
        },
    }
}

```

The left border of any cell is always a '|' character to begin the row with the cell value.

```

<<cell methods>>=
fn drawLeftBorder(
    self: Cell,
    grid: [][]u8,
) void {

```

```

if (self.span != .horizontal_span) {
    grid[self.row_coord + 1][self.col_coord] = '|';
}
}

```

The bottom border is drawn along the last row of the table. For bottom border purposes, all cells are treated as `.no_span` cells.

```

<<cell methods>>=
fn drawBottomBorder(
    self: Cell,
    grid: [][]u8,
) void {
    const bottom_row = self.row_coord + self.height;
    grid[bottom_row][self.col_coord] = '+';
    const dash_col = self.col_coord + 1;
    const dash_width = self.width - 1;
    @memset(grid[bottom_row][dash_col .. dash_col + dash_width], '-');
}

```

For the right border, we want to make sure the vertically spanned cells have a '`|`' to show the vertical continuation. Otherwise, we need a '`+`' character to mark the cell boundary.

```

<<cell methods>>=
fn drawRightBorder(
    self: Cell,
    grid: [][]u8,
) void {
    const border_col = self.col_coord + self.width;
    grid[self.row_coord + 1][border_col] = '|';

    switch (self.span) {
        .no_span, .horizontal_span => {
            grid[self.row_coord][border_col] = '+';
        },
        .vertical_span => {
            grid[self.row_coord][border_col] = '|';
        },
    }
}

```

RelationTable

To produce the formatted output, the `RelationTable` type performs the following operations:

1. Depending upon whether we are formatting tuples or relation values, a separate function is provided to begin the operation.

2. The value to be output is scanned to determine the number of heading and body rows and the number of columns.
3. Initialization obtains the required memory to hold the cells of the table.
4. The layout of the value into cells is calculated, starting with the heading and followed by the body.
5. The cells are “squared up” to make a rectangular rendering of the values.
6. Open horizontal space is coalesced to span the required columns.
7. Open vertical space is coalesced to span the required rows.
8. The coordinate locations within the formatted output string of the cell are calculated.

Once the layout completes without error, the RelFormatter requests that the layout be drawn onto a character string that it supplies.

```
<<reformat module declarations>>=
const RelationTable = struct {
    width: LayoutCapacity = 0,
    height: LayoutCapacity = 0,
    heading_row_count: LayoutCapacity = 0,
    body_row_count: LayoutCapacity = 0,
    col_count: LayoutCapacity = 0,
    cells: []Cell,
    rows: [][]Cell,

    const Dimension = struct {
        row_count: LayoutCapacity = 0,
        col_count: LayoutCapacity = 0,
    };
}

<<relation table methods>>
};
```

```
<<relation table methods>>=
/// Layout a relation tuple into cells.
fn layoutRelTuple(
    allocator: Allocator,
    value: anytype,
) !RelationTable {
    const ValueType = switch (@typeInfo(@TypeOf(value))) {
        .pointer => @TypeOf(value.*),
        else => @TypeOf(value),
    };

    const heading_dim = sizeHeading(ValueType);
    const body_row_count = sizeTupleBody(value);
    var self = try RelationTable.init(
        allocator,
```

```

        heading_dim.row_count,
        body_row_count,
        heading_dim.col_count,
    );

    _ = self.layoutHeading(ValueType, 0, 0);
    const occupied_rows = self.layoutTupleBody(value, heading_dim.row_count, 0);
    assert(occupied_rows == body_row_count);

    self.squareUp();
    self.coalesceTableHorizontal(ValueType, 0, 0);
    self.coalesceTableVertical();
    self.locate(0, 0);

    return self;
}

```

```

<<relation table methods>>=
/// Layout a relation value into cells.
fn layoutRelValue(
    allocator: Allocator,
    value: anytype,
) !RelationTable {
    const ValueType = switch (@typeInfo(@TypeOf(value))) {
        .pointer => @TypeOf(value.*),
        else => @TypeOf(value),
    };

    const heading_dim = sizeHeading(ValueType);
    const body_row_count = sizeValueBody(value);
    var self = try RelationTable.init(
        allocator,
        heading_dim.row_count,
        body_row_count,
        heading_dim.col_count,
    );

    _ = self.layoutHeading(ValueType, 0, 0);
    const occupied_rows = self.layoutValueBody(value, heading_dim.row_count, 0);
    assert(occupied_rows == body_row_count);

    self.squareUp();
    self.coalesceTableHorizontal(ValueType, 0, 0);
    self.coalesceTableVertical();
    self.locate(0, 0);

    return self;
}

```

```

<<relation table methods>>=
fn init(
    allocator: Allocator,
    heading_row_count: LayoutCapacity,
    body_row_count: LayoutCapacity,
    col_count: LayoutCapacity,
) !RelationTable {
    const row_count = heading_row_count + body_row_count;
    const cells = try allocator.alloc(Cell, row_count * col_count);
    errdefer allocator.free(cells);
    @memset(cells, Cell{});

    // Slice up the cells into rows to make a grid.
    // Indexing is simpler this way.
    var rows = try allocator.alloc([]Cell, row_count);
    var cell_cursor: usize = 0;
    var row_cursor: usize = 0;
    while (row_cursor < row_count) :
        ({ row_cursor += 1; cell_cursor += col_count; }) {
            rows[row_cursor] = cells[cell_cursor..cell_cursor + col_count];
        }

    return RelationTable{
        .width = 0,
        .height = 0,
        .heading_row_count = heading_row_count,
        .body_row_count = body_row_count,
        .col_count = col_count,
        .cells = cells,
        .rows = rows,
    };
}

```

```

<<relation table methods>>=
fn deinit(
    self: *RelationTable,
    allocator: Allocator,
) void {
    allocator.free(self.cells);
    allocator.free(self.rows);
    self.* = undefined;
}

```

```

<<relation table methods>>=
/// Compute the number of rows and columns in the layout of a relational heading.
fn sizeHeading(
    comptime Heading: type,

```

```

) Dimension {
    const min_heading_rows: LayoutCapacity = 3;
    var row_count: LayoutCapacity = min_heading_rows;
    var col_count: LayoutCapacity = 0;

    const attr_ids = comptime enums.values(Heading.AttributeId);
    inline for (attr_ids) |attr_id| {
        const AttrType = Heading.AttributeType(attr_id);
        if (comptime (isRelTupleType(AttrType) or isRelValueType(AttrType))) {
            const tuple_dim = sizeHeading(AttrType);
            const additional_rows = min_heading_rows + tuple_dim.row_count - 1;
            row_count = @max(row_count, additional_rows);
            col_count += tuple_dim.col_count;
        } else {
            col_count += 1;
        }
    }

    return Dimension{
        .row_count = row_count,
        .col_count = col_count,
    };
}

```

```

<<relation table methods>>=
/// Compute the number of rows in the layout of a relation tuple.
fn sizeTupleBody(
    value: anytype,
) LayoutCapacity {
    const ValueType = switch (@typeInfo(@TypeOf(value))) {
        .pointer => @TypeOf(value.*),
        else => @TypeOf(value),
    };

    var max_attr_rows: LayoutCapacity = 0;
    const attr_ids = comptime enums.values(ValueType.AttributeId);
    inline for (attr_ids) |attr_id| {
        const AttrType = ValueType.AttributeType(attr_id);
        if (comptime isRelValueType(AttrType)) {
            const attr_value = value.extract(attr_id);
            max_attr_rows = @max(max_attr_rows, sizeValueBody(attr_value));
        } else {
            max_attr_rows = @max(max_attr_rows, 1);
        }
    }

    return max_attr_rows;
}

```

```

<<relation table methods>>
/// Compute the number of rows in the layout of a relation value.
fn sizeValueBody(
    value: anytype,
) LayoutCapacity {
    const ValueType = switch (@typeInfo(@TypeOf(value))) {
        .pointer => @TypeOf(value.*),
        else => @TypeOf(value),
    };

    var total_body_rows: LayoutCapacity = 0;
    const attr_ids = comptime enums.values(ValueType.AttributeId);
    for (value.body()) |value_tuple| {
        var max_tuple_rows: LayoutCapacity = 0;
        inline for (attr_ids) |attr_id| {
            const AttrType = ValueType.AttributeType(attr_id);
            if (comptime isRelValueType(AttrType)) {
                const attr_value = value_tuple.extract(attr_id);
                max_tuple_rows = @max(max_tuple_rows, sizeValueBody(attr_value));
            } else {
                max_tuple_rows = @max(max_tuple_rows, 1);
            }
        }
        total_body_rows += max_tuple_rows;
    }
    return total_body_rows;
}

```

```

<<relation table methods>>
/// Compute the cell layout for a relational heading.
fn layoutHeading(
    self: *RelationTable,
    comptime Heading: type,
    row: LayoutCapacity,
    column: LayoutCapacity,
) LayoutCapacity {
    self.layoutBannerRow(Heading, row, column);
    return self.layoutAttrNames(Heading, row + 1, column);
}

```

```

<<relation table methods>>
/// Compute the cell layout for the banner row of a relational heading.
fn layoutBannerRow(
    self: *RelationTable,
    comptime Heading: type,

```

```

    row: LayoutCapacity,
    column: LayoutCapacity,
) void {
    const banner_width: usize = if (comptime isRelTupleType(Heading))
        "RelTuple".len
    else if (comptime isRelValueType(Heading))
        "RelValue".len
    else
        comptime unreachable;

    self.rows[row][column] = Cell{
        .type = .banner,
        .width = banner_width + 1,
        .height = 2,
    };
}

```

```

<<relation table methods>>=
/// Compute the cell layout for the attribute names in a relational heading.
fn layoutAttrNames(
    self: *RelationTable,
    comptime Heading: type,
    row: LayoutCapacity,
    column: LayoutCapacity,
) LayoutCapacity {
    const attr_ids = comptime enums.values(Heading.AttributeId);
    var occupied_cols: LayoutCapacity = 0;
    const name_row = self.rows[row];
    var col_index = column;
    inline for (attr_ids) |attr_id| {
        name_row[col_index] = Cell{
            .type = .name,
            .span = .no_span,
            .width = @truncate(@tagName(attr_id).len + 1),
            .height = 2,
        };

        const AttrType = Heading.AttributeType(attr_id);
        if (comptime (isRelTupleType(AttrType) or isRelValueType(AttrType))) {
            const heading_cols = self.layoutHeading(AttrType, row + 1, @truncate(
                col_index));
            occupied_cols += heading_cols;
            col_index += heading_cols;
        } else {
            const type_cols = self.layoutAttrType(Heading, attr_id, row + 1, col_index);
            occupied_cols += type_cols;
            col_index += type_cols;
        }
    }
}

```

```

    return occupied_cols;
}

```

```

<<relation table methods>>=
/// Compute the cell layout for the attribute data types in a relational heading.
fn layoutAttrType(
    self: *RelationTable,
    comptime Heading: type,
    comptime attr_id: Heading.AttributeId,
    row: LayoutCapacity,
    column: LayoutCapacity,
) LayoutCapacity {
    var occupied_cols: LayoutCapacity = 0;
    const type_row = self.rows[row];
    const AttrType = Heading.AttributeType(attr_id);
    if (comptime (isRelTupleType(AttrType) or isRelValueType(AttrType))) {
        occupied_cols = self.layoutHeading(AttrType, row, @truncate(column));
    } else {
        type_row[column] = Cell{
            .type = .type,
            .span = .no_span,
            .width = @truncate(@typeName(AttrType).len + 1),
            .height = 2,
        };
        occupied_cols = 1;
    }
    return occupied_cols;
}

```

```

<<relation table methods>>=
/// Compute the cell layout for the body of a relation tuple.
fn layoutTupleBody(
    self: *RelationTable,
    value: anytype,
    row: LayoutCapacity,
    column: LayoutCapacity,
) LayoutCapacity {
    const ValueType = switch (@typeInfo(@TypeOf(value))) {
        .pointer => @TypeOf(value.*),
        else => @TypeOf(value),
    };
    const attr_ids = comptime enums.values(ValueType.AttributeId);
    var occupied_rows: LayoutCapacity = 0;
    var col_index = column;
    if (attr_ids.len == 0) {

```

```

        occupied_rows = 1;
    } else {
        inline for (attr_ids) |attr_id| {
            const AttrType = ValueType.AttributeType(attr_id);
            const attr_value = value.extract(attr_id);
            if (comptime isRelTupleType(AttrType)) {
                const tuple_rows =
                    self.layoutTupleBody(attr_value, row, @truncate(col_index));
                occupied_rows = @max(occupied_rows, tuple_rows);
                col_index += @truncate(AttrType.tuple_degree);
            } else if (comptime isRelValueType(AttrType)) {
                const body_rows =
                    self.layoutValueBody(attr_value, row, @truncate(col_index));
                occupied_rows = @max(occupied_rows, body_rows);
                col_index += @truncate(AttrType.Tuple.tuple_degree);
            } else {
                var cell = &self.rows[row][col_index];
                cell.type = .value;
                cell.span = .no_span;
                // +1 for the left border character
                cell.width = scalarValueWidth(attr_value) + 1;
                occupied_rows = @max(occupied_rows, 1);
                col_index += 1;
            }
        }
    }

    return occupied_rows;
}

```

```

<<relation table methods>>=
fn scalarValueWidth(
    attr_value: anytype,
) LayoutCapacity {
    return if (comptime trait.isZigString(@TypeOf(attr_value)))
        @truncate(attr_value.len)
    else
        @truncate(fmt.count("{any}", .{attr_value}));
}

```

```

<<relation table methods>>=
/// Compute the cell layout for the body of a relation value.
fn layoutValueBody(
    self: *RelationTable,
    value: anytype,
    row: LayoutCapacity,
    column: LayoutCapacity,

```

```

) LayoutCapacity {
    var next_row = row;
    var occupied_rows: LayoutCapacity = 0;
    const body = value.body();
    if (body.len == 0) {
        occupied_rows = 1;
    } else {
        for (body) |rel_tuple| {
            const tuple_rows = self.layoutTupleBody(rel_tuple, next_row, column);
            next_row += tuple_rows;
            occupied_rows += tuple_rows;
        }
    }

    return occupied_rows;
}

```

```

<<relation table methods>>=
/// Adjust the width of cells in a table so that the table is rectangular and
/// cells in the same column are the same size.
fn squareUp(
    self: *RelationTable,
) void {
    // +2 -> space for right "|" border character and trailing "\n"
    self.width = self.totalWidth() + 2;
    self.height = @truncate(self.rows.len * 2 + 1);
}

```

```

<<relation table methods>>=
/// Compute the total width of a table. As a side effect, the width of
/// cells in each column is adjusted to be the size of the maximum cell
/// width of the column.
fn totalWidth(
    self: *RelationTable,
) LayoutCapacity {
    // Walk the columns to determine the maximum width of any given cell.
    // During the walk, total up the width of the entire layout.
    var total_width: LayoutCapacity = 0;

    const col_count: LayoutCapacity = @truncate(self.rows[0].len);
    for (0..col_count) |col_index| {
        var max_col_width: LayoutCapacity = 0;
        for (self.rows) |row| {
            const cell = &row[col_index];
            max_col_width = @max(max_col_width, cell.width);
        }
    }
}

```

```

// Walk the columns again, setting each cell width to the maximum found.
for (self.rows) |row| {
    const cell = &row[col_index];
    cell.width = max_col_width;
}

total_width += max_col_width;
}

return total_width;
}

```

```

<<relation table methods>>=
/// Scan the table to group contiguous horizontally spanned cells.
fn coalesceTableHorizontal(
    self: *RelationTable,
    ValueType: type,
    row_index: LayoutCapacity,
    col_index: LayoutCapacity,
) void {
    const table_cols = self.coalesceHorizontal(ValueType, row_index + 1, col_index);

    if (table_cols > 1) {
        markHorizontal(self.rows[row_index][col_index + 1 .. table_cols]);
    }
}

```

```

<<relation table methods>>=
fn coalesceHorizontal(
    self: *RelationTable,
    ValueType: type,
    row_index: LayoutCapacity,
    col_index: LayoutCapacity,
) LayoutCapacity {
    var total_col_width: LayoutCapacity = 0;
    var next_col: LayoutCapacity = col_index;

    const attr_ids = comptime enums.values(ValueType.AttributeId);
    inline for (attr_ids) |attr_id| {
        const AttrType = ValueType.AttributeType(attr_id);
        if (comptime (isRelValueType(AttrType) or isRelTupleType(AttrType))) {
            const value_cols = self.coalesceHorizontal(AttrType, row_index + 2, next_col);
            if (value_cols > 1) {
                const next_row = row_index + 1;
                const horiz_col = next_col + 1;
                markHorizontal(self.rows[row_index][horiz_col .. horiz_col + value_cols - 1]);
            }
        }
    }
}

```

```

        markHorizontal(self.rows[next_row][horiz_col .. horiz_col + value_cols - 1]);
    }
    next_col += value_cols;
    total_col_width += value_cols;
} else {
    next_col += 1;
    total_col_width += 1;
}
}

return total_col_width;
}

```

```

<<relation table methods>>=
fn markHorizontal(
    cells: []Cell,
) void {
    for (cells) |*cell| {
        cell.span = .horizontal_span;
    }
}

```

```

<<relation table methods>>=
/// Scan the table to group contiguous vertically spanned cells.
fn coalesceTableVertical(
    self: *RelationTable,
) void {
    // Going bottom to top on each column,
    // find vertical spanning cells and accumulate their heights onto
    // the first cell that is not unused.
    const row_count = self.rows.len;
    const col_count = self.rows[0].len;

    for (0..col_count) |col_index| {
        var row_index = row_count - 1;
        for (0..row_count) |_| {
            const cell = &self.rows[row_index][col_index];
            if (cell.type == .unused and cell.span == .no_span) {
                cell.span = .vertical_span;
            }
            row_index -= 1;
        }
    }
}

```

The coordinate system of the printable buffer system increases from top-to-bottom and from left-to-

right as is typical in drawing schemes. The coordinates of each cell are the upper, left edge where the border is placed.

```
<<relation table methods>>=
/// Determine the location of each cell in the table in an output
/// character buffer.
pub fn locate(
    self: *RelationTable,
    row_index: LayoutCapacity,
    col_index: LayoutCapacity,
) void {
    // Row by row, column by column, walk the table layout
    // calculating the coordinates for each cell.
    var row_coord = row_index;
    for (self.rows) |row| {
        var col_coord = col_index;
        for (row) |*cell| {
            cell.row_coord = row_coord;
            cell.col_coord = col_coord;

            col_coord += cell.width;
        }
        row_coord += 2;
    }
}
```

Drawing the Table

Once the layout is computed and each cell is given a location in the output buffer grid, we then scan the cells again placing the contents of each cell at the proper location in the buffer.

```
<<relation table methods>>=
/// Render the table cells into the output buffer grid.
fn drawTable(
    self: *RelationTable,
    value: anytype,
    grid: [][]u8,
) !void {
    const ValueType = switch (@typeInfo(@TypeOf(value))) {
        .pointer => @TypeOf(value.*),
        else => @TypeOf(value),
    };

    _ = self.drawHeading(ValueType, grid, 0, 0);

    if (comptime isRelTupleType(ValueType)) {
        _ = self.drawTupleBody(value, grid, self.heading_row_count, 0);
    }
}
```

```

} else if (comptime isRelValueType(ValueType)) {
    _ = self.drawValueBody(value, grid, self.heading_row_count, 0);
} else {
    @compileError("unexpected value type: '" ++ @typeName(ValueType) ++ "'\n");
}

self.drawUnusedCells(grid);
self.drawTableBottomBorder(grid);
self.drawTableRightBorder(grid);
lineTerminate(grid);
}

```

```

<<relation table methods>>=
/// Draw the heading for the table.
fn drawHeading(
    self: *RelationTable,
    comptime Heading: type,
    grid: [][]u8,
    row_index: LayoutCapacity,
    col_index: LayoutCapacity,
) LayoutCapacity {
    self.drawBannerRow(Heading, grid, row_index, col_index);
    return self.drawAttributeInfo(Heading, grid, row_index + 1, col_index);
}

```

The table heading consists of the banner row describing the type of the value.

```

<<relation table methods>>=
fn drawBannerRow(
    self: *RelationTable,
    comptime Heading: type,
    grid: [][]u8,
    row_index: LayoutCapacity,
    col_index: LayoutCapacity,
) void {
    // Banner cell is "RelTuple" or "RelValue"
    const banner_value = if (comptime isRelTupleType(Heading))
        "RelTuple"
    else if (comptime isRelValueType(Heading))
        "RelValue"
    else
        comptime unreachable;

    const banner_cell = self.rows[row_index][col_index];
    banner_cell.draw(grid, banner_value);
}

```

```

<<relation table methods>>
/// Draw the attribute name / attribute data types for the heading.
fn drawAttributeInfo(
    self: *RelationTable,
    comptime Heading: type,
    grid: [][]u8,
    row_index: LayoutCapacity,
    col_index: LayoutCapacity,
) LayoutCapacity {
    var occupied_cols: LayoutCapacity = 0;
    var dest_col = col_index;
    const name_row = self.rows[row_index];

    const attr_ids = comptime enums.values(Heading.AttributeId);
    inline for (attr_ids) |attr_id| {
        name_row[dest_col].draw(grid, @tagName(attr_id));

        const AttrType = Heading.AttributeType(attr_id);
        const type_row = row_index + 1;
        if (comptime (isRelTupleType(AttrType) or isRelValueType(AttrType))) {
            const heading_cols =
                self.drawHeading(AttrType, grid, type_row, dest_col);
            occupied_cols += heading_cols;
            dest_col += heading_cols;
        } else {
            self.rows[type_row][dest_col].draw(grid, @typeName(AttrType));
            occupied_cols += 1;
            dest_col += 1;
        }
    }
    return occupied_cols;
}

```

```

<<relation table methods>>
/// Draw the values of a tuple body into the character grid.
fn drawTupleBody(
    self: *RelationTable,
    value: anytype,
    grid: [][]u8,
    row_index: LayoutCapacity,
    col_index: LayoutCapacity,
) Dimension {
    const ValueType = switch (@typeInfo(@TypeOf(value))) {
        .pointer => @TypeOf(value.*),
        else => @TypeOf(value),
    };

```

```

const attr_ids = comptime enums.values(ValueType.AttributeId);
var occupied_cells: Dimension = .{};
inline for (attr_ids) |attr_id| {
    const next_col = col_index + occupied_cells.col_count;
    const AttrType = ValueType.AttributeType(attr_id);
    const attr_value = value.extract(attr_id);
    if (comptime isRelTupleType(AttrType)) {
        const tuple_occupied =
            self.drawTupleBody(attr_value, grid, row_index, next_col);
        occupied_cells.row_count =
            @max(occupied_cells.row_count, tuple_occupied.row_count);
        occupied_cells.col_count += tuple_occupied.col_count;
    } else if (comptime isRelValueType(AttrType)) {
        const body_occupied =
            self.drawValueBody(attr_value, grid, row_index, next_col);
        occupied_cells.row_count =
            @max(occupied_cells.row_count, body_occupied.row_count);
        occupied_cells.col_count += body_occupied.col_count;
    } else {
        var cell = &self.rows[row_index][next_col];
        cell.type = .value;
        cell.span = .no_span;
        cell.writeValue(grid, attr_value);
        occupied_cells.row_count = @max(occupied_cells.row_count, 1);
        occupied_cells.col_count += 1;
    }
}
return occupied_cells;
}

```

```

<<relation table methods>>=
/// Draw the tuples of a relation value body into the character grid.
fn drawValueBody(
    self: *RelationTable,
    value: anytype,
    grid: [][]u8,
    row_index: LayoutCapacity,
    col_index: LayoutCapacity,
) Dimension {
    var occupied_cells: Dimension = .{};
    var next_row = row_index;

    const body = value.body();
    // Empty relation values still occupy space in the output,
    // but any cells are filled with tilde characters.
    if (body.len == 0) {
        occupied_cells.row_count = 1;
        occupied_cells.col_count = @truncate(value.degree());
        var col_offset = col_index;

```

```

        for (0 .. occupied_cells.col_count) |attr_offset| {
            col_offset += @truncate(attr_offset);
            const cell = &self.rows[row_index][col_offset];
            cell.type = .value;
            cell.span = .no_span;
            cell.drawFill(grid, '~');
        }
    } else {
        for (body) |rel_tuple| {
            const tuple_occupied =
                self.drawTupleBody(rel_tuple, grid, next_row, col_index);
            next_row += tuple_occupied.row_count;
            occupied_cells.row_count += tuple_occupied.row_count;
            occupied_cells.col_count = tuple_occupied.col_count;
        }
    }

    return occupied_cells;
}

```

Unused cells must still be drawn onto the grid in order to get the appropriate top and left border characters drawn.

```

<<relation table methods>>=
fn drawUnusedCells(
    self: *RelationTable,
    grid: [][]u8,
) void {
    for (self.rows) |row| {
        for (row) |*cell| {
            if (cell.type == .unused) {
                cell.draw(grid, "");
            }
        }
    }
}

```

```

<<relation table methods>>=
/// Draw the bottom border on the table.
fn drawTableBottomBorder(
    self: *RelationTable,
    grid: [][]u8,
) void {
    const last_row = self.rows[self.rows.len - 1];
    for (0..last_row.len) |col_index| {
        last_row[col_index].drawBottomBorder(grid);
    }
}

```

```

// Final '+' at the lower left corner.
const bottom_right_cell = &last_row[last_row.len - 1];
const last_grid_row = bottom_right_cell.row_coord + bottom_right_cell.height;
const last_grid_col = bottom_right_cell.col_coord + bottom_right_cell.width;
grid[last_grid_row][last_grid_col] = '+';
}

```

```

<<relation table methods>>=
/// Draw the right border on the table.
fn drawTableRightBorder(
    self: *RelationTable,
    grid: [][]u8,
) void {
    const last_col = self.rows[0].len - 1;
    for (self.rows) |cell_row| {
        cell_row[last_col].drawRightBorder(grid);
    }
}

```

```

<<relation table methods>>=
/// Add a newline character at the end of each row in the character grid.
fn lineTerminate(
    grid: [][]u8,
) void {
    const last_char_loc = grid[0].len - 1;
    for (grid) |row| {
        row[last_char_loc] = '\n';
    }
}

```

Tests

Since we are dealing with relation values, we must manage the relation value arena allocator.

```

<<reformat module declarations>>=
const relvalue_allocator = relvalue.relvalue_allocator;
const relExprBegin = relvalue.relExprBegin;
const relExprEnd = relvalue.relExprEnd;

```

```

<<reformat module tests>>=
test "tuple formatting" {
    relExprBegin(); defer relExprEnd();
}

```

```

const TestTuple = RelTuple(struct {
    name: []const u8,
    number: usize,
    street: []const u8,
    city: []const u8,
    state: []const u8,
    zip: []const u8,
});

const test_1 = TestTuple.create(.{
    .name = "John",
    .number = 100,
    .street = "Elm",
    .city = "Las Vegas",
    .state = "NV",
    .zip = "55512",
});

var formatter_1 = try RelFormatter.init(testing.allocator);
defer formatter_1.deinit();
const result_1 = try formatter_1.formatRelTuple(test_1);

const expected_1 =

+-----+
|RelTuple
+-----+
|name      |number|street      |city       |state      |zip
+-----+
|[]const u8|usize |[]const u8|[]const u8|[]const u8|[]const u8|
+-----+
|John      |100   |Elm        |Las Vegas |NV        |55512
+-----+

;

try testing.expectEqualStrings(expected_1, result_1);

const test_2 = test_1.wrap("address", &{ .number, .street });
var formatter_2 = try RelFormatter.init(testing.allocator);
defer formatter_2.deinit();
const result_2 = try formatter_2.formatRelTuple(test_2);

const expected_2 =

+-----+
|RelTuple
+-----+
|name      |city       |state      |zip       |address
+-----+
|[]const u8|[]const u8|[]const u8|[]const u8|RelTuple
+-----+
||          ||          ||          +-----+
||          ||          ||          |number  |street
||          ||          ||          +-----+
||          ||          ||          |usize   |[]const u8
+-----+

;

```

```

VV|John      |Las Vegas |NV      |55512    |100     |Elm    |
VV+-----+-----+-----+-----+-----+-----+
;
try testing.expectEqualStrings(expected_2, result_2);
}

```

```

//<<reformat module tests>>=
test "value formatting" {
    // TODO: This test causes problem in the target.
    if (builtin.os.tag == .freestanding) return error.SkipZigTest;

    relExprBegin(); defer relExprEnd();

    const TestValue = RelValue(struct {
        name: []const u8,
        number: usize,
        street: []const u8,
        city: []const u8,
        state: []const u8,
        zip: []const u8,
    });
}

const test_1 = TestValue.create(&.{.
    .{ .name = "John", .number = 100, .street = "Elm",
        .city = "Las Vegas", .state = "NV", .zip = "55512",
    },
    .{ .name = "John", .number = 200, .street = "Maple",
        .city = "Sacramento", .state = "CA", .zip = "55512",
    },
    .{ .name = "John", .number = 700, .street = "Cypress",
        .city = "Lake Charles", .state = "LA", .zip = "55512",
    },
    .{ .name = "Jane", .number = 101, .street = "Vicki",
        .city = "Carson City", .state = "NV", .zip = "55512",
    },
    .{ .name = "Jane", .number = 201, .street = "Sunset",
        .city = "San Francisco", .state = "CA", .zip = "55512",
    },
});
}

var formatter_1 = try RelFormatter.init(testing.allocator);
defer formatter_1.deinit();
const result_1 = try formatter_1.formatRelValue(test_1);

const expected_1 =
    \\+-----+-----+-----+-----+-----+-----+
    \\|RelValue
    \\+-----+-----+-----+-----+-----+-----+
    \\|name      |number|street      |city          |state       |zip      |
    \\+-----+-----+-----+-----+-----+-----+

```

```

\\>[]const u8|usize |[]const u8|[]const u8 |[]const u8|[]const u8|
\\+-----+-----+-----+-----+-----+
\\|John |100 |Elm |Las Vegas |NV |55512 |
\\+-----+-----+-----+-----+-----+
\\|John |200 |Maple |Sacramento |CA |55512 |
\\+-----+-----+-----+-----+-----+
\\|John |700 |Cypress |Lake Charles |LA |55512 |
\\+-----+-----+-----+-----+-----+
\\|Jane |101 |Vicki |Carson City |NV |55512 |
\\+-----+-----+-----+-----+-----+
\\|Jane |201 |Sunset |San Francisco |CA |55512 |
\\+-----+-----+-----+-----+-----+
;

try testing.expectEqualStrings(expected_1, result_1);

const test_2 = test_1.wrap("address", &{ .number, .street });
var formatter_2 = try RelFormatter.init(testing.allocator);
defer formatter_2.deinit();
const result_2 = try formatter_2.formatRelValue(test_2);

const expected_2 =
\\+-----+
\\|RelValue
\\+-----+-----+-----+-----+-----+
\\|name |city |state |zip |address |
\\+-----+-----+-----+-----+-----+
\\|[]const u8|[]const u8 |[]const u8|[]const u8|RelTuple |
\\| | | | | |
\\| | | | | number |street |
\\| | | | | usize |[]const u8|
\\+-----+-----+-----+-----+-----+
\\|John |Las Vegas |NV |55512 |100 |Elm |
\\+-----+-----+-----+-----+-----+
\\|John |Sacramento |CA |55512 |200 |Maple |
\\+-----+-----+-----+-----+-----+
\\|John |Lake Charles |LA |55512 |700 |Cypress |
\\+-----+-----+-----+-----+-----+
\\|Jane |Carson City |NV |55512 |101 |Vicki |
\\+-----+-----+-----+-----+-----+
\\|Jane |San Francisco |CA |55512 |201 |Sunset |
\\+-----+-----+-----+-----+-----+
;

try testing.expectEqualStrings(expected_2, result_2);

const test_3 = test_2.wrap("location", &{ .city, .state, .zip });
var formatter_3 = try RelFormatter.init(testing.allocator);
defer formatter_3.deinit();
const result_3 = try formatter_3.formatRelValue(test_3);

const expected_3 =
\\+-----+

```

```

\\RelValue
\\+-----+
\\|name |address |location
\\+-----+
\\|[]const u8|RelTuple |RelTuple
\\| +-----+-----+-----+-----+
\\| |number |street |city |state |zip |
\\| +-----+-----+-----+-----+
\\| |usize |[]const u8|[]const u8 |[]const u8|[]const u8|
\\+-----+-----+-----+-----+-----+
\\|John |100 |Elm |Las Vegas |NV |55512 |
\\+-----+-----+-----+-----+-----+
\\|John |200 |Maple |Sacramento |CA |55512 |
\\+-----+-----+-----+-----+-----+
\\|John |700 |Cypress |Lake Charles |LA |55512 |
\\+-----+-----+-----+-----+-----+
\\|Jane |101 |Vicki |Carson City |NV |55512 |
\\+-----+-----+-----+-----+-----+
\\|Jane |201 |Sunset |San Francisco|CA |55512 |
\\+-----+-----+-----+-----+-----+
;

try testing.expectEqualStrings(expected_3, result_3);

const test_4 = test_3.group("domiciles", &{ .address, .location });
var formatter_4 = try RelFormatter.init(testing.allocator);
defer formatter_4.deinit();
const result_4 = try formatter_4.formatRelValue(test_4);
const expected_4 =
\\+-----+
\\|RelValue
\\+-----+
\\|name |domiciles
\\+-----+
\\|[]const u8|RelValue
\\| +-----+
\\| |address |location
\\| +-----+-----+
\\| |RelTuple |RelTuple
\\| +-----+-----+-----+
\\| |number |street |city |state |zip |
\\| +-----+-----+-----+-----+
\\| |usize |[]const u8|[]const u8 |[]const u8|[]const u8|
\\+-----+-----+-----+-----+-----+
\\|John |100 |Elm |Las Vegas |NV |55512 |
\\| +-----+-----+-----+-----+-----+
\\| |200 |Maple |Sacramento |CA |55512 |
\\| +-----+-----+-----+-----+
\\| |700 |Cypress |Lake Charles |LA |55512 |
\\| +-----+-----+-----+-----+-----+
\\| |Jane |101 |Vicki |Carson City |NV |55512 |
\\| +-----+-----+-----+-----+-----+
\\| |201 |Sunset |San Francisco|CA |55512 |
\\| +-----+-----+-----+-----+-----+

```

```

\\+-----+-----+-----+-----+-----+
;
try testing.expectEqualStrings(expected_4, result_4);
const test_5 = test_4.project(&.{.domiciles, .name});
var formatter_5 = try RelFormatter.init(testing.allocator);
defer formatter_5.deinit();
const result_5 = try formatter_5.formatRelValue(test_5);

const expected_5 =
\\+-----+
\\|RelValue
\\+-----+
\\|domiciles
\\+-----+
\\|RelValue          |[]const u8|
\\+-----+
\\|address          |location
\\+-----+
\\|RelTuple          |RelTuple
\\+-----+
\\|number   |street    |city      |state     |zip
\\+-----+
\\|usize    |[]const u8|[]const u8  |[]const u8|[]const u8|
\\+-----+
\\|100     |Elm       |Las Vegas |NV        |55512      |John
\\+-----+
\\|200     |Maple      |Sacramento |CA        |55512      |
\\+-----+
\\|700     |Cypress    |Lake Charles |LA        |55512      |
\\+-----+
\\|101     |Vicki      |Carson City |NV        |55512      |Jane
\\+-----+
\\|201     |Sunset     |San Francisco |CA        |55512      |
\\+-----+
;
try testing.expectEqualStrings(expected_5, result_5);
}

```

```

//<<reformat module tests>>=
test "relation-valued attribute formatting" {
    // TODO: This test causes problem in the target.
    if (builtin.os.tag == .freestanding) return error.SkipZigTest;

    relExprBegin(); defer relExprEnd();

    const Employee = RelValue(struct { name: []const u8, id: u16, dept: []const u8 });
    const employee = Employee.create(&{
        .{ .name = "Harry", .id = 27, .dept = "Finance" },
        .{ .name = "Sally", .id = 42, .dept = "Sales" },
        .{ .name = "George", .id = 175, .dept = "Finance" },
    });

```

```

.{ .name = "Harriet", .id = 200, .dept = "Sales" },
.{ .name = "Tim", .id = 83, .dept = "Executive" },
});

const Department = RelValue(struct { dept: []const u8, manager: []const u8 });
const dept = Department.create(&{
    .{ .dept = "Sales", .manager = "Harriet" },
    .{ .dept = "Production", .manager = "Charles" },
});

// groupedJoin creates relation-valued attributes.
const managers = employee.groupedJoin(dept, "managed_by");
var formatter_1 = try RelFormatter.init(testing.allocator);
defer formatter_1.deinit();
const result_1 = try formatter_1.formatRelValue(managers);

const expected_1 =
    \\+-----+
    \\|RelValue           |
    \\+-----+-----+-----+
    \\|name      |id |dept      |managed_by|
    \\+-----+-----+-----+
    \\|[]const u8|u16|[]const u8|RelValue   |
    \\|          |   |          +-----+
    \\|          |   |          |manager   |
    \\|          |   |          +-----+
    \\|          |   |          |[]const u8|
    \\+-----+-----+-----+
    \\|Harriet   |200|Sales      |Harriet   |
    \\+-----+-----+-----+
    \\|Tim       |83 |Executive  |~~~~~|
    \\+-----+-----+-----+
    \\|Sally     |42 |Sales      |Harriet   |
    \\+-----+-----+-----+
    \\|Harry     |27 |Finance    |~~~~~|
    \\+-----+-----+-----+
    \\|George    |175|Finance    |~~~~~|
    \\+-----+-----+-----+
;

try testing.expectEqualStrings(expected_1, result_1);

const employs = dept.groupedJoin(employee, "employs");
var formatter_2 = try RelFormatter.init(testing.allocator);
defer formatter_2.deinit();
const result_2 = try formatter_2.formatRelValue(employs);
const expected_2 =
    \\+-----+
    \\|RelValue           |
    \\+-----+-----+-----+
    \\|dept      |manager |employs   |
    \\+-----+-----+-----+
    \\|[]const u8|[]const u8|RelValue   |

```

```

\\|      +-----+---+
\\|      |name    |id |
\\|      +-----+---+
\\|      |[]const u8|u16|
\\+-----+-----+-----+
\\|Sales   |Harriet |Sally   |42 |
\\|      +-----+---+
\\|      |Harriet |200|
\\+-----+-----+-----+
\\|Production|Charles |~~~~~|~~~
\\+-----+-----+-----+
;

try testing.expectEqualStrings(expected_2, result_2);
}

```

```

<<reformat module tests>>=
test "dum / dee relvalue formatting" {
    relExprBegin(); defer relExprEnd();

    var formatter_dum = try RelFormatter.init(testing.allocator);
    // These defers for this test case only are causing an integer
    // overflow on Zig 0.14.0-dev.3026+c225b780e.
    // Don't know why it is only the deferred_deinit on these two
    // test cases. For now they are commented out and that doesn't
    // cause any problems because the relation expression allocator
    // is an arena allocator.
    //defer formatter_dum.deinit();
    const result_dum = try formatter_dum.formatRelValue(relvalue.dum());
    const expected_dum =
        \\+-----+
        \\|RelValue|
        \\+-----+
        \\|~~~~~|
        \\+-----+
    ;
    try testing.expectEqualStrings(expected_dum, result_dum);

    var formatter_dee = try RelFormatter.init(testing.allocator);
    //defer formatter_dee.deinit();
    const result_dee = try formatter_dee.formatRelValue(relvalue.dee());
    const expected_dee =
        \\+-----+
        \\|RelValue|
        \\+-----+
        \\|~~~~~|
        \\+-----+
        \\|~~~~~|
        \\+-----+
    ;
    try testing.expectEqualStrings(expected_dee, result_dee);
}

```

```
}
```

Code layout

```
<<reformat.zig>>
///! The `reformat.zig` module provides the ability to textually format
///! tuple and relation values in a tabular manner. This module is used
///! by the `RelTuple` and `RelValue` modules to provide formatted output
///! through the Zig standard library output formatting scheme.

<<copyright info>>
const std = @import("std");
const builtin = @import("builtin");
const testing = if (builtin.os.tag == .freestanding)
    @import("resee_testing")
else
    std.testing;
const assert = std.debug.assert;

const mem = std.mem;
const Allocator = memAllocator;
const fmt = stdfmt;
const meta = std.meta;
const enums = std.enums;
const trait = @import("zigtrait.zig");

<<reformat module declarations>>

<<reformat module public functions>>

<<reformat module private functions>>

<<reformat module tests>>
```

Tagged Catalogs

This module is an extension on the ideas contained in the standard library, `std.enums.EnumArray`, type function. The concept is to define a tagged union to serve as an array element for an `EnumArray`. The elements of the array may be obtained by the functions of the `EnumArray` and the active tag of the obtained element is the same as the enumeration used to access the array.

This in effect creates a catalog that is indexed by an enum literal and the array element has the same active tag as the index. The intent is to support larger catalogs where using a switch statement to determine the active tag is clumsy. This also supports tagging related structures to use them polymorphically. For example, if all the elements represent structures that are statically different data types but have the same methods, then invoking a method polymorphically can be made to appear as a simple method invocation with an additional argument of an enum literal to select the correct target.

Catalog Representation

```
<<tagged catalog module declarations>>=
pub const CatalogSpec = struct {
    tag_name: [:0]const u8,
    tag_type: type,
};
```

```
<<tagged catalog module public functions>>=
pub fn TaggedCatalog(
    tag_map: []const CatalogSpec,
) type {
    return struct {
        const Self = @This();

        <<catalog fields>>
        <<catalog declarations>>
        <<catalog public functions>>
    };
}
```

```
<<catalog declarations>>=
pub const CatalogTag = blk: {
    var tag_fields: [tag_map.len]EnumField = undefined;
    for (tag_map, 0..) |tag_entry, tag_index| {
        tag_fields[tag_index] = .{
            .name = tag_entry.tag_name,
            .value = tag_index,
        };
    }
};
```

```

    },
    break :tblk @Type(.{
        .@"enum" = .{
            .tag_type = math.IntFittingRange(0, tag_map.len),
            .fields = &tag_fields,
            .decls = &.{},
            .is_exhaustive = true,
        },
    });
}

```

```

<<catalog declarations>>=
pub const CatalogEntry = eblk: {
    var union_fields: [tag_map.len]UnionField = undefined;
    for (tag_map, 0..) |tag_entry, tag_index| {
        union_fields[tag_index] = .{
            .name = tag_entry.tag_name,
            .type = tag_entry.tag_type,
            .alignment = @alignOf(tag_entry.tag_type),
        };
    }

    break :eblk @Type(.{
        .@"union" = .{
            .layout = .auto,
            .tag_type = CatalogTag,
            .fields = &union_fields,
            .decls = &.{},
        },
    });
}

```

```

<<catalog declarations>>=
pub const Catalog = enums.EnumArray(CatalogTag, CatalogEntry);

```

```

<<catalog fields>>=
catalog: Catalog,

```

```

<<catalog declarations>>=
pub const len = Catalog.len;

```

```

<<catalog public functions>>=
pub fn init(

```

```

    init_values: enums.EnumFieldStruct(CatalogTag, CatalogEntry, null),
) Self {
    return .{
        .catalog = Catalog.initDefault(null, init_values),
    };
}

```

```

<<catalog public functions>>=
pub fn initDefault(
    comptime default: ?CatalogEntry,
    init_values: enums.EnumFieldStruct(CatalogTag, CatalogEntry, null),
) Self {
    return .{
        .catalog = Catalog.initDefault(default, init_values),
    };
}

```

```

<<catalog public functions>>=
pub fn initUndefined() Self {
    return .{
        .catalog = Catalog.initUndefined(),
    };
}

```

```

<<catalog public functions>>=
pub fn initFill(
    value: CatalogEntry,
) Self {
    return .{
        .catalog = Catalog.initFill(value),
    };
}

```

```

<<catalog public functions>>=
// get the value and unwrap it from the tagged union
pub fn get(
    self: Self,
    comptime key: CatalogTag,
) @FieldType(CatalogEntry, @tagName(key)) {
    return @field(self.catalog.get(key), @tagName(key));
}

```

```

<<catalog public functions>>=
pub fn getPtr(
    self: *Self,

```

```

    comptime key: CatalogTag,
) *@FieldType(CatalogEntry, @tagName(key)) {
    return &@field(self.catalog.getPtr(key), @tagName(key));
}

```

```

<<catalog public functions>>=
pub fn getPtrConst(
    self: *const Self,
    comptime key: CatalogTag,
) *const @FieldType(CatalogEntry, @tagName(key)) {
    return &@field(self.catalog.getPtrConst(key), @tagName(key));
}

```

```

<<catalog public functions>>=
pub fn set(
    self: *Self,
    comptime key: CatalogTag,
    value: @FieldType(CatalogEntry, @tagName(key)),
) void {
    self.catalog.set(key, @unionInit(CatalogEntry, @tagName(key), value));
}

```

```

<<catalog public functions>>=
pub fn iterator(
    self: *Self,
) Catalog.Iterator {
    return self.catalog.iterator();
}

```

Tests

```

<<tagged catalog module tests>>=
test "tagged catalogs" {
    const NumberCatalog = TaggedCatalog(&.{.
        .{ .tag_name = "integer", .tag_type = u16 },
        .{ .tag_name = "real", .tag_type = f32 },
    });
    var nums = NumberCatalog.init(.{
        .integer = @unionInit(NumberCatalog.CatalogEntry, "integer", 27),
        .real = @unionInit(NumberCatalog.CatalogEntry, "real", 5.7),
    });
}

```

```

try testing.expectEqual(@as(u16, 27), nums.get(.integer));

nums.set(.integer, 42);
try testing.expectEqual(@as(u16, 42), nums.get(.integer));

const num_ptr = nums.getPtr(.real);
try testing.expectEqual(@as(f32, 5.7), num_ptr.*);
num_ptr.* = 8.8;
try testing.expectEqual(@as(f32, 8.8), num_ptr.*);

const mod_ptr = nums.getPtrConst(.real);
try testing.expectEqual(@as(f32, 8.8), mod_ptr.*);

var catalog_iter = nums.iterator();
while (catalog_iter.next() |num| {
    switch(num.value.*) {
        .integer => |value| try testing.expectEqual(@as(u16, 42), value),
        .real => |value| try testing.expectEqual(@as(f32, 8.8), value),
    }
}
}

```

Code layout

```

<<tagged_catalog.zig>>=
<<zig edit warning>>
<<copyright info>>
/// This module is a variation on the ideas contained in the standard library,
/// std.enums.EnumArray type function. The concept is to define an array whose
/// elements are a tagged union. The elements of the array may be indexed by
/// using `@intFromEnum()` and the active tag of the element at that index
/// is also tagged with the same enum. This in effect creates a catalog that
/// is indexed by an enum literal and the array element has the same active tag
/// as the index. The intent is to support larger catalogs where using a
/// switch statement to determine the active tag is clumsy. This also supports
/// tagging related structures to use them polymorphically. For example,
/// if all the elements represent structures that are statically different
/// data types but have the same methods, then invoking a method polymorphically
/// can be made to appear as a simple method invocation with an additional
/// argument of an enum literal to select the correct target.
const std = @import("std");
const math = std.math;
const meta = std.meta;
const enums = std.enums;
const testing = std.testing;

const EnumField = std.builtin.Type.EnumField;
const UnionField = std.builtin.Type.UnionField;

pub const version = "1.0.0-a1";

```

```
<<tagged catalog module declarations>>  
<<tagged catalog module public functions>>  
<<tagged catalog module tests>>
```

Zig Traits Module

During the 0.12.0 development cycle, the standard library module `meta/trait.zig` was removed. All the trait functions were removed except for `isZigString`. This function determines if a given type will coerce to a `[]const u8`, which is the canonical type for a string in Zig.

The following code is a copy of the `isZigString` function. The filename has been changed to `zigtrait.zig` to avoid any confusion.

```
<<zigtrait.zig>>=
/// Returns true if the passed type will coerce to []const u8.
/// Any of the following are considered strings:
/// ``
/// []const u8, [:S]const u8, *const [N]u8, *const [N:S]u8,
/// []u8, [:S]u8, *[S]u8, *[N:S]u8.
/// ``
/// These types are not considered strings:
/// ``
/// u8, [N]u8, [*]const u8, [*:0]const u8,
/// [*]const [N]u8, []const u16, []const i8,
/// *const u8, ?[]const u8, ?*const [N]u8.
/// ```

pub inline fn isZigString(comptime T: type) bool {
    return blk: {
        // Only pointer types can be strings, no optionals
        const info = @typeInfo(T);
        if (info != .pointer) break :blk false;

        const ptr = &info.pointer;
        // Check for CV qualifiers that would prevent coercion to []const u8
        if (ptr.is_volatile or ptr.is_allowzero) break :blk false;

        // If it's already a slice, simple check.
        if (ptr.size == .slice) {
            break :blk ptr.child == u8;
        }

        // Otherwise check if it's an array type that coerces to slice.
        if (ptr.size == .one) {
            const child = @typeInfo(ptr.child);
            if (child == .array) {
                const arr = &child.array;
                break :blk arr.child == u8;
            }
        }

        break :blk false;
    };
}

const testing = @import("std").testing;
```

```

test "isZigString" {
    try testing.expect(isZigString([]const u8));
    try testing.expect(isZigString([]u8));
    try testing.expect(isZigString([:0]const u8));
    try testing.expect(isZigString([:0]u8));
    try testing.expect(isZigString([:5]const u8));
    try testing.expect(isZigString([:5]u8));
    try testing.expect(isZigString(*const [0]u8));
    try testing.expect(isZigString(*[0]u8));
    try testing.expect(isZigString(*const [0:0]u8));
    try testing.expect(isZigString(*[0:0]u8));
    try testing.expect(isZigString(*const [0:5]u8));
    try testing.expect(isZigString(*[0:5]u8));
    try testing.expect(isZigString(*const [10]u8));
    try testing.expect(isZigString(*[10]u8));
    try testing.expect(isZigString(*const [10:0]u8));
    try testing.expect(isZigString(*[10:0]u8));
    try testing.expect(isZigString(*const [10:5]u8));
    try testing.expect(isZigString(*[10:5]u8));

    try testing.expect(!isZigString(u8));
    try testing.expect(!isZigString([4]u8));
    try testing.expect(!isZigString([4:0]u8));
    try testing.expect(!isZigString([*]const u8));
    try testing.expect(!isZigString([*]const [4]u8));
    try testing.expect(!isZigString([*c]const u8));
    try testing.expect(!isZigString([*c]const [4]u8));
    try testing.expect(!isZigString([*:0]const u8));
    try testing.expect(!isZigString([*:0]const u8));
    try testing.expect(!isZigString(*[]const u8));
    try testing.expect(!isZigString(?[]const u8));
    try testing.expect(!isZigString(?*const [4]u8));
    try testing.expect(!isZigString([]allowzero u8));
    try testing.expect(!isZigString([]volatile u8));
    try testing.expect(!isZigString(*allowzero [4]u8));
    try testing.expect(!isZigString(*volatile [4]u8));
}

```

Supplemental materials

Bibliography

Books

- [defguide] Joseph Yiu, *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*, Elsevier (2014), ISBN 13:978-0-12-408082-9.
- [mb-xuml] Stephen J. Mellor and Marc J. Balcer, *Executable UML: a foundation for model-driven architecture*, Addison-Wesley (2002), ISBN 0-201-74804-5.
- [rs-xuml] Chris Raistrick, Paul Francis, John Wright, Colin Carter and Ian Wilkie, *Model Driven Architecture with Executable UML*, Cambridge University Press (2004), ISBN 0-521-53771-1.
- [mtoc] Leon Starr, Andrew Mangogna and Stephen Mellor, *Models to Code: With No Mysterious Gaps*, Apress (2017), ISBN 978-1-4842-2216-4
- [ls-build], Leon Starr, *How to Build Shlaer-Mellor Object Models*, Yourdon Press (1996), ISBN 0-13-207663-2.
- [sm-data] Sally Shlaer and Stephen J. Mellor, *Object Oriented Systems Analysis: Modeling the World in Data*, Prentice-Hall (1988), ISBN 0-13-629023-X.
- [sm-states] Sally Shlaer and Stephen J. Mellor, *Object Lifecycles: Modeling the World in States*, Prentice-Hall (1992), ISBN 0-13-629940-7.
- [halbwachs] Halbwachs, Nicolas, *Synchronous programming of reactive systems*, Springer Science+Business Media (1993), ISBN 978-1-4419-5133-5.
- [brooks] Brooks, Frederick, P. Jr., *The Mythical Man-Month*, Addison-Wesley (1995), ISBN 0-201-83595-9.
- [ahu-dsa] Alfred V. Aho, John E. Hopcroft, and Jeffery D. Ullman, *Data Structures and Algorithms*, Addison-Wesley (1987), ISBN 0-201-00023-7.

Articles

- [ls-articulate] Leon Starr, *How to Build Articulate UML Class Models*, 2008, <http://www.modelint.com/how-to-build-articulate-uml-class-models/>
- [ls-time] Leon Starr, *Time and Synchronization in Executable UML*, 2008, <http://www.modelint.com/time-and-synchronization-in-executable-uml/>
- [mosely-marks] Ben Mosely and Peter Marks, *Out of the Tar Pit*, 2006, <http://curtclifton.net/papers/MoseleyMarks06a.pdf>
- [foote-yoder] Brian Foote and Joseph Yoder, *Big Ball of Mud*, 1999, <http://www.laputan.org/mud/mud.html>
- [SM-Method] Sally Shlaer and Stephen J. Mellor *The Shlaer-Mellor Method*, 1996, <http://www.oatool.com/docs/SMMETHOD96.pdf>
- [SM-OOA96] Sally Shlaer and Neil Lang, *The Shlaer-Mellor Method: The OOA96 Report*, 1996, <http://oatool.com/docs/OOA96.pdf>

Videos

- [starr-spec-p1] Leon Starr, *How to Specify Complex Safety Critical Software Systems – Introduction Part 1*, 2024, <https://www.youtube.com/watch?v=ZE6RQCQIS3U>
- [starr-spec-p2] Leon Starr, *How to Specify Complex Safety Critical Software Systems – Introduction Part 2*, 2024, <https://www.youtube.com/watch?v=UGGqfCVa5xQ>

Glossary of Terms

AAPCS	ARM® Architecture Procedure Call Standard
ABI	application binary interface
ADC	analog to digital converter
BCD	binary coded decimal
BLE	Bluetooth low energy
BSS	block started by symbol — an historical assembler pseudo-operation carried forward into current usage to mean an area in memory where space for a variable has been allocated but no initial value has been given
CE	current era of calendar time
CMSIS	common microcontroller software interface standard
DAC	digital to analog converter
DMA	direct memory access
DSP	digital signal processing
DWT	data watchpoint and trace unit of the ARM Cortex processor
FPU	floating point unit
GPIO	general purpose input / output
I2C	Inter-Integrated Circuit
IRQ	Interrupt Request
KiB	kibibyte — equals 1024_{10} bytes
LED	light emitting diode
LFRC	low frequency R/C oscillator
LMA	load memory address
LSB	least significant bit
MPU	memory protection unit
MSP	main stack pointer
N.B.	note well — an abbreviation for the Latin term, <i>nota bene</i>
NMI	non-maskable interrupt
PC	program counter

POR	power on reset
PSP	process stack pointer
PWM	pulse width modulation
RAM	random access memory
RDMS	relational database management system
RTC	real time clock
ReSEE	reactive system execution environment
SDK	software development kit
SPI	Serial Peripheral Interface
SWO	software output — an output facility on Cortex-M devices
TCM	tightly coupled memory
UART	universal asynchronous receiver-transmitter
VLA	variable length array — a C99 addition
VMA	virtual memory address
XT	crystal oscillator, typically of high accuracy
YAGNI	you aren't gonna need it

Colophon

This document was formatted using the **asciidoc** document formatting program. The DejaVu font family was used throughout.

As a sad indication of our times, no part of this book was written using any form of, so called, artificial intelligence. All the design and writing were done strictly by the author. The mistakes you find are the result of human limitations.

Appendix A: Literate Programming

The source for this document conforms to [asciidoc](#) syntax. This document is also a [literate program](#). The source code for the implementation is included directly in the document source and the build process extracts the source code which is then given to the Zig compiler. This process is known as *tangleing*. The program, [atangle](#), is available to extract source code from the document source and the asciidoc tool chain can be used to produce a variety of different output formats, although PDF is the intended choice.

The goal of a literate program is to explain the logic of the program in an order and fashion that facilitates human understanding of the program and then *tangle* the document source to obtain the code in an order suitable for a language processor. Briefly, code is extracted from the literate source by defining a series of *chunks* that contain the source. A chunk is *defined* by including its name as:

```
<<chunk name>>=
```

The trailing = sign denotes a definition. A chunk definition ends at the end of the source block or at the beginning of another chunk definition. A chunk may be *referenced* from within a chunk definition by using its name without the trailing = sign, as in:

```
<<chunk definition>>=
  <<chunk reference>>
```

Chunk names are arbitrary strings. Multiple definitions with the same name are simply concatenated in the order they are encountered. There are one or more *root chunks* which form the conceptual tree for the source files that are contained in the literate source. By convention, root chunks are named the same as the file name to which they will be tangled. Tangling is then the operation of starting at a root chunk and recursively substituting the definition for the chunk references that are encountered.

For readers that are not familiar with the literate style and who are adept at reading source code directly, the chunk definitions and reordering provided by the tangle operation can be a bit disconcerting at first. You can, of course, examine the tangled source output, but if you read the program as a document, you will have to trust that the author managed to arrange the chunk definitions and references in a manner so that the tangled output is in an acceptable order.

Appendix B: Target Platform

This section summarizes some of the aspects of the target platform on which the code from the book runs.

Microcontroller architecture

The processor used was specifically chosen to have the ARM® v7e-M architecture. It is the microcontroller CPU architecture features that allow many of the facilities demonstrated in Part I.

SparkFun Micromod Board

The development of the software in the book was done using MicroMod modules from SparkFun. MicroMod is a module system where you pair a processor module with a carrier or function board. For this development, the [SparkFun MicroMod Artemis Processor](#) was used in conjunction with the [MicroMod ATP Carrier Board](#).

The Artemis Processor board uses the Ambiq® Apollo 3 Blue SOC.

For debugging, a SEGGER JLink was used. This required soldering in a 3-pin header the carrier board and supplying a breakout adapter and cable to the make the connection to the standard ARM connector. The adapter and cable are available from [Adafruit](#).

Tool chain

Zig Compiler Version

The Zig code in this literate program was compiled using the following version of the Zig compiler.

Example 6. Zig compiler version

```
0.16.0-dev.254+6dd0270a1
```

Appendix C: Copyright Information

The following is copyright and licensing information. for the original material in this book

```
<<copyright info>>=
//
// The MIT License
//
// This software is copyrighted 2021 - 2026 by G. Andrew Mangogna.
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
// THE SOFTWARE.
```

The following is the copyright information for those parts of the code in this book which were directly or indirectly derived from the Ambiq Software Development Kit (SDK).

```
<<ambiq copyright info>>=
//*****
//
// Copyright (c) 2020, Ambiq Micro, Inc.
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
//
// 1. Redistributions of source code must retain the above copyright notice,
// this list of conditions and the following disclaimer.
//
// 2. Redistributions in binary form must reproduce the above copyright
// notice, this list of conditions and the following disclaimer in the
// documentation and/or other materials provided with the distribution.
//
// 3. Neither the name of the copyright holder nor the names of its
// contributors may be used to endorse or promote products derived from this
// software without specific prior written permission.
```

```
//  
// Third party software included in this distribution is subject to the  
// additional license terms as defined in the /docs/licenses directory.  
  
//  
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"  
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE  
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE  
// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE  
// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR  
// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF  
// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS  
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN  
// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)  
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE  
// POSSIBILITY OF SUCH DAMAGE.  
  
//  
//*****
```

Appendix D: Edit Warning

We want to make sure to warn readers that the source code is extracted and not manually written.

```
<<edit warning>>=
//  
// DO NOT EDIT THIS FILE!  
// THIS FILE IS AUTOMATICALLY EXTRACTED FROM A LITERATE PROGRAM SOURCE FILE.  
//
```

Index

C

const
AllocationStatus, 509
Bucket, 521
bucket_count, 521
Cell, 931
default_load_factor, 415
DependentAttrProperties, 645
Index, 518
ProbeResult, 524
ProbeStatus, 524
pub
codomain_capacity, 571
codomain_constraint, 571
CodomainTupleRefSet, 572, 591
domain_capacity, 571
domain_constraint, 571
DomainTupleRefSet, 571
emptyCodomainTupleRefSet, 572, 591
emptyDomainTupleRefSet, 572
PartitionId, 589
RelationMatrix, 572
RelSortContext, 496
reltuple, 505, 639
relvalue, 505, 639
relvar, 639
TransposeMatrix, 572

D

data
 external
 systemCoreClock, 42
data type
 CopySegmentDesc, 35
 ElfNoteSectionDesc, 52
 InitSegmentDesc, 36

E

enum
 pub
 RefConstraint, 583
error
 RelValueError, 417
 RelVarError, 514

F

file
 path-to-main-test.zig, 57
 start_main.zig, 45
 startup_apollo3.zig, 28
fn
 activate, 516

cloneCapacity, 420
coalesceHorizontal, 944
coalesceTableHorizontal, 944
coalesceTableVertical, 945
contains, 428
createCapacity, 419
DefineGenAdjMatrix, 742
deinit, 937
destroy, 421
draw, 931
drawAttributeInfo, 948
drawBannerRow, 947
drawBottomBorder, 934
drawFill, 932
drawHeading, 947
drawLeftBorder, 933
drawRightBorder, 934
drawTable, 946
drawTableBottomBorder, 950
drawTableRightBorder, 951
drawTopBorder, 933
drawTupleBody, 948
drawUnusedCells, 950
drawValue, 932
drawValueBody, 949
evalConsistency, 583
formatTable, 930
free, 516
GroupHeading, 484
IndexEqFunction, 522
IndexHashFunction, 522
init, 936
insertTuple, 422
isCodomainConsistent, 584
isDependentAttr, 645
isDomainConsistent, 584
Iterator, 499
layoutAttrNames, 940
layoutAttrType, 941
layoutBannerRow, 939
layoutHeading, 939
layoutRelTuple, 935
layoutRelValue, 936
layoutTupleBody, 941
layoutValueBody, 942
lineTerminate, 951
locate, 946
markHorizontal, 945
pub
 AssociationAdjMatrix, 570
 classify, 597
 format, 586
 GeneralizationAdjMatrix, 588

image, 580, 602
 imageComplement, 580
 isAssociationConsistent, 585
 isGeneralizationConsistent, 604
 preimage, 580, 602
 preimageComplement, 581
 printGenMatrices, 605
 reclassify, 599
 relate, 574, 595
 relrelateCodomainTuple, 579
 rereleaseDomainTuple, 578
 unrelate, 575, 595
 unrelateCodomainTuple, 577, 597
 unrelateDomainTuple, 576, 597
 unrelateSubclassInstances, 747
 unrelateSuperclassInstances, 746
RelVarIndex, 520
reorderTuple, 430
scalarValueWidth, 942
sizeHeading, 937
sizeTupleBody, 938
sizeValueBody, 939
SortLessThanFn, 497
squareUp, 943
TcloseHashContext, 492
totalWidth, 943
TupleHashContext, 416
tupleNeedsReordering, 429
UngroupHeading, 487
validateDependentAttrs, 642
validateGeneralizationSubclasses, 741
validateIdentifiers, 507
validateRelVarHeading, 506
validateTupleHeading, 374
function
 external
 main, 164
 STIMER_IRQHandler, 32
weak
 systemAbEnd, 24

I

inline
fn
 nextBucket, 524

P

pub
const
 AttributedId, 376, 414, 508
 Builder, 420
 CatalogEntry, 961
 CatalogSpec, 960
 CatalogTag, 960
 ClassGeneratedId, 640
 dependent_attr_default, 641

DependentAttrFormula, 640
 EmptyHeadingRelValue, 418
 GenerationNumber, 510
 InstanceRef, 647
 InstanceSet, 659
 len, 961
 Loading, 520
 RelationCapacity, 417
 RelationTable, 935
 RelFormatter, 928
 relvalue_allocator, 412
 RelValueType, 508
 relvar_capacity, 508
 relvar_identifiers, 508
 SequenceCount, 520
 SubclassId, 742
 SuperclassInstanceSet, 745
 TaggedCatalog, 960
 Tuple, 414, 508
 tuple_degree, 376
 TupleBase, 414, 508
 TupleRef, 529
 TupleRefSet, 537

fn

activateReserved, 652
 activateReservedOrError, 652
 annihilateSelector, 449
 Association, 684
 attributeEql, 380
 attributeLessThan, 497
 attributeName, 377, 423
 attributes, 377, 423
 attributeSlice, 425
 AttributeType, 377
 attributeUpdater, 471
 body, 427
 cardinality, 423, 542, 659, 661
 classify, 749
 comparativeSelector, 452
 complement, 662
 conjunctiveComparativeSelector, 452
 conjunctiveEqualitySelector, 450
 contains, 665
 create, 378, 417, 650
 createEmpty, 418
 createOrError, 650
 createReserved, 651
 createReservedOrError, 651
 dee, 419
 degree, 377, 423, 542, 659
 deinit, 929
 destroy, 662
 disjointUnion, 446
 disjunctiveComparativeSelector, 453
 disjunctiveEqualitySelector, 450
 DomainTupleRefSet, 592
 dum, 418

empty, 428, 661
emptyDomainTupleRefSet, 592
eq, 381, 430, 664
equalitySelector, 450
equivalentHeading, 379
extend, 392, 464
ExtendHeadingMulti, 394
ExtendHeadingSingle, 391
extendMany, 395
extract, 383, 427
formatRelTuple, 929
formatRelValue, 929
Generalization, 740
get, 962
getPtr, 962
getPtrConst, 963
group, 481
groupedJoin, 460
identitySelector, 449
incrAttributeUpdater, 471
init, 928, 961
insertFromRelValue, 554, 654
intersect, 441, 663
isConsistent, 698, 747
isConstraintConditional, 686
isRelValueType, 415
isRelVarType, 509
iterator, 498, 963
join, 406, 455
JoinHeadings, 405
limit, 662
manyAttributeUpdater, 471
minus, 442, 663
notEmpty, 428, 661
populate, 545
project, 447
properSubSetOf, 435
properSuperSetOf, 433
put, 963
rank, 468
readAttribute, 558, 673
reclassify, 749
relate, 691, 745
RelClass, 641
relExprAlloc, 413
relExprBegin, 412
relExprEnd, 412
RelTuple, 374
RelValue, 413
relValueFromTuple, 408
relValueFromTupleSet, 552
relValueOf, 672
RelVar, 505
rename, 389, 476
RenameHeading, 388
reserve, 514
reserveAndUpdate, 515
restrict, 449
restrictAllComparison, 453
restrictAllEqual, 451
restrictAtLeastOneComparison, 454
restrictAtLeastOneEqual, 451
restrictComparison, 453
restrictEqual, 451
select, 550, 668
selectAll, 547, 667
selectAllComparison, 551, 670
selectAllEqual, 551, 669
selectAnyComparison, 552, 670
selectAnyEqual, 551, 669
selectByIdentifier, 547, 667
selectComparison, 551, 669
selectEqual, 550, 668
SubclassInstanceSet, 745
subSetOf, 434
superSetOf, 432
symmetricDifference, 444
tag, 466
tclose, 489
traverseFromSubclass, 750
traverseToSubclass, 750
tupleFromRelValue, 427
ungroup, 485
unionInstanceSet, 663
unrelate, 746
unwrap, 402, 478
UnwrapHeading, 401
update, 384, 470
updateAllAttributes, 472
updateAttribute, 557, 674, 675
updateFromRelValue, 554, 654
updateMany, 385
updateManyWhereAllEqual, 474
updateManyWhereAtLeastOneEqual, 475
updateManyWhereEqual, 473
updateOrInsertFromRelValue, 555, 654
updateReserved, 515
updateWhereAllEqual, 473
updateWhereAtLeastOneEqual, 474
updateWhereEqual, 472
validateReflexiveConstraints, 685
wrap, 399, 478
WrapHeading, 398
pub fn unionRelValue, 439

T

test

adjacency matrix initialization, 574
any to any, 737
any to at-most-one, 729
any to one, 720
array from relation value, 426
association image and preimage, 581

association tests, 706
at-least-one to any, 734
at-least-one to at-least-one, 731
at-least-one to at-most-one, 725
at-least-one to one, 717
at-most-one to at-most-one, 723
at-most-one to one, 714
attribute access tests, 558, 675
class creation, 655
class initialization, 646
create relation value, 422
creating as TupleRef, 535
creating as TupleRefSet, 545
destroy tests, 666
disjoint union relation value, 447
dum / dee relvalue formatting, 958
extend relation value, 465
generalization consistency, 604, 748
generalization image and preimage, 603
generalization initialization, 594, 744
generalization printing, 606
generalization reclassification, 601
generalization relating, 596
generalization traversal, 752
generalization unrelating elements, 598
group relation value, 484
grouped join relation value, 462
init RelVar, 517
intersect relation value, 442
iteration, 499
join relation value, 456
minus relation value, 443
multiple generalization tests, 606, 753
one to one association, 712
project relation value, 448
reflexive associations, 709
relating elements, 575
relation value equality, 431
relation value proper subset, 435
relation value proper superset, 433
relation value subset, 434
relation value superset, 432
relation-valued attribute formatting, 956
relValueFromTupleSet tests, 553
relValueOf tests, 672
rename relation value, 477
restrict relation value, 454
selection tests, 671
symmetric difference relation value, 445
tclose relation value, 492
tuple create, 378
tuple equal, 381
tuple extend, 393
tuple extendMany, 396
tuple extract, 384
tuple formatting, 951
tuple join, 407

tuple project, 388
tuple rename, 390
tuple unwrap, 403
tuple update, 385
tuple wrap, 400
union relation value, 440
unrelating instances, 577
update relation value, 475
updateFromRelValue tests, 556
value formatting, 953
wrap/unwrap relation value, 479

V

var
allocator_instance, 412
child_allocator, 412

Z

Zig version, 974