

Adamant

User Guide



Created at:
Laboratory for Atmospheric and Space Physics
University of Colorado
Boulder, Colorado

Contents

1 About this Guide	1
1.1 Purpose	1
1.2 Using the Guide	1
2 Overview of Adamant	2
2.1 Motivation	2
2.2 Component-based Design	3
2.3 Model-driven Development	3
2.4 Ada as Implementation Language	6
3 Getting Started	9
3.1 Project Setup	9
3.1.1 Project Directory	9
3.1.2 Virtual Development Environment	10
3.1.3 Project Configuration	10
3.2 Adamant File Structure	13
3.3 Using the Build System	14
3.3.1 Using Redo	15
3.3.2 The Build Target	19
3.3.3 The Build Path	22
4 Basic Examples	24
4.1 Hello World	24
4.2 Creating a Package	26
4.3 Testing a Package	27
4.3.1 A Simple Unit Test	27
4.3.2 Unit Testing with Adamant	28
4.3.3 Unit Test Logger	34
4.3.4 Unit Test Coverage Report	36
4.3.5 Unit Test Documentation	37
4.4 Object-oriented Design	38
4.4.1 Creating an Object-oriented Package	38
4.4.2 "White-box" Unit Testing	40
5 Packed Types	44
5.1 Packed Records	44
5.1.1 Packed Record Ada Specification	46
5.1.2 Packed Record Type Conversion	50
5.1.3 Packed Record Serialization	50
5.1.4 Packed Record Validation	51
5.1.5 Packed Record String Representation	53
5.1.6 Packed Record Unit Test Assertions	56
5.1.7 Packed Record Documentation	58
5.1.8 Packed Record Python Class	60
5.1.9 Packed Record MATLAB Class	64
5.1.10 Nested Packed Records	71
5.1.11 Variable Length Packed Records	71
5.1.12 C/C++ Packed Records	73
5.2 Packed Arrays	74
5.2.1 Packed Array Ada Specification	75
5.2.2 Packed Array Type Conversion	78
5.2.3 Packed Array Serialization	78
5.2.4 Packed Array Validation	78
5.2.5 Packed Array String Representation	78
5.2.6 Packed Array Unit Test Assertions	80
5.2.7 Packed Array Documentation	80

5.2.8	Packed Array Python Class	81
5.2.9	Packed Array MATLAB Class	81
5.2.10	Nested Packed Arrays	81
5.2.11	C/C++ Packed Arrays	81
5.3	Enumerations	81
5.3.1	Enumeration Specification	82
5.3.2	Enumeration String Representation	83
5.3.3	Enumeration Unit Test Assertions	84
5.3.4	Enumeration Documentation	86
5.3.5	Using Enumerations In Packed Types	87
5.3.6	Enumeration Python Class	88
5.3.7	Enumeration MATLAB Class	90
5.4	Packed Registers	91
5.4.1	Volatile, Atomic, and Volatile Full Access in Ada	92
5.4.2	Packed Register Type	93
5.4.3	Packed Register Set	99
5.4.4	Packed Register Arrays	103
6	Components	111
6.1	Creating a Passive Component	111
6.1.1	Creating a Component Diagram	114
6.1.2	Anatomy of a Component	114
6.1.3	Component Base Package	116
6.1.4	Component Implementation Package	120
6.2	Unit Testing a Passive Component	124
6.2.1	Anatomy of a Component Unit Test	125
6.2.2	Creating a Unit Test Model	126
6.2.3	The "Reciprocal" Tester Component	129
6.2.4	Implementing the Unit Tests	133
6.2.5	Component Unit Test Logger	142
6.2.6	"White-box" Unit Testing	148
6.2.7	Unit Test Coverage Report	152
6.2.8	Unit Test Documentation	154
6.3	Creating a Queued Component	155
6.3.1	The Component Queue	155
6.3.2	Implementing a Queued Component	156
6.3.3	Unit Testing a Queued Component	167
6.3.4	Implementing a Priority Queued Component	172
6.4	Creating an Active Component	174
6.4.1	The Component Task	174
6.4.2	Implementing an Active Component	177
6.4.3	Unit Testing an Active Component	179
6.5	Creating a Background Component	183
6.5.1	Implementing a Background Component	183
6.5.2	Unit Testing a Background Component	188
6.6	Protected Objects within Components	191
6.7	Component Initialization	191
6.7.1	Component Instantiation	192
6.7.2	Component Base Initialization	195
6.7.3	Component Set ID Bases	196
6.7.4	Component Map Data Dependencies	196
6.7.5	Component Implementation Initialization	197
6.7.6	Component Set Up	201
6.8	Component IDed Entities	201
6.9	Component Events	202
6.9.1	The Event Type	203
6.9.2	Creating an Event Model	203

6.9.3	Using the Event Package	207
6.9.4	Event Unit Testing	208
6.9.5	Event String Representation	210
6.9.6	Event Documentation	211
6.10	Component Data Products	211
6.10.1	The Data Product Type	211
6.10.2	Creating a Data Product Model	212
6.10.3	Using the Data Product Package	216
6.10.4	Data Product Unit Testing	216
6.10.5	Data Product String Representation	218
6.10.6	Data Product Documentation	219
6.11	Component Data Dependencies	219
6.11.1	The Data Product Fetch and Return Types	220
6.11.2	Creating a Data Dependency Model	221
6.11.3	Fetching Data Dependencies	225
6.11.4	Data Dependency Unit Testing	232
6.11.5	Data Dependency Documentation	235
6.12	Component Packets	235
6.12.1	The Packet Type	235
6.12.2	Creating a Packet Model	236
6.12.3	Using the Packet Package	241
6.12.4	Packet Unit Testing	242
6.12.5	Packet String Representation	243
6.12.6	Packet Documentation	244
6.13	Component Commands	245
6.13.1	The Command Type	245
6.13.2	Creating a Command Model	246
6.13.3	Implementing Command Handlers	249
6.13.4	The Command Response Type	251
6.13.5	Command Unit Testing	252
6.13.6	Command Documentation	254
6.14	Component Parameters	255
6.14.1	The Parameter Type	255
6.14.2	The Parameter Update Type	256
6.14.3	Creating a Parameter Model	257
6.14.4	Updating and Using Parameters	260
6.14.5	Parameter Unit Testing	264
6.14.6	Parameter Documentation	266
6.15	Component Faults	266
6.15.1	The Fault Type	267
6.15.2	Creating a Fault Model	267
6.15.3	Using the Fault Package	271
6.15.4	Fault Unit Testing	272
6.15.5	Fault Documentation	274
6.16	Component Documentation	274
6.16.1	Component Requirements	275
6.16.2	Component Design Document	276
6.17	Component Metrics	279
6.18	Advanced Component Topics	279
6.18.1	Generic Components	279
6.18.2	Component Interrupt Handling	282
6.18.3	Component Subtasks	286
6.18.4	Assembly-parameterized Components	289
7	Assemblies	291
7.1	Creating an Assembly	291
7.1.1	The Example Components	291

7.1.2	Modeling an Assembly	296
7.1.3	Creating an Assembly Diagram	306
7.1.4	Running an Assembly	306
7.2	Subassemblies	308
7.3	Assembly IDed Entities	309
7.3.1	The Example Component IDed Entity Models	309
7.3.2	IDed Entity ID Assignment	311
7.3.3	IDed Entity Ada Specification	313
7.3.4	IDed Entity Documentation	314
7.4	Assembly Views	315
7.4.1	Creating a View Model	315
7.4.2	Using View Rules	320
7.4.3	Configuring View Graph Layout	322
7.5	Assembly Documentation	324
7.5.1	Assembly HTML	324
7.5.2	Assembly Design Document	327
7.6	Assembly Metrics	329
7.7	Assembly Ground System Integration	329
7.7.1	OpenC3 COSMOS Integration	330
7.7.2	Hydra Integration	330
8	Ground Tools	332
9	Advanced Topics	333
9.1	Interfacing with C and C++	333
9.2	IDE Integration	335
9.3	Coding Style	336
9.4	Static Analysis	338
9.5	Code Metrics	341
9.5.1	SLOC and Cyclomatic Complexity	341
9.5.2	YAML SLOC	347
9.5.3	Build Artifacts	348
9.6	Creating a SPARK Package	349
9.7	The Last Chance Handler	351
9.8	Creating a Register Map	352
9.9	Creating a Memory Map	356
9.10	Advanced Build System Topics	362
9.10.1	Modifying the Build Path	362
9.10.2	Debug Mode	364
9.10.3	Setting the Environment	364
9.10.4	Adding a Build Target	365
9.10.5	Extending the Build System	366
9.10.6	Model Caching	366
9.11	The Generator System	367
9.11.1	Anatomy a Generator	367
9.11.2	Adding a Generator	368
A	Appendix A Model Schemas	370
A.1	Assembly Model Schema	370
A.2	Commands Model Schema	374
A.3	Component Model Schema	375
A.4	Data Dependencies Model Schema	380
A.5	Data Products Model Schema	380
A.6	Enumeration Model Schema	381
A.7	Events Model Schema	382
A.8	Faults Model Schema	383
A.9	Packed Array Model Schema	383

A.10 Packed Record Model Schema	384
A.11 Packets Model Schema	386
A.12 Parameters Model Schema	387
A.13 Requirements Model Schema	388
A.14 Unit Test Model Schema	388
A.15 View Model Schema	389
A.16 SPARK Prove Model Schema	391
A.17 Memory Map Model Schema	391

1 About this Guide

1.1 Purpose

Adamant is a component-based, model-driven framework designed to construct reliable and reusable embedded, real-time software systems. The purpose of this document is to discuss how to use the Adamant framework to design and implement a software system for a particular project. The intended audience of this document is embedded software engineers who want to understand how to use Adamant to develop their systems. It is expected that readers of this document have already read the [Architectural Description Document](#) for Adamant to ensure understanding of the architectural principles which govern the framework. This User Guide expands on the Architectural Description Document by providing details of how the architecture is implemented and most importantly, how it can be used. This document also assumes reader knowledge of object-oriented design and its common patterns. A cursory understanding of Ada, the implementation language of the Adamant framework, is also expected in order to understand the examples within fully.

1.2 Using the Guide

The beginning of this guide provides an overview of Adamant, explains why it exists, and presents its governing principles. The remainder of the guide is written as a set of tutorials, with background information and design rational provided along the way. The best way to use this guide is to set up the Adamant development environment on your machine, see Section 3, and then follow the tutorials by implementing the examples yourself, or using them as inspiration for your own creations. It is not necessary to read through this guide linearly. Feel free to use the table of contents and jump to sections that interest you most, or are most pertinent to the work you are trying to accomplish. It is recommended, however, that you have a good understanding of the concepts presented in Sections 3 and 4 before you begin working.

All of the examples provided in this guide exist in the Adamant repository in *doc/example_architecture*. The examples are all compilable, and thus can be explored and run by the reader to gain more understanding. This has the dual benefit of making sure that the examples in this guide are valid syntactically and provides the reader ready-made programs that can be copied and used as a starting point for their own work.

The User Guide itself lives in *doc/user_guide* of the Adamant repository. The L^AT_EX source can also point you to the exact location of examples used in this document. The guide can even be compiled (once you set up the Adamant development environment), which will, in-turn, compile all the examples for you by running:

```
> cd doc/user_guide  
> redo build_user_guide
```

within the Adamant repository.

Enjoy, and I hope that this document proves most useful to you.

2 Overview of Adamant

2.1 Motivation

Adamant was created to serve two purposes:

1. Provide a technical solution that addresses the unique challenges of spacecraft flight software, with a focus on performance
2. Allow for direct, verbatim code reuse from project to project to reduce development costs and increase reliability

Both topics are touched on here, but it is suggested that the reader fully understand the concepts presented in the Adamant [Architectural Description Document](#), which provides thorough motivation and formulation of this framework.

Spacecraft computing platforms are required to operate under different constraints than their earth-bound counterparts. Spacecraft are often required to function for long periods without contact from an operator. Robustness is paramount since physically accessing the system for repair is almost always cost-prohibitive. Space avionics must also deal with radiation and charged particles, which can cause unexpected bit flips and latch-ups. As a result, the embedded flight software that runs on these avionics usually executes on radiation hardened processors with a memory footprint and execution speed many orders of magnitudes smaller than is expected of terrestrial software. The software must also be extremely reliable and autonomous, able to deal with problems that may arise, without intervention from a human operator.

Flight software for spacecraft usually serves one of two tasks 1) fast rate deterministic control of the physical system needed for tasks like pointing the spacecraft, firing thrusters, and maintaining a safe thermal environment or 2) orchestrating the spacecraft's high level mission, performing many disparate tasks such as collecting data from instruments, communicating with a ground system, and scheduling various onboard activities at specific times. The first task usually requires a high performance, deterministic thread of execution to meet the mission's control requirements. The second task usually mandates the use of a system tailored for efficient multi-tasking. These two purposes are at odds, and it often makes sense to separate a flight software system into two different execution domains, one which provides high rate control and the other which performs multi-tasking and high level orchestration of the system.

The goal of Adamant is to provide a system that is both efficient and deterministic enough to be used for high rate control, but also expressive enough to act as the multi-tasking "brain" of a spacecraft. For cost-conscious spacecraft developers, such a system is enticing as it can simplify both the hardware and software design.

Another driver of cost in the development of flight software is the reuse of code from project to project. Spacecraft are designed to perform a wide variety of missions. Each necessitates a different set of requirements and hardware to fulfill that mission. Because of this variation in requirements, the software for each spacecraft must also change for each mission. However, there are many functions that almost all spacecraft share, such as command handling, pointing control, thermal management, gathering and packaging telemetry for downlink, etc. Because of this shared functionality, with careful design, there is an opportunity to reuse a significant amount of software from one mission to the next. By directly reusing software without modification, all testing and validation performed by previous missions are leveraged. If done effectively, the reuse of code and the associated validation can compound from project to project, not only to decrease development cost but also to increase the reliability of the software.

Adamant addresses the unique challenge of creating reusable and reliable embedded software for use in space through the application of two principles: 1) component-based design and 2) model-driven

development, which are discussed in the following sections.

2.2 Component-based Design

In Adamant, an engineer considers the design of their system by breaking it apart into a set of logical units called *components*. The first step in the design process is performing this decomposition and determining the interfaces necessary to facilitate communication between these components. The function of components can be compartmentalized into those that are mission-specific and those that are generic and can be reused on future projects. The details of the Adamant component-based architecture is presented in the Adamant [Architectural Description Document](#). Understanding the architectural concepts presented in that document is required before proceeding to use the remainder of this User Guide.

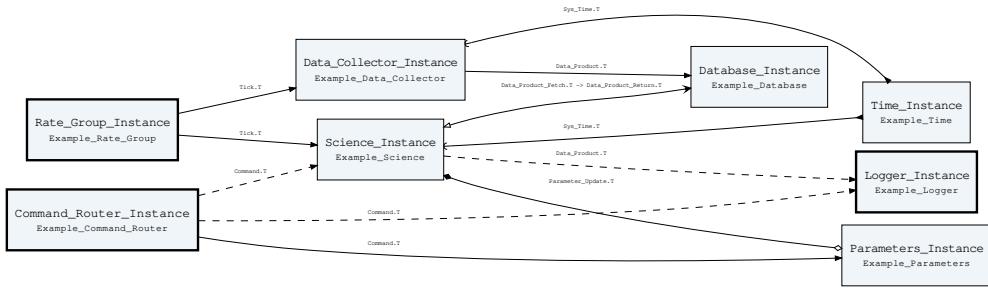


Figure 1: Adamant is a component-based framework, where logical units called *components* communicate with each other through strongly typed *connectors*.

2.3 Model-driven Development

Adamant encourages software engineers to carefully consider requirements and high level design before diving into the implementation of their system. The primary mechanism for achieving this in Adamant is through models. All models are input by the user through a specification language written in [YAML](#) format. These models are then used to automatically generate a variety of products that aid in the development of the software. These products include autogenerated “structural” code for the system, unit test harnesses, interface specifications, documentation, and visual diagrams.

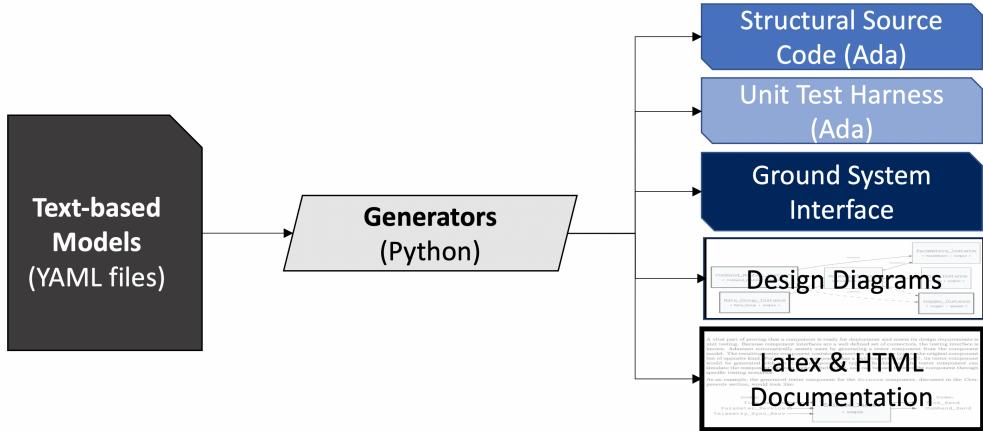


Figure 2: Adamant employs model-driven development. The system is modeled in text-based YAML files, and python generators are used to produce a variety of output products including autogenerated “structural” code for the system, unit test harnesses, interface specifications, documentation, and visual diagrams.

The use of models within Adamant formalizes the traditional approach to software development. To build up a system using Adamant, an engineer first breaks down their system into *components*,

as described in the [Architectural Description Document](#). This is often conducted on a whiteboard or piece of paper prior to codifying the design in the Adamant modeling language. Part of this process is carefully defining the set of *packed types* that can be transmitted over specific connectors between components, see Section 5. Once these types are modeled, a component can be modeled, and implementation and unit testing can begin, see Section 6. Using this process, engineers can build up an entire system of well-tested components and eventually assign these components to particular assemblies using an assembly model, see Section 7. At this point, the assemblies can be compiled and executed, and acceptance testing can be performed on the integrated software to ensure that it meets the requirements for the project.

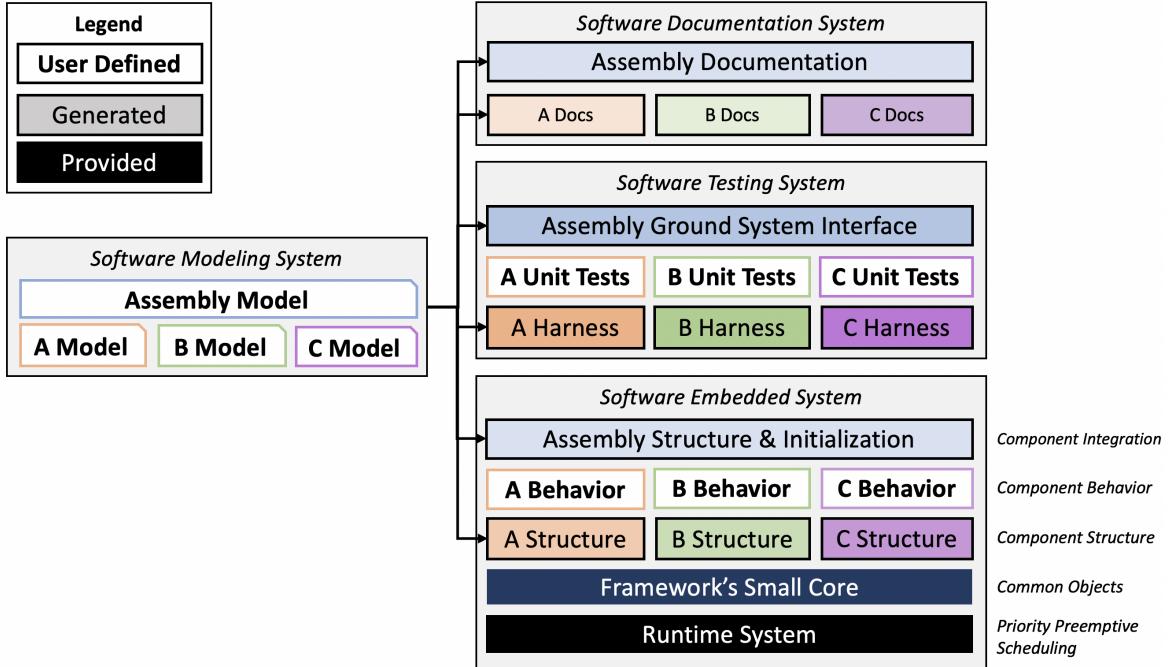


Figure 3: Adamant Architecture Layer Diagram for a simplified 3 component system. Modeling is used to ensure consistency between the embedded system, the testing system, and the software documentation.

The figure above shows an Adamant-based system with only 3 components. In the software modeling system, these 3 components (*A*, *B*, and *C*) and the assembly they are part of are modeled. The assembly describes how the components are interconnected. From these models we can autogenerated the “structural” code for each component that will execute on the embedded system. The autogenerated structural code defines a set of abstract subprograms that must be hand-coded by the engineer to define the component’s behavior. From the assembly model, Adamant can autogenerated code that properly initializes all the components and connects them together. The assembly of components can then execute together on the embedded system on top of the Adamant framework’s small set of core code and the language runtime system, described in more detail in the next section. In this way, the model defines the structure of the software running on the embedded system, and the programmer defines the behavior.

The models are also used to test the software thoroughly. For each component model, Adamant autogenerated a unit test harness. This harness consists of a “reciprocal” component, which serves as an environment for unit testing each component in isolation. Again, Adamant defines the structure of the unit test harness, but the programmer defines the behavior, or in this case, the test cases to run. Using the assembly model, Adamant can autogenerated the interface for a ground system. This interface definition usually consists of the protocol and packet layout for communicating with the embedded system. In this way, the assembly model allows the programmer to begin high-level testing with confidence that the communication protocols on both sides of the system match.

Finally, documentation for each component and the assembly as a whole can be directly generated from the models. A significant amount of information can be understood from these autogenerated documents because of the precise nature of the Adamant modeling system. Human-written documentation can also be added to augment the autogenerated. The documentation is guaranteed to be up to date and reflective of the actual embedded software design because both are generated from the same model source.

Adamant embraces a holistic approach to modeling. Not only are components and assemblies modeled, but data types, commands, telemetry, parameters, unit tests, and even requirements are part of the modeling system. The result is that systems designed using Adamant end up having a significant portion of the underlying code structure automatically generated. It is not uncommon for 70% of the total software for a project to be autocode.

The advantage of using models during the development process is that they ease many of the most challenging aspects of writing good software. Some of the key topics are discussed below.

Effective Communication of Design

Communicating design ideas in software among a group of software engineers is difficult on its own. Presenting these ideas to a broader audience of engineers, which is common in fields like aerospace, is even more challenging. The Adamant modeling system consists of a visual domain-specific language that is described in the [Architectural Description Document](#). This language is simple to understand and explain, even to engineers unfamiliar with the details of software development. The notations in Adamant component diagrams have well defined semantics. The advantage is that when debating solutions to software problems, engineers find it easier to express their ideas, and less time is wasted on misunderstandings.

Complete and Accurate Documentation

Documentation is often neglected in the design of software. Weak or nonexistent documentation is a common problem. This can hinder effective review of the design and makes familiarizing new engineers on a project difficult. Because of Adamant's holistic approach to modeling, adequate documentation can often be generated straight from the model with a single command, see Section 6.16. Furthermore, this documentation is guaranteed to be reflective of the actual implementation since both the code structure and the documentation are generated from the same model.

Interface Control Management

Embedded systems are usually controlled through protocols managed by an Interface Control Document (ICD). For spacecraft, this includes command and telemetry definitions. Since these interface entities are modeled in Adamant, both sides of the interface can be generated using the Adamant modeling and generator system. For instance, if a command is described in an Adamant model, code can be generated in Adamant to allow the system to route and execute that command, see Section 6.13. In addition, the proper command packet definition can be generated for the ground system so that it can send the properly formatted command. In this way, the communication protocols on both sides can be generated from the same model. This ensures consistency and reduces the challenge of integrating two complex systems.

Unit Testing Assistance

Since the interface of a component is known in the model, Adamant can autogenerate a “reciprocal” component that can be used to unit test, see Section 6.2. The “reciprocal” component contains the exact opposite connector definitions of the component under test, and acts as the perfect interface for interacting with the component in an isolated unit testing environment. This property makes unit testing components in Adamant straightforward to implement and understand.

Code Clarity

Using a software framework often requires the developer to write a lot of boilerplate code to follow the design patterns for that framework. Too much boilerplate code is frowned upon because it makes code less readable, and is often prone to copy and paste errors. Models allow Adamant to manage the entire “structural” layer of the software. That structural layer defines what each component looks like and how the components communicate with each other. All of the code necessary to create this structure is autocoded into an abstract base package. This autocode serves a few important purposes. First, it encapsulates the structural code and, because it is generated, is much less prone to defects. Second, it declares abstract subprograms that the developer must hand-code to implement the component’s behavior. This ensures that what the designer specifies in the model truthfully gets implemented in the code. Any deviation results in a compilation error. Finally, this process results in hand-code that has a high level of clarity, as it only shows the behavioral logic of the component. The structural code, hidden within the autocoded abstract base package, rarely needs to be scrutinized, since the same exact semantics are described in the model and the visual diagrams generated from it. This entire process is described in Section 6.

2.4 Ada as Implementation Language

The implementation language for Adamant is the Ada 2012 programming language. It is recognized that the most significant obstacle in the adoption of the Adamant framework is likely the use of Ada. Since C or C++ are typically chosen for embedded systems, especially in the U.S., the decision to use Ada for Adamant necessitates justification.

The objective of Adamant is to produce reliable and reusable embedded, real-time software. In this theme, the utility of Ada and Adamant align. The comparison of Ada to C-like languages is best summarized by John Barnes in the introduction to [Safe and Secure Software](#):

One of the trends of the second half of the twentieth century was a universal concern with freedom. But there are two aspects of freedom... Maybe A would like the freedom *to* smoke in a pub whereas B wants freedom *from* smoke in a pub. Concern with health in this example is changing the balance between these freedoms. Maybe the twenty-first century will see further shifts from “freedom to” to “freedom from”.

In terms of software, the languages Ada and C have very different attitudes towards freedom. Ada introduces restrictions and checks, with the goal of providing freedom from errors. On the other hand C gives the programmer more freedom, making it easier to make errors.

One of the historical guidelines in C was “trust the programmer”. This would be fine were it not for the fact that programmers, like all humans, are frail and fallible beings. Experience shows that whatever techniques are used, it is hard to write “correct” software. It is good advice therefore to use tools that can help by finding bugs and preventing bugs. Ada was specifically designed for this purpose.

Ada was designed from the ground up for embedded and real-time systems in the 1970s in an effort to promote safe and modular programming. Since then, the Ada language has expanded to include full concurrency, object-oriented features, and a formally provable subset called SPARK.

During Adamant’s inception, the choice of language was a thoroughly debated and researched topic. The implementation of Adamant requires several important features in a programming language. The

features necessary to support the architecture include:

1. A tasking system that contains a priority-preemptive scheduler
2. Object-oriented features, particularly inheritance and abstract classes
3. The ability to create highly reliable software

The following paragraphs explain how Ada fulfills these criteria compared to other languages, such as C/C++.

Tasking Support:

For embedded systems that need tasking support, a real time operating system (RTOS) is generally used such as VxWorks, RTEMS, FreeRTOS, etc. This is necessary in languages like C and C++ because they do not have constructs for tasking contained within the language. Instead, they must call into an RTOS API to spawn tasks and interact between tasks in a thread-safe manner.

In contrast, tasking, task synchronization, and thread-safe data sharing is built directly as first-class syntax into the Ada programming language. An underlying runtime system implements the tasking semantics. The runtimes used by Adamant are based on the [Ravenscar Profile](#), which is a committee-designed profile that defines a subset of the Ada tasking system, designed specifically for safety-critical, hard real-time computing. The use of concurrency is explicit, and thread safety is achieved through lightweight *protected objects*. Unlike C or C++ where mutual exclusion is achieved through semaphores and mutex locks, in Ada, shared variables are encapsulated and declared *protected*. Access to these variables is expressed through the use of read-only functions, write-protected procedures, and synchronized entries. The compiler ensures mutual exclusion and guarantees deadlock-free execution by code insertion, which results in low overhead execution. This implementation produces a runtime that is both deterministic and small, making it straightforward to understand.

Using Ada instead of C/C++ relieves the programmer of needing to choose, understand, configure, and deploy a real-time operating system, most of which are much more complicated than the Ada Ravenscar runtimes. This reduces the complexity of the execution stack, making the system easier to understand and control by developers. It also relieves Adamant from needing an additional operating system abstraction layer (OSAL) to allow it to work with multiple RTOS implementations, which would certainly be in a requirement in a C or C++ implementation.

In regards to Adamant, Ada provides lean tasking support on top of bare-metal hardware using only the Ada language syntax. This simplifies the system, making it easier to manage and configure, and most importantly, easier to validate.

Full Featured:

Adamant requires a fully featured, object-oriented language to provide the proper compile-time guarantees that the modeling language expects. Ada is a very mature language with many of the popular features found in Java and C++. The language fully supports object-oriented design patterns, including inheritance, polymorphism, dynamic dispatching, overriding, and multiple-inheritance via mix-ins. The language is fully modular and supports separate compilation and privatization of code. It also includes features such as generics (similar to templates in C++) and interfaces, which greatly enhance code reusability. All of these features make it suitable for use in a reusable framework such as Adamant.

It should be noted that Ada compilers used with Adamant (usually `gcc`) compile to the same object code as C/C++. Interfacing between the languages is not difficult. Importing C/C++ libraries into Adamant is fully supported.

Because Ada uses the same object code as C/C++, it also uses the same back-end compilation optimizations. This makes Ada execute as fast as C/C++, with the caveat that Ada inserts extra runtime checks into the final executable. These runtime checks implement the semantics of the strong Ada type system, discussed more below. Runtime checks are included to make the code more robust, but can be turned off for performance-critical sections of code if execution time becomes a concern.

Reliable:

Ada is designed from the ground up to support highly reliable computing on embedded systems. The most differentiating feature of Ada compared to other embedded languages is its rich and strong type system. The type system gives the programmer tools to create powerful abstractions. When provided to the compiler, these definitions aid in the automatic detection of many logical and design errors before they can become bugs. Ada provides compile-time and runtime safety checks of variables against their valid ranges for all types upon assignment or type conversion. Valid ranges for any type can be set by the programmer through use of a *range constraint*. This feature allows the Ada compiler and runtime to detect array indexes that overflow, integers that go out of range, or enumeration types that become undefined. Notably, Ada lacks the widespread use of implicit type conversions, which are a common source of bugs in C/C++. An experienced Ada programmer embraces the expressiveness and strictness of the type system, using it to their advantage to write readable, defect-free code.



Ada also provides contracts, preconditions, postconditions, invariants, predicates, and assertions to allow the user to insert their own runtime checking in addition to what the language already provides. These constructs allow programmers to codify their assumptions about program execution, enforce the required state of variables at function call boundaries, and ensure that specific properties are upheld during program flow. These same constructs can also be used in the context of SPARK, a formally verifiable subset of Ada, to *prove* that a program is free of runtime errors and meets its design requirements prior to execution. Adamant supports compilation and formal analysis of SPARK packages natively within the build system. SPARK packages can easily be developed, proved, and included within Adamant components. This allows users to formally prove important subsets of code for their projects. Note that Adamant core packages do not yet make heavy use of the SPARK language subset, but there are plans in the road map to begin converting the core to SPARK.

Note that Adamant also provides support to link in C and C++ libraries. This allows for the clean reuse of legacy software or other external packages.

The learning curve for Ada is not unreasonably steep, especially if you have prior experience in a language like C++ or Java. Some of the best resources for diving into Ada are listed below:

- [AdaCore Learn](#) - an interactive tutorial for learning Ada and SPARK basics
- [Ada for the C++ or Java Developer](#) - a tutorial which teaches Ada features by comparing it to analogous features in C++ and Java
- [awesome-ada](#) - a curated list of awesome resources related to the Ada and SPARK programming language
- [Ada Programming Wiki](#) - a good resource for looking up Ada syntax and features
- [Ada Gems](#) - a set of blog posts demonstrating some of the most powerful and unique features of Ada

3 Getting Started

This section contains what you need to know in order to get started with Adamant, including setting up the development environment and the basics of using the Adamant build system.

3.1 Project Setup

One of the goals of Adamant is to produce highly modular embedded software. This goal does not end with the final deployed system, however. The same principles permeates through the implementation of Adamant and even to how one sets up their project. The Adamant framework itself is currently contained in a single repository that contains the generic code, generators, reusable components, and build system. What is not included in the Adamant repository is a final deployment of embedded software, known in Adamant as an *assembly*. *Assemblies*, by design, are always tailored to a specific project's requirements and target platform. In the Adamant ecosystem, this code is thought of as mission-specific code and thus is always contained in a repository separate from the Adamant framework, which is scoped to be generic and reusable.

When working with Adamant you will always have a minimum of 2 repositories, one that includes the Adamant framework, and another that contains your mission specific deployment, or *assembly*. However, it may make sense to have 3, 4, or more repositories that make up your system. Adamant poses no limit on how you modularize your system, and allows you to tie the compilation of everything together via the *build path*, which is detailed in Section 3.3.3.

Setting up a mission-specific repository is often different for each mission, based on the needs for that mission. Included with Adamant is an Example project which provides a good starting point for most projects. This section details how to set up Adamant with the [example repository](#) in order to get up and running with the system. Once you have had a chance to go through this guide and experiment with the example repository, it is recommended that you make a copy of the example repository and begin modifying it to form your own project repository.

3.1.1 Project Directory

Usually, all repositories for a project are cloned into a single “project directory”. This is not strictly necessary with Adamant, due to the flexibility of the build system, however the steps below present it this way and have been tested. The first step when getting started is to create a project directory and clone the Example and Adamant repositories:

```
> mkdir project # make project directory
> cd project
> git clone https://github.com/lasp/adamant_example.git
> git clone https://github.com/lasp/adamant.git
```

This setup implies a flat structure, where the many repositories are stored within the root `project` / directory. Each repository has a separate history and configuration management. Keeping them in sync can be done via a versioning process, branch naming conventions, or any other myriad of ideas which are not discussed here and should be tailored for your program. Alternatively, one could use git submodules. The complexity of git submodules is not discussed here, but nothing in Adamant precludes you from working in that manner.

Now that we have the directory set up with the repositories, we need to set up the virtual development environment, described in the next section.

3.1.2 Virtual Development Environment

Adamant uses a virtual environment for development for a few reasons. First, using a virtual environment makes it easy to install all the necessary dependencies without cluttering up the host system. Secondly, it bypasses the classic "it works on my machine" problem that is so prevalent in software development, since all developers are using the same exact machine.

Developers can be picky about the tools they install, the editors they use, and whether or not to use spaces or tabs (you should use spaces!). Adamant does its best to not force the use of any tool on any developer, with the exception of using `redo`, discussed in Section 3.3.1. To accomplish this, the Adamant virtual environment shares the project directory, discussed in the last section, between the host and the virtual machine. In this way, the developer can install whatever tools they want on either the host or the virtual machine, to interact with the system how they please. The only requirement is that you must run build commands, ie. `redo`, from within the virtual machine via the virtual environment GUI or an ssh session.

The virtual environment for Adamant is based on Linux, because it is ubiquitous, easy to configure, open source, and widely supported. Currently the virtual environment managed by `Docker`. To ensure that you use the most up to date instructions for setting up the example repository it is recommended you follow the repository specific directions stored in [this README.md](#). Please follow those instructions to bring up your machine.

3.1.3 Project Configuration

Adamant provides a single configuration file that can be used to tailor the framework for a specific project. The default version of this configuration file is stored in `config/adamant.configuration.yaml.original` and is shown below:

`adamant.configuration.yaml.original`:

```
1 # adamant.configuration.yaml
2 #
3 # This file provides global definitions for Adamant. These variables are
4 # used to configure Adamant for a specific project. Changeable entities
5 # include the sizing of different core data types, etc.
6 #
7 # Note: This file is not meant to be checked into a version control system
8 # in this location, as it will be different for every usage of Adamant.
9 # Instead consider version controlling it inside of your project repository
10 # and copying it into this location on startup. Alternatively, the Adamant
11 # build system looks for the location of this file using the environment
12 # variable ADAMANT_CONFIGURATION_YAML. If you set that variable to a
13 # configuration
14 # file stored in your project repository, then Adamant will load your desired
15 # variables.
16 #
17 # Variables defined below should be simple YAML maps, mapping a string key to a
18 # numeral value.
19 #
20 # To reference these variables within an Adamant YAML file you need to use Jinja
21 # format for variable reference: https://jinja.palletsprojects.com/en/2.11.x/
22 # For example to use value of the data_product_buffer_size variable within your
23 # YAML
24 # you would write {{ data_product_buffer_size }} in the location you wish the
25 # substitution to occur.
# 
# To reference these variables within an Ada file you need to "with" the package
```

```

26 # Configuration
27 ---
28 # Description of this configuration file.
29 description: This is the default Adamant configuration.
30 #####] ↵ ##
31 # REQUIRED - Adamant Configuration Values
32 #####] ↵ ##
33 #
34 # Type Sizing Variables - The following are used to size core types within
35 ↵ Adamant
36 # for your project.
37 #
38 # The size of the buffer within the data product type (in bytes). This buffer is
39 # used to serialize the data product type. Choose a size that fits the largest
40 # data product type in your system:
41 data_product_buffer_size: 32
42 #
43 # The size of the buffer within the command type (in bytes). This buffer is
44 # used to serialize the command arguments. Choose a size that fits the largest
45 # command argument in your system:
46 command_buffer_size: 255
47 #
48 # The size of the buffer within the event type (in bytes). This buffer is
49 # used to serialize the event parameters. Choose a size that fits the largest
50 # event parameters in your system:
51 event_buffer_size: 32
52 #
53 # The size of the buffer within the parameter type (in bytes). This buffer is
54 # used to serialize the parameter type. Choose a size that fits the largest
55 # parameter type in your system:
56 parameter_buffer_size: 32
57 #
58 # The size of the buffer within the packet type (in bytes).
59 packet_buffer_size: 1246
60 #
61 # The size of the buffer within the fault type (in bytes). This buffer is
62 # used to serialize the fault parameters. Choose a size that fits the largest
63 # fault parameters in your system:
64 fault_buffer_size: 8
65 #
66 # The size of the buffer within the CCSDS space packet type. If you are not
67 # using CCSDS then you can leave this value as the default.
68 ccsds_packet_buffer_size: 1274
69 #
70 # Other Variables - Various configurations that do not have to do with core
71 ↵ types
72 #
73 # The stack margin in bytes. This value is used in src/core/task/task_util.adb
74 ↵ to
75 # define the amount of "usable" stack used by the Stack Monitor component. You
76 ↵ should
77 # size this value as small as possible, but the value MUST be larger than the
    ↵ amount
    # of stack that a task uses prior to calling the Cycle procedure the first time,
    ↵ and MUST
    # be smaller than that smallest task stack defined in the system. When in doubt,
    ↵ leave

```

```

78 # this value at the default.
79 #
80 # Note, this value only affects bareboard runtimes. Linux uses a predefined
81 # stack margin
82 # of 12KB defined in src/core/task/linux/stack_margin.ads.
83 stack_margin: 1000
84
85 # The command registration delay in microseconds. This value is used by each
86 # commandable component
87 # when registering commands. After sending out the command registration, the
88 # component will sleep
89 # for this many microseconds before registering its next command. The purpose of
90 # this delay is to
91 # reduce the stress on the queue of the component who is receiving the command
92 # registrations,
93 # usually the Command Router. By sleeping, it allows time for the command router
94 # to process the
95 # registration before a new item is put on its queue. If registration items are
96 # falling off the
97 # Command Router's queue at initialization you can increase this sleep time, or
98 # increase the
99 # queue size of the Command Router depending on which is more suitable for your
100 # mission.
101 command_registration_delay: 250 # microseconds
102
103 #####]]#
# OPTIONAL - Custom Project Specific Configuration Values
#####]]#
# Define your own key value pairs below. You will be able to use these
# within your own YAML or Ada files using the methods described at the
# top of this file.
#
# Examples of a project specific variables:
102 project_specific_variable: 17
103 project_specific_string: "This project rocks!"
```

As can be seen, most of the configuration available is changing the size of core types used within the system including commands, events, parameters, etc. As described in the comments, you can also add your own variables within this YAML file and then use them within your source code or YAML model definitions.

The Adamant framework will not function without a configuration file in place. To try out Adamant using the default configuration you can run:

```
> cp adamant.configuration.yaml.original adamant.configuration.yaml
```

You can now modify the values in *adamant.configuration.yaml* to work for your project. Note that *adamant.configuration.yaml* has been added to *.gitignore*, and should NOT be checked into your version control system if you intend on merging changes from your Adamant repository back into the framework root repository.

Version controlling the configuration file is usually desirable. To accomplish this, copy the Adamant default configuration into your project specific repository:

```
> cp adamant.configuration.yaml.original /prj_dir/config/project.configuration.yaml
```

You can now modify `project.configuration.yaml` and version control it within your project directory. To tell Adamant to use this file instead of the default configuration file, you need to set the `ADAMANT_CONFIGURATION_YAML` variable, ie.

```
> export ADAMANT_CONFIGURATION_YAML=/prj_dir/config/project.configuration.yaml
```

It is advisable to include the export command above within your environment files that get sourced whenever a new shell is opened, ie. from `.bashrc`.

A good example for how to set up the configuration file for your project is to look at the [example repository](#) which includes its configuration within the `config/` directory, and sets the `ADAMANT_CONFIGURATION_YAML` variable using the `env/setenv.sh` script.

3.2 Adamant File Structure

This section provides a brief tour of the Adamant repository layout so that a user may more easily find what they are looking for. Adamant is a framework that consists of many parts that all function together to help you write good embedded software. The main constituents are:

- **core code** - hand-written (Ada) code that implements the foundational elements for the Adamant architecture; this code is compiled and run on the embedded target
- **components** - reusable components that can be deployed to the embedded target
- **generators** - the Adamant generator system, used for generating structural autocode, configuration files, diagrams, and documentation
- **build system** - the Adamant build system, used for compiling source code and documentation
- **ground code** - hand-written (python and MATLAB) code that provides tooling to assist in testing the embedded system
- **environment** - files for configuring and provisioning the Adamant development environment

The most important directories in the Adamant repository are listed below with descriptions of what you can find in each:

- **config/** - Configuration files for tuning the Adamant framework to fit a specific project
- **doc/** - Adamant documentation including the Architectural Description Document, this User Guide, the Quick Start Guide, and more
- **env/** - environmental files to aid in provisioning and configuring the Adamant development environment
- **gen/** - Adamant generator system used for generating structural autocode, configuration files, diagrams, and documentation
- **gen/generators** - python generators which tie in with the Adamant build system to create build rules for each generated file
- **gen/models** - python model classes which ingest and calculate all modeling information used by the generators
- **gen/schemas** - YAML schemas used to validate user model files prior to running generators
- **gen/templates** - Jinja2 templates used in outputting various autocoded files
- **gen/test** - unit tests that verify the correct function of the Adamant generators
- **gnd/** - ground code (python) that provides tooling to assist in testing the embedded system

- **src/** - source code intended to be compiled and run on the embedded target
- **src/components** - source code for reusable components
- **src/core** - source code for Adamant core constructs: connectors, components, etc.
- **src/data_structures** - source code implementing common Adamant data structures: queues, trees, etc.
- **src/types** - source code implementing common Adamant types: commands, parameters, etc.
- **src/unit_test** - source code implementing Adamant unit test infrastructure: histories, smart assertions, etc.
- **src/util** - source code implementing various utility packages
- **redo/** - the Adamant redo-based build system, included is the build system core code and build rules for compiling source code, L^AT_EX, Graphviz, and much more
- **redo/rules** - Adamant build rules
- **redo/targets** - supported Adamant build targets

As discussed in Section 3.1 the Adamant repository only contains the generic framework code, and expects a separate repository to use Adamant for a specific project. Provided with Adamant is the Example project which demonstrates how Adamant can be used for a specific application. The most important directories in that repository are listed below with descriptions of what you can find in each:

- **config/** - Adamant configuration file for the example project
- **env/** - files used to provision and configure the Example project environment
- **src/** - example repository source code for deployment to the embedded system
- **src/assembly** - executable assemblies for the Example project
- **src/components** - project-specific components for the Example project
- **docker/** - houses the Docker configuration which spawns the virtual environment for the Example project, also contains directions for setting up your machine

Both repositories contain *README.md* files in many directories that explain their contents further.

3.3 Using the Build System

Adamant uses a top-down, general purpose build tool called `redo`. `redo` is designed in the spirit of the long-lived `make` program, but is much simpler and, surprisingly, much more powerful. See [here](#) for the official documentation on `redo` and [here](#) for the source code of the implementation used within Adamant. You can write custom `.do` scripts for `redo` anywhere in Adamant and expect them to function according to the `redo` documentation. Adamant itself, however, uses a sophisticated python based build system that utilizes `redo` under the hood to manage dependencies and rebuild logic.

The Adamant build system is largely “discovery”-based. There are very few build system configuration files in Adamant, unlike what you might find within most build systems based on `make`, CMake, `scons`, etc. Instead, the build system looks at the filesystem and generates build rules based on the file types that it finds. To support this, Adamant uses specific file naming conventions that are used in the rest of this document. The build system looks for filenames of a specific format and generates rules that operate on that file type. What this means for an Adamant user is less time spent telling the build system how to build their code, and more time spent writing good software.

Similar to python, the Adamant build system works with the concept of a *build path* in the form `/path/to/directory1:/path/to/directory2:etc`. Source code (and model files) found in the build path can see each other during compilation and thus can interact via Ada with statements or other model based import mechanisms to be presented. The *build path* is automatically calculated by `redo` prior to running any `redo` command, so the user does not have to think too much about it. However, the build path can be fine tuned, and may need to be, for various reasons. See Section 3.3.3 for more details.

The Adamant build system also supports compilation and cross-compilation for many different platforms from the same set of source code. This is supported through the use of *build targets*. See section 3.3.2 for more details.

3.3.1 Using Redo

The primary method for interacting with the Adamant build system is through `redo` commands run at the command line. Integration of the Adamant build system within an IDE ecosystem is work that has yet to be started.

`redo` is run by providing it a single argument that is the item to build, also referred to as the *command*. When `redo` is run, the program searches for a build rule that matches your argument, executes it, and tracks any dependencies needed to rebuild it intelligently in the future. If no argument is provided then `redo` assumes you want to build the `all` command. In Adamant, when the `all` command is run, the build system attempts to build any object and most source code that can be constructed within that directory.

It should be noted that most of the discussion below is specific to the Adamant build system, which *uses* `redo`. These are NOT features of `redo` itself, which is a rather simple, but powerful, tool.

Below are some example `redo` commands that you can run within Adamant:

```
> redo                               # run "redo all" in this directory
> redo all                           # run "redo all" in this directory
> redo ../../all                     # run "redo all" two directories above this one
> redo output.txt                   # tell redo to build output.txt in this directory
> redo build/src/file.txt          # tell redo to build file.txt in build/src
```

In Adamant, all autogenerated source code, documentation, diagrams, compiled objects, and compiled executables is constructed in a *build/* directory directly below the YAML model or source used to generate the output. Below is a list of common subdirectories you might find in a *build/* directory and what is contained in each:

- **build/src** - autogenerated source files, which are automatically added to the build path if the directory containing *build/* is in the build path.
- **build/template** - autogenerated source files that are meant to be copied out and used as starting points (ie. "templates") for handwritten code
- **build/obj/Linux** - compiled object files for the Linux target
- **build/obj/Linux_Test** - compiled object files for the Linux_Test target
- **build/bin/Linux** - compiled executable files (*.elf*) for the Linux target
- **build/bin/Linux_Test** - compiled executable files (*.elf*) for the Linux_Test target
- **build/py** - autogenerated python source files

- **build/m** - autogenerated MATLAB source files
- **build/html** - HTML documentation
- **build/pdf** - PDF documentation
- **build/tex** - Latex files used to create PDF documentation
- **build/metric/Linux** - metrics reports for source code used to build a Linux object
- **build/csv** - CSV files
- **build/svg** - Scalar Vector Graphics diagrams
- **build/dot** - Graphviz "dot" specification files for generating diagrams
- **build/eps** - Encapsulated PostScript files used to present images in PDF files
- **build/png** - PNG images

As can be seen, most output files are generated in a directory that contains the file extension for that file type. The *build/obj/* and *build/bin/* directories are unique in that they always contain another set of subdirectories that specify the *build target* that they were compiled for. More details on *build targets* can be found in Section 3.3.2.

The use of *build/* directories also makes “cleaning” fast. To remove all generated outputs you can run:

```
> redo clean      # remove any build/ directories in this directory
> redo clean_all # remove any build/ directories for entire repository
```

One of the most useful *redo* commands in Adamant is *redo what*, which shows you all the possible *redo* commands you can run in your current directory. Below is an example of running *redo what* from Adamant’s *src/core/component* directory:

```
> redo what # show "redo" commands that can be run in this directory
redo what
redo all
redo clean
redo clean_all
redo clear_cache
redo templates
redo publish
redo targets
redo prove
redo analyze
redo style
redo pretty
redo test_all
redo analyze_all
redo coverage_all
redo build/metric/Linux/component.adb.txt
redo build/metric/Linux/component.ads.txt
redo build/metric/Linux/component.o.txt
redo build/obj/Linux/component.o
```

We can see that in this directory, *redo what* has found a lot of valid build commands. We can run commands to compile an object, *build/obj/Linux/component.o*, or even run some code metrics on a

source code file, `build/metric/Linux/component.adb.txt`.

In the Adamant virtual environment, see Section 3.1.2, tab completion is available in Bash for redo commands, which, under the hood, uses `redo what`. This can be used to quickly discover and complete available redo commands without having to type out everything by hand. For instance, running:

```
> redo build/src/<TAB><TAB>
build/src/component-example_component.adb
build/src/component-example_component.ads
```

shows the available Ada files that can be generated via redo within the example component directory.

In addition to specific commands, Adamant also provides a set of build commands that can be used in almost any project directory. These are described below:

- **redo what** - show available redo commands to run in this directory
- **redo all** - build all objects, documentation, and source code (most) that can be built from this directory
- **redo path** - runs `redo all` from all directories in the build path
- **redo recursive** - runs `redo all` from this directory and all below it
- **redo clean** - removes all `build/` directories from this directory and all below it
- **redo clean_all** - removes all `build/` directories from this entire repository
- **redo clear_cache** - remove the persistent model cache that the build system uses to increase performance, see Section 9.10.6
- **redo templates** - finds and builds all autocoded templates (in `build/template/`) that can be built from this directory
- **redo test_all** - finds and runs all unit tests from this directory and all below it. Note: Add a `.skip_test` file to a directory to exclude it from being run by `redo test_all`.
- **redo coverage_all** - finds and runs all unit tests from this directory and all below it and generates coverage reports for each. Note: Add a `.skip_test` or `.skip_coverage` file to a directory to exclude it from being run by `redo coverage_all`.
- **redo publish** - builds all autocoded documentation from this directory and all below it, and copies over the “published” version of that documentation
- **redo targets** - shows a list of all available `build targets`, see Section 3.3.2
- **redo prove** - runs GNATprove to analyze any SPARK code found within the current directory, see Section 9.6
- **redo test** - compiles and runs a unit test - note: this command is only available if a `test.adb`, `test.c(pp)`, or `test.py` file is found in this directory
- **redo run** - compiles and runs a main program - note: this command is only available if a `main.adb` file is found in this directory
- **redo style** - compiles all objects found in this directory with coding style checks enabled
- **redo pretty** - “pretty” formats all hand-written code found in this directory and saves them in `build/template/`
- **redo analyze** - runs a static analyzer on all code found in this directory
- **redo analyze_all** - finds and runs static analysis on all unit tests from this directory and

all below it. Note: Add a `.skip_test` file to a directory to exclude it from being run by `redo analyze_all`.

One unique feature of `redo` is that it prints what it is currently building while it runs, and also shows you each build rule's dependencies via indentation. Below is an example of `redo` being run to generate the component documentation for the `Command_Router` component (run in `src/components/command_router/doc`) Some of the output has been removed for brevity:

```
> redo build/pdf/command_router.pdf
redo  build/pdf/command_router.pdf
redo    ./build/eps/command_router.eps
redo    build/tex/command_router_init.tex
redo    build/tex/command_router_description.tex
redo    build/tex/command_router_events.tex
redo    build/tex/command_router_connectors.tex
redo    build/tex/command_router_requirements.tex
redo    build/tex/command_router_commands.tex
redo    build/tex/command_router_types.tex
redo      ../../types/event/build/tex/event_header.tex
redo      ../../types/event/build/tex/event.tex
redo      ../../types/data_products/build/tex/data_product.tex
redo      ../../types/data_products/build/tex/data_dependency.tex
redo      ../../types/command/build/tex/command_id_status.tex
redo      ../../types/command/build/tex/invalid_command_info.tex
redo      ../../types/command/build/tex/command_id.tex
redo      ../build/tex/command_router_arg.tex
```

As can be seen, building `command_router.pdf` depends on an autogenerated `.eps` diagram and various autogenerated Latex (`.tex`) files. One of these Latex files, `build/tex/command_router_types.tex`, in turn, depends on another whole set of autogenerated Latex files. Using the `redo` output a user can inspect the autogenerated dependencies being built for the command they are running which can aid in understanding how their output is being constructed.

For speed, `redo` can also run build rules in parallel using the `-j` option. Here is an example telling `redo` to use 3 CPUs simultaneously to build:

```
> redo -j3 # run "redo all" using 3 CPU cores
```

Note that when using this option the printed output order is not preserved, so no dependency information can be gleaned from it. Also, parallel builds should be considered an experimental and unstable feature, and should not be used for critical compilations.

For more `redo` options you can run:

```
> redo -h # display help for redo
```

Very rarely, `redo` can get itself into an unknown state. If things are not behaving correctly you can wipe out the `redo` database, and force things to rebuild fresh, which often fixes any issues that arise. To be safe, you can also remove the model cache that the build system uses to increase performance. The best procedure to do this is:

```

> rm -rf ~/.redo      # remove the redo database
> redo clean_all      # clean the entire repository
> redo clear_cache    # remove the model cache if ENABLE_PERSISTENT_MODEL_CACHE is set
> redo <your_command> # retry the redo command that was not working

```

3.3.2 The Build Target

The Adamant build system supports building binary object and executable files for multiple target platforms. For instance, two binaries built from the same source code can be constructed in both a native Linux format and an embedded ARM-based target. Determining which platform a binary is compiled for is determined by the *build target*. In Adamant, the *build target* is a compilation configuration that determines the following:

1. The set of code that is used to compile for the target, ie. *the build path* discussed in the next section
2. The platform for which the code is being compiled, ie. the name of the compiler, linker, and binding programs
3. The compiler configuration determined by compiler options (flags)
4. The Ada *runtime* that the application code will run on top of

The Ada compiler used in Adamant is generally GNAT based `gcc` for Native Linux compiles, and `gcc`-based cross compilers for embedded targets (ie. `arm-eabi-gcc` for ARM). The compiler, compilation flags, and runtime are all determined by the *build target*. The Adamant *build path* determines which source code is compilable for a given *build target* and is discussed in the following section. To add a custom build target for your system see Section 9.10.4 for details.

As background, it is important to understand the Ada runtime system, on top of which an Ada application runs. The Ada runtime provides the tasking logic necessary to implement the Ada language. Adamant utilizes bare metal runtimes provided by AdaCore, which come in the following flavors:

- **ZFP** - The zero-footprint runtime only allows the non-tasking semantics of the Ada language. The resulting program must be single-threaded. This runtime is designed for formal software certification and/or for systems that are severely resource constrained.
- **SFP** - The small footprint runtime implements a restricted subset of the Ada tasking system described by the Ravenscar profile. This runtime provides minimal tasking capabilities, without exception propagation, to make formal software certification feasible.
- **Full** - The full runtime implements a restricted subset of the Ada tasking system described by the Jorvik profile. This profile is a slightly relaxed version of the Ravenscar profile, and allows exception propagation, designed for use on most embedded systems that do not require formal software certification. Note that the Jorvik profile is still very minimal compared to the full Ada language semantics which offers `select` statements, `rendezvous`, etc. These more flexible facilities are still not available in the Jorvik profile.

There is also the native runtime, such as compiling directly for Linux or MacOS, which uses underlying operating system calls to implement the Ada tasking system. Adamant ensures that all software produced adheres to the Ravenscar profile (except for unit tests), such that either an SFP or Full runtime can be used. The ZFP runtime is not currently supported by Adamant components, but can be added if a project needs it in the future.

As previously mentioned, *build targets* also determine the compiler options in effect during compilation. In an attempt to standardize build targets within Adamant, the following table describes the different compiler *modes* used. A compiler mode is simply a standardized set of compilation options, used to make talking about compilation easier among developers.

Table 1: Adamant Standard Compilation Modes

Mode Name	Runtime Mode	Optimization	Validity Checking	Configuration Pragmas	Purpose
Production	Production	-O2 and inlining	RM-defined checks (-gnatVd)	Ravenscar	Used for deployed system in production.
Development	Production	-O2 and inlining	All checks (-gnatVa)	Ravenscar, Initialize_Scalars*	Used for most development testing on target system.
Debug	Debug	-O0	All checks (-gnatVa)	Ravenscar, Initialize_Scalars	Used for debugging hard to track down issues.
Test	Debug	-O0	All checks (-gnatVa)	Initialize_Scalars	Linked with AUnit for unit test.
Coverage	Debug	-O0	All checks (-gnatVa)	Initialize_Scalars	Same as Test with coverage options enabled.

Note that runtimes can be compiled with debug flags (*Debug* mode) or without debug flags and with optimization enabled (*Production* mode). Optimization and back-end inlining is applied to the application code in *Production* and *Development* builds, but is not applied in any other modes. More information on Ada compiler optimization can be found [here](#).

Ada allows the configuration of runtime validity checks within the language. It is advisable to keep the default reference manual checks on in *Production* mode to prevent any undefined behavior. In all other modes, all validity checking provided by GNAT is enabled. This provides much more extensive type checking than the default Ada checking, which can help track down hard to find bugs at the cost of extra code insertion into the binary. More information on validity checks within the Ada compiler can be found [here](#).

Ada also allows the user to restrict the use and behavior of the language by specifying configuration pragmas that apply to all the source code in the compilation. In *Production*, *Development*, and *Debug* mode the Ravenscar profile is applied which restricts the tasking semantics allowed in the source code. This ensures that these builds will work with embedded runtimes, which support limited tasking semantics of the Ada language. Ravenscar is not enforced in *Test* and *Coverage* mode, to enable more elaborate unit testing setups. All modes except *Production* (*and *Development* on bareboard targets) specify the pragma `Initialize_Scalars`, which when used in conjunction with all validity checks enabled (-gnatVa) can greatly aid in the detection of uninitialized variable bugs. Note that this pragma purposely initializes all uninitialized variables to values outside the range for that type in order to force runtime errors for any variables that are used prior to initialization. See this [case study](#) to explore the benefits of `Initialize_Scalars` pragma.

In summary, the *Production* mode should be used for deployed software on target hardware. The

Development mode should be the most commonly used mode, as it should be preferred for most development and testing on target hardware. The *Debug* mode turns off optimization of application and runtime code to make tracking down hard to find bugs easier. The *Test* and *Coverage* modes are designed for unit testing. These modes are an attempt at standardization of build configurations, but of course you can create a custom build target that does not adhere to any of these conventions. See Section 9.10.4 for details.

Adamant *build targets* are generally named in the form *platform_mode*, ie. `Linux_Test` or `ARM_Production`. To configure which target is currently being used you must set the `TARGET` environment variable to the name of the build target you wish to use. If `TARGET` is not set, then the Adamant system assumes you want to build for the native target, and uses the result of the `uname` command as `TARGET`. This is usually "Linux" on a Linux based system. The `TARGET` variable can be configured from the command line using the following commands:

```
> export TARGET=Linux_Test # set the target to Linux_Test
> export TARGET=Linux      # set the target to Linux
> export TARGET=           # unset the target and use the default (`uname`)
```

The target can also be set globally in your shell configuration file (ie. `.bashrc`) or from any local `.do` or `env.py` file. To see the all the supported build targets on your system run:

```
> redo targets
Linux
Description: The default Linux target. This is simply a rename of Linux_Debug.
Project File: /home/user/adamant/redo/targets/gpr/linux_debug.gpr
Path Files: .all_path, .64bit_path, .Linux_Base_path, .Linux_Debug_path, .Linux_path
Python Module: /home/user/adamant/redo/targets/linux.py
Python Class: Linux

Linux_Debug
Description: This native 64-bit Linux target has no optimization, compiles with debug
Project File: /home/user/adamant/redo/targets/gpr/linux_debug.gpr
Path Files: .all_path, .64bit_path, .Linux_Base_path, .Linux_Debug_path, .Linux_path
Python Module: /home/user/adamant/redo/targets/linux.py
Python Class: Linux_Debug

Linux_Test
Description: Same as Linux_Debug except it does not enforce the Ravenscar profile and
Project File: /home/user/adamant/redo/targets/gpr/linux_test.gpr
Path Files: .all_path, .64bit_path, .Linux_Base_path, .Linux_Test_path, .Linux_path
Python Module: /home/user/adamant/redo/targets/linux.py
Python Class: Linux_Test

Pico
Description: This is the default Raspberry Pi Pico microprocessor cross compile target
Project File: /home/user/adamant_example/redo/targets/gpr/pico_development.gpr
Path Files: .all_path, .Pico_Development_path, .Pico_Base_path, .Pico_path, .arm_bare_board
Python Module: /home/user/adamant_example/redo/targets/pico.py
Python Class: Pico

Pico_Debug
Description: This target compiles for the Raspberry Pi Pico microprocessor. It has optimized
Project File: /home/user/adamant_example/redo/targets/gpr/pico_debug.gpr
Path Files: .all_path, .Pico_Debug_path, .Pico_Base_path, .Pico_path, .arm_bare_board
Python Module: /home/user/adamant_example/redo/targets/pico.py
Python Class: Pico_Debug
```

```

Pico_Development
Description: This target compiles for the Raspberry Pi Pico microprocessor. It has op
Project File: /home/user/adamant_example/redo/targets/gpr/pico_development.gpr
Path Files: .all_path, .Pico_Development_path, .Pico_Base_path, .Pico_path, .arm_ba
Python Module: /home/user/adamant_example/redo/targets/pico.py
Python Class: Pico_Development

Pico_Production
Description: This target compiles for the Raspberry Pi Pico microprocessor. It has op
Project File: /home/user/adamant_example/redo/targets/gpr/pico_production.gpr
Path Files: .all_path, .bb_path, .Pico_Base_path, .Pico_path, .arm_bare_board_path,
Python Module: /home/user/adamant_example/redo/targets/pico.py
Python Class: Pico_Production

```

which produces a list of all available targets. A *build target* is made up of two parts, the GPRBuild project file (.gpr) which defines the runtime, compiler, and compiler options, and the *path files* which determine the source code that applies for that target. These two parts are tied together by a python module, whose location is shown for each build target. For more details on how a *build target* is constructed see Section 9.10.4.

3.3.3 The Build Path

When writing software for embedded systems it is common to have different versions of source code that only works on certain hardware, such as device drivers. It is desirable to have a method for specifying the source files used when compiling for different platforms. In Adamant, this is supported by configuring the *build path* which is calculated based on the *build target*, see Section 3.3.2.

The Adamant build system uses a *build path* in the form /path/to/directory1:/path/to/directory2:etc in order to locate the appropriate source, object, and model files for compiling and linking executables. The build path for a specific redo command can be configured in many different ways. By default, when any redo command is invoked, the build path is calculated. The build system traverses your project, looking for presence of specific *path files*, and uses the following rules to construct the build path:

1. Any directory found with a .all_path file present is added to the build path.
2. Any directory found with a .TARGET_path file present, where TARGET is the name of the currently set build target (ie. the TARGET environment variable), is added to the build path.
3. Any directory found with a *path file* present that is specified in the build target's path_files definition (see Section 9.10.4) is added to the build path.

Path files are always empty, as their contents serve no purpose. The build system only uses the presence of these files to determine the build path. *Path files* should be checked in and tracked in your version control system. *Path files* allow the user to easily separate platform-specific code from more portable platform-agnostic code. In general, if you are not writing a hardware-specific software package, adding a .all_path to the directory where your source code lives will add it to the build path and allow the build system to compile and link it with other packages. If you are writing piece of software that should only be compiled for a specific target, say the Linux target, then you can create a *path file* called .Linux_path. Creating a *path file* is as simple as running.

```

> cd to/dir/to/add/to/path
> touch .all_path      # This code can be compiled for any target, or
> touch .Linux_path    # This code can only be compiled for the "Linux" target, or
> touch .Linux_Test_path # This code can only be compiled for the "Linux_Test" target

```

Note that the directory that `redo` is run within (usually your current directory) is added to the build path by default for convenience. Main programs (`main.adb` files) are generally NOT added to the build path using a *path file*, since they are not imported by any other package.

Also note that all model files and source files found within the Adamant build path MUST have unique names. Adamant disallows two files with the same name, or two models that could generate packages of the same name, within the build path. An error will be produced if this is detected. Adamant does this to prevent a situation where the inclusion of one package over another same-named package would be determined by link order, which can be the cause of many hard-to-track-down bugs. This feature, which can sometimes be annoying, is intended to prevent the compiling of ambiguous packages into the final binary. Due to this restriction, users must be meticulous about only including unique package names in their project

4 Basic Examples

This document goes into great detail about how you can create Adamant constructs like *components* and *assemblies*. However, you can also build standard packages and programs using the Adamant build system. This section walks you through a “hello world” example and provides direction for building and unit test a stand-alone Ada package.

4.1 Hello World

This section illustrates the steps needed to create a simple “hello world” program in Adamant using Ada. The program can also be written in C or C++ and the following steps still apply (rename the file and change syntax as appropriate).

First let’s create a directory for our hello world program.

```
> mkdir hello_world  
> cd hello_world
```

Next we create a file called *main.adb* inside our new directory with the following program written inside of it:

main.adb:

```
1  with Ada.Text_IO; use Ada.Text_IO;  
2  
3  procedure Main is  
4  begin  
5      Put_Line ("Hello, World!");  
6  end Main;
```

Adamant automatically detects the presence of this new *main.adb* file and provides new build rules for it. To see the new rules run:

```
> redo what  
redo what  
redo all  
redo clean  
redo clean_all  
redo templates  
redo publish  
redo targets  
redo prove  
redo analyze  
redo style  
redo pretty  
redo test_all  
redo analyze_all  
redo coverage_all  
redo build/bin/Linux/main.elf  
redo build/gpr/main.gpr  
redo build/metric/Linux/main.adb.txt  
redo build/metric/Linux/main.elf.txt  
redo build/metric/Linux/main.o.txt  
redo build/obj/Linux/main.o  
redo run
```

We can see many build rules are available in addition to the standard rules that are al-

ways available (see Section 3.3.1). In particular, there are rules to compile the *main.adb* into an object file, *redo build/obj/Linux/main.o*, link the object into a binary *.elf*, *redo build/bin/Linux/main.elf*, and a special rule, *redo run*, which will compile and run the hello world program. Let's try running the program:

```
> redo run
redo run
redo    build/bin/Linux/main.elf
redo    build/obj/Linux/main.o
Hello, World!
```

From the output we can see that the *redo run* command first tries to build the executable (*main.elf*). The executable depends on the object file (*main.o*), so that is compiled. Finally, after both the object and executable are constructed, the executable is run, and the expected string is output to the terminal.

If we run *redo run* again:

```
> redo run
redo run
Hello, World!
```

We see that this time the program is immediately run and the string is output. *redo* detects that the object and executable have already been built, and so just runs the program instead.

To gain more insight into what the Adamant build system produces when compiling we can look at all the files constructed within the *build/* directory:

```
> find build
build
build/obj
build/obj/Linux
build/obj/Linux/main.o
build/obj/Linux/b__main.ali
build/obj/Linux/main.odeps
build/obj/Linux/main.bexch
build/obj/Linux/b__main.adb
build/obj/Linux/b__main.o
build/obj/Linux/main.ali
build/obj/Linux/main.adb.stderr
build/obj/Linux/main.adb.stdout
build/obj/Linux/b__main.ads
build/bin
build/bin/Linux
build/bin/Linux/main.elf
```

We can see that two directories were created under *build/*: *obj/*, which contains *main.o* and other associated files that are generated during Ada compilation, and *bin/*, which contains our executable *main.elf*. All of these files are stored under a subdirectory named *Linux/* because that is default *build target*. See Section 3.3.2 for more details on *build targets*.

A few things should be noted about this example. First, we did not need to add the *hello_world* directory to the *build path*, see Section 3.3.3, since this is a main file, and will not be imported by any other package. Adamant automatically adds your current working directory to the *build path* so was able to compile the program without issue. Second, a *redo run* rule will be available any time a *main.adb* is found in a directory. This rule allows effortless compilation and running of any

executable program, so long as it is named `main.adb`. You will likely end up with many `main.adb` programs in your repository during development. Again, none of these directories should be added to the *build path* because 1) they do not need to be and 2) there would be many name conflicts that violate the Adamant constraint of only allowing unique file names within the *build path*.

The next section further expands on this example by creating an Ada package and unit testing it.

4.2 Creating a Package

Another common task, not tied to Adamant constructs, is building and compiling a simple Ada package. The following example shows you how to construct a simple package that contains a single subprogram. We then unit test this subprogram in the following section.

Let's start by making a directory for our package:

```
> mkdir simple_package
> cd simple_package
```

Next, we create two files, an Ada specification file called `simple_package.ads` and an Ada implementation file called `simple_package.adb`. Below is shown `simple_package.ads`:

```
1 package Simple_Package is
2
3     -- Create a single function that adds two signed integers together and
4     -- returns the result.
5     function Add_Two_Numbers (Left : in Integer; Right : in Integer) return
6         Integer;
7
8 end Simple_Package;
```

and `simple_package.adb`:

```
1 package body Simple_Package is
2
3     function Add_Two_Numbers (Left : in Integer; Right : in Integer) return
4         Integer is
5     begin
6         return Left + Right;
7     end Add_Two_Numbers;
8
9 end Simple_Package;
```

We can see that the package contains a single subprogram, specifically a function, that returns the sum of two `Integer` types. We can easily compile the package to make sure there are no syntax errors:

```
> redo
redo  all
redo  build/obj/Linux/simple_package.o
```

Nice! Now there is one more step. We must add this package to the *build path* so that it can be imported by other packages and programs within our system. To do this we need to create a *path file* in this directory.

```
> touch .all_path # Create a path file
```

We create a file called `.all_path`, since the code we wrote is not platform specific and should compile for any target hardware. See Section 3.3.3 for more details on *path files*. In practice, to create a new

package a user often copies an already existing package directory and then proceeds to modify it. In that case, the old directory likely already contains a *path file*, making creating one using the method above unnecessary.

OK, now we are ready to unit test the package.

4.3 Testing a Package

This section walks through different ways to unit test the package that we wrote in the previous section, and the different features available to you.

4.3.1 A Simple Unit Test

Now let's unit test the Ada package we wrote in the previous section. Unit test code for a package is usually stored in a subdirectory called *test/* within the package directory. If there is more than one set of unit test code needed then there is usually multiple unit test directories created, each with a directory name starting with "test_".

```
> mkdir test # Make test/ within simple_package/
> cd test
```

Next we create a single test main file called *test.adb*. Here are the contents:

```
1  with Simple_Package;
2
3  procedure Test is
4      A : constant Integer := 7;
5      B : constant Integer := -15;
6      C : Integer;
7  begin
8      -- Run the test:
9      C := Simple_Package.Add_Two_Numbers (A, B);
10
11     -- Run check using the basic Ada assertion statement, this should pass.
12     pragma Assert (C = -8, "Failed assertion 1, c = " & Integer'Image (C) & ".");
13
14     -- Expect this check to fail.
15     pragma Assert (C /= -8, "Failed assertion 2, c = " & Integer'Image (C) &
16                     ".");
16 end Test;
```

In this test we import the package we wrote with a *with* statement, and then call the function within. We run two checks using the Ada assertion *pragma*. We expect the second check to fail. To run the test we simply run:

```
> redo test
```

Note that a *redo test* command is available whenever Adamant detects a *test.adb* file in a directory. Running the test produces the output:

```
raised ADA ASSERTIONS ASSERTION_ERROR : Failed assertion 2, c = -8.
```

as expected. The output text is also copied to a log file that can be found in *build/log/test.elf.log*.

While effective, this form of simple unit testing is not the preferred method within Adamant. The next section shows a better way to create unit tests using the Adamant unit test framework.

4.3.2 Unit Testing with Adamant

The preferred method of unit testing is by using the Adamant unit test framework. This method is slightly more complicated than the previous section but there are many benefits. As will be shown, we gain access to a powerful unit testing framework, complete with test fixtures, see better error messages for unit test failures, and autogenerate documentation for our unit tests.

Note that Adamant uses the [AUnit](#), Ada Unit Testing Framework, under the hood. However, the details of AUnit are mostly abstracted away from the user by Adamant, so you need not be concerned with exactly how it works.

To get started let's make another testing subdirectory under *simple_package*:

```
> mkdir test_better # Make test_better/ within simple_package/  
> cd test_better
```

Next, we are going to create an Adamant unit test *model file* called *simple_package.tests.yaml*. Unit test model files should always be of the form *name.tests.yaml* where *name* describes the package you are testing. There is an extension of this naming convention for unit testing an Adamant component, see Section 6.2, but that is not be discussed here. Here are the contents of *simple_package.tests.yaml*:

```
1  ---  
2  # Optional - description of unit test suite  
3  description: This is a set of unit tests for the Simple_Package package.  
4  # Required - list of tests  
5  tests:  
6      # Required - name of test  
7      - name: Test_That_Should_Pass  
8          # Optional - description of test  
9          description: This test should pass.  
10     - name: Test_That_Should_Fail  
11         description: This test should not pass.
```

As can be seen, the format of the YAML file is pretty simple. We list the unit tests that we plan on writing and a description for each. That is it!

OK, there is one more detail to take care of. Because we are using the AUnit library, we need to make sure our *build target* is set to `Linux_Test`, as the `Linux` target does not link in AUnit by default. We can run the command `export TARGET=Linux_Test`, but then we would have to remember to do this every time we want to run the test. A better approach is to create an *env.py* that does this for us. Adamant uses the same *env.py* for almost all unit tests, so you can easily copy it from another unit test directory. Here are the standard contents of *env.py*, which is created in the *test_better* directory.

```
1  from environments import test # noqa: F401
```

If all of that *env.py* discussion is news to you, check out Section 9.10.3 for more details.

OK, now let's see what Adamant can do with the YAML model file:

```
> redo what  
redo what  
redo all  
redo clean  
redo clean_all  
redo templates  
redo publish
```

```

redo targets
redo prove
redo analyze
redo style
redo pretty
redo test_all
redo analyze_all
redo coverage_all
redo build/bin/Linux_Test/test.elf
redo build/gpr/test.gpr
redo build/html/simple_package_tests.html
redo build/metric/Linux_Test/simple_package_tests-implementation-suite.adb.txt
redo build/metric/Linux_Test/simple_package_tests-implementation-suite.ads.txt
redo build/metric/Linux_Test/simple_package_tests-implementation.adb.txt
redo build/metric/Linux_Test/simple_package_tests-implementation.ads.txt
redo build/metric/Linux_Test/simple_package_tests.adb.txt
redo build/metric/Linux_Test/simple_package_tests.ads.txt
redo build/metric/Linux_Test/test.adb.txt
redo build/metric/Linux_Test/test.elf.txt
redo build/metric/Linux_Test/test.o.txt
redo build/obj/Linux_Test/simple_package_tests-implementation-suite.o
redo build/obj/Linux_Test/simple_package_tests-implementation.o
redo build/obj/Linux_Test/simple_package_tests.o
redo build/obj/Linux_Test/test.o
redo build/src/simple_package_tests-implementation-suite.adb
redo build/src/simple_package_tests-implementation-suite.ads
redo build/src/simple_package_tests.adb
redo build/src/simple_package_tests.ads
redo build/template/simple_package_tests-implementation.adb
redo build/template/simple_package_tests-implementation.ads
redo build/template/test.adb
redo coverage
redo output.txt
redo test

```

Whoa! There are a lot of new files we can build here. Most of these are boilerplate autocoded files that allow your unit tests to function nicely with the underlying AUnit library. You will probably never need to look at this autogenerated source code unless you are the curious type. What is most important here are the autogenerated files in *build/template*. These files are autogenerated files meant to be a starting point for us to start working on our unit tests. The standard procedure is to copy the autogenerated files out of the *build/template* directory into our current directory, and then we can start modifying them however we like. Let's build the template files and copy them out now:

```

> redo build/template/simple_package_tests-implementation.adb
> redo build/template/simple_package_tests-implementation.ads
> redo build/template/test.adb
> cp build/template/* . # copy the template files into test_better/

```

You can also use the special `redo templates` command to do the same thing:

```

> redo templates
> cp build/template/* . # copy the template files into test_better/

```

Note, it is common while developing to write the source code for few unit tests only to realize later that you need some more unit tests defined to fully test your package. In this case, the standard procedure is to update the model file, ie. *simple_package.tests.yaml*, with your new test definitions,

rebuild the templates as seen above, and then manually merge the templates with the unit test code that you have been writing. `diff`, `git diff`, or a more sophisticated merge tool may aid you in this process. While this process can be cumbersome, achieving this automatically would be difficult and error prone, at best.

Know that if you make a mistake while merging, Adamant will catch most issues, like forgetting to implement a test, through modeling errors and compilation errors. Specifically, Adamant ensures that you implement all the tests that you define in your model through the use of abstract functions defined at the unit test base package level. Feel free to take a look at these abstract definitions in the autocoded files `build/src/simple_package_tests.ads` and `build/src/simple_package_tests.adb`. These files are not shown here for brevity.

Let's take a look at the templates we created in the steps above, and then we can start modifying them.

simple_package_tests-implementation.ads:

```
1 -----  
2 -- Simple_Package Tests Spec  
3 -----  
4  
5 -- This is a set of unit tests for the Simple_Package package.  
6 package Simple_Package_Tests.Implementation is  
7  
8     -- Test data and state:  
9     type Instance is new Simple_Package_Tests.Base_Instance with private;  
10    type Class_Access is access all Instance'Class;  
11  
12  private  
13      -- Fixture procedures:  
14      overriding procedure Set_Up_Test (Self : in out Instance);  
15      overriding procedure Tear_Down_Test (Self : in out Instance);  
16  
17      -- This test should pass.  
18      overriding procedure Test_That_Should_Pass (Self : in out Instance);  
19      -- This test should not pass.  
20      overriding procedure Test_That_Should_Fail (Self : in out Instance);  
21  
22      -- Test data and state:  
23      type Instance is new Simple_Package_Tests.Base_Instance with record  
24          null;  
25      end record;  
26  end Simple_Package_Tests.Implementation;
```

As we can see there are two procedures at the top of the package called `Set_Up_Test` and `Tear_Down_Test`. These are test fixtures that are run before and after every unit test, respectively. These procedures can be used to set up a consistent test environment for each test, and then tear down that environment after each test. We do not need to use these for our simple package.

simple_package_tests-implementation.adb:

```
1 -----  
2 -- Simple_Package Tests Body  
3 -----  
4  
5 with AUnit.Assertions; use AUnit.Assertions;  
6  
7 package body Simple_Package_Tests.Implementation is  
8  
9     -----  
10    -- Fixtures:
```

```

11 -----
12
13 overriding procedure Set_Up_Test (Self : in out Instance) is
14 begin
15     -- TODO Insert custom set up code here.
16     null;
17 end Set_Up_Test;
18
19 overriding procedure Tear_Down_Test (Self : in out Instance) is
20 begin
21     -- TODO Insert custom cleanup code here.
22     null;
23 end Tear_Down_Test;
24
25 -----
26 -- Tests:
27 -----
28
29 -- This test should pass.
30 overriding procedure Test_That_Should_Pass (Self : in out Instance) is
31     -- TODO declarations
32 begin
33     -- TODO replace the following with actual test code.
34     Assert (False, "Test 'Test_That_Should_Pass' is unimplemented.");
35 end Test_That_Should_Pass;
36
37 -- This test should not pass.
38 overriding procedure Test_That_Should_Fail (Self : in out Instance) is
39     -- TODO declarations
40 begin
41     -- TODO replace the following with actual test code.
42     Assert (False, "Test 'Test_That_Should_Fail' is unimplemented.");
43 end Test_That_Should_Fail;
44
45 end Simple_Package_Tests.Implementation;

```

This is the file where we add our actual unit test logic. You can see TODO comments in the source code where we need to write our code. Right now, the unit tests should compile but will fail when run.

test.adb:

```

1 -----  

2 -- Simple_Package_Tests  

3 -----  

4  

5 with AUnit.Reporter.Text;  

6 with AUnit.Run;  

7 with Simple_Package_Tests.Implementation.Suite;  

8 -- Make sure any terminating tasks are handled and an appropriate  

9 -- error message is printed.  

10 with Unit_Test_Termination_Handler;  

11 pragma Unreferenced (Unit_Test_Termination_Handler);  

12  

13 procedure Test is  

14     -- Create runner for test suite:  

15     procedure Runner is new AUnit.Run.Test_Runner  

16         (Simple_Package_Tests.Implementation.Suite.Get);  

17     -- Use the text reporter:  

18     Reporter : AUnit.Reporter.Text.Text_Reportert;  

19 begin  

    -- Add color output to test run:  


```

```

20     AUnit.Reporter.Text.Set_Use_ANSI_Colors (Reporter, True);
21     -- Run tests:
22     Runner (Reporter);
23 end Test;

```

This file contains boilerplate AUnit test running code. It never needs to be modified, or even looked at again. It does however, need to be copied from *build/template* for things to function properly.

At this point we can actually run our unit tests, and we should see them fail.

```
> redo test
```

which should produce the output:

```

FAIL Test_That_Should_Pass
    Test 'Test_That_Should_Pass' is unimplemented.
    at simple_package_tests-implementation.adb:33
FAIL Test_That_Should_Fail
    Test 'Test_That_Should_Fail' is unimplemented.
    at simple_package_tests-implementation.adb:40

Total Tests Run:    2
Successful Tests:  0
Failed Assertions: 2
Unexpected Errors: 0

```

which tells us both of our tests failed because they are unimplemented. This nice output message is a result of using the Adamant unit test framework.

Note that the testing output text is also copied to a log file that can be found in *build/log/test.elf.log*.

Now it is time to implement our tests. Below is shown the implementation file again, but this time, it contains actual test code.

simple_package_tests-implementation.adb:

```

1 -----
2 -- Simple_Package Tests Body
3 -----
4
5 with Basic_Assertions; use Basic_Assertions;
6 with Simple_Package;
7
8 package body Simple_Package_Tests.Implementation is
9
10    -----
11    -- Fixtures:
12    -----
13
14    overriding procedure Set_Up_Test (Self : in out Instance) is
15    begin
16        null;
17    end Set_Up_Test;
18
19    overriding procedure Tear_Down_Test (Self : in out Instance) is
20    begin
21        null;

```

```

22  end Tear_Down_Test;
23
24  -----
25  -- Tests:
26  -----
27
28  overriding procedure Test_That_Should_Pass (Self : in out Instance) is
29      Ignore : Instance renames Self;
30      A : constant Integer := 7;
31      B : constant Integer := -15;
32  begin
33      -- Result should equal -8
34      Integer Assert.Eq (Simple_Package.Add_Two_Numbers (A, B), -8);
35      -- Result should not equal -9
36      Integer Assert.Neq (Simple_Package.Add_Two_Numbers (A, B), -9);
37      -- Result should be less than 100
38      Integer Assert.Lt (Simple_Package.Add_Two_Numbers (A, B), 100);
39      -- Result should be greater than or equal to -100
40      Integer Assert.Ge (Simple_Package.Add_Two_Numbers (A, B), -100);
41  end Test_That_Should_Pass;
42
43  overriding procedure Test_That_Should_Fail (Self : in out Instance) is
44      Ignore : Instance renames Self;
45      A : constant Integer := 7;
46      B : constant Integer := -15;
47  begin
48      -- Result should equal 99? This test will fail, oh no!
49      Integer Assert.Eq (Simple_Package.Add_Two_Numbers (A, B), 99);
50  end Test_That_Should_Fail;
51
52 end Simple_Package_Tests.Implementation;

```

For these tests we use the Adamant provided `Basic_Assertions` package which implements the Adamant `Smart_ASSERT` package for many common Ada types, including `Integer`. The assertions provided by the `Basic_Assertions` package are easier to use than `pragma Assert` and provide much more informative error messages with little effort. The comments in the implementation show different ways to use this package.

Running the unit test again:

```
> redo test
```

produces:

```

OK Test_That_Should_Pass

FAIL Test_That_Should_Fail
    Assertion:
    -8
    =
    99
failed.
    at simple_package_tests-implementation.adb:49

Total Tests Run: 2
Successful Tests: 1
Failed Assertions: 1
Unexpected Errors: 0

```

As expected the first test passes with no errors and the second test fails with an informative error message. If we fix the second test to make all the tests pass and then run `redo test` again, we get the lovely output.

```
OK Test_That_Should_Pass
OK Test_That_Should_Fail

Total Tests Run:    2
Successful Tests:  2
Failed Assertions: 0
Unexpected Errors: 0
```

This section shows how to unit test a very simple Ada package. Most packages within Adamant use object-oriented concepts. The following section introduces you to the standard way objects (*tagged types*) are coded using Ada within Adamant, and the differences when unit testing them.

4.3.3 Unit Test Logger

When creating a new unit test, Adamant provides the ability for the user to print text to a test log file. Logging can give the user more insight into how their program is running. A new log is open for each individual unit test that has been defined by the user. A user can then simply add their own log statements using the provided `Self.Log` procedure. An example of this is shown for the same unit test of `simple_package` as seen below:

`simple_package_tests-implementation.adb`:

```
1  -----
2  -- Simple_Package Tests Body
3  -----
4
5  with Basic_Assertions; use Basic_Assertions;
6  with Simple_Package;
7
8  package body Simple_Package_Tests.Implementation is
9
10   -----
11  -- Fixtures:
12  -----
13
14  overriding procedure Set_Up_Test (Self : in out Instance) is
15  begin
16      -- Call base Set_Up:
17      -- Simple_Package_Tests.Base_Instance (Self).Set_Up;
18      null;
19  end Set_Up_Test;
20
21  overriding procedure Tear_Down_Test (Self : in out Instance) is
22  begin
23      -- Call base Tear_Down:
24      -- Simple_Package_Tests.Base_Instance (Self).Tear_Down;
25      null;
26  end Tear_Down_Test;
27
28  -----
29  -- Tests:
30  -----
31
32  overriding procedure Test_That_Should_Pass (Self : in out Instance) is
```

```

33      Ignore : Instance renames Self;
34      A : constant Integer := 7;
35      B : constant Integer := -15;
36  begin
37      -- Result should equal -9
38      Self.Log ("Performing equals assertion: Added = " &
39                  → Simple_Package.Add_Two_Numbers (A, B)'Image & " Expected = -8");
40      Integer Assert.Eq (Simple_Package.Add_Two_Numbers (A, B), -8);
41      -- Result not equal -9
42      Self.Log ("Performing not equal assertion: Added = " &
43                  → Simple_Package.Add_Two_Numbers (A, B)'Image & " Expected /= -9");
44      Integer Assert.Neq (Simple_Package.Add_Two_Numbers (A, B), -9);
45      -- Result should less than 100
46      Self.Log ("Performing less than assertion: Added = " &
47                  → Simple_Package.Add_Two_Numbers (A, B)'Image & " Expected < 100");
48      Integer Assert.Lt (Simple_Package.Add_Two_Numbers (A, B), 100);
49      -- Result should be greater than or equal to -100
50      Self.Log ("Performing greater or equal than assertion: Added = " &
51                  → Simple_Package.Add_Two_Numbers (A, B)'Image & " Expected >= -100");
52      Integer Assert.Ge (Simple_Package.Add_Two_Numbers (A, B), -100);
53  end Test_That_Should_Pass;

54
55  overriding procedure Test_That_Should_Fail (Self : in out Instance) is
56      Ignore : Instance renames Self;
57      A : constant Integer := 7;
58      B : constant Integer := -15;
59  begin
60      -- Result should equal 99? This test will fail, oh no!
61      Integer Assert.Neq (Simple_Package.Add_Two_Numbers (A, B), 99);
62  end Test_That_Should_Fail;
63
64 end Simple_Package_Tests.Implementation;

```

Unit test logs will always be written to the subdirectory *build/log*. An example of the output for each test is shown below:

```

build/log/Test_That_Should_Pass.log
1752616167.761982000 Beginning log for Test_That_Should_Pass at 2025-07-15 21:49:27
1752616167.761990000 Starting Set_Up for test Test_That_Should_Pass
1752616167.761991000 Finishing Set_Up for test Test_That_Should_Pass
1752616167.761992000 Performing equals assertion: Added = -8 Expected = -8
1752616167.761997000 Performing not equal assertion: Added = -8 Expected /= -9
1752616167.761997000 Performing less than assertion: Added = -8 Expected < 100
1752616167.761998000 Performing greater or equal than assertion: Added = -8 Expected >=
1752616167.762000000 Starting Tear_Down for test Test_That_Should_Pass
1752616167.762000000 Finishing Tear_Down for test Test_That_Should_Pass
1752616167.762001000 Ending log for Test_That_Should_Pass at 2025-07-15 21:49:27 and to

```

The first test here is the passing test where all the messages are logged. Logs always appear in the form *<timestamp> <message>* where *<timestamp>* is Unix seconds. The user can log anything available to them in the unit test, such as the individual items being tested.

The next test is the failure test, but this time there are no user defined messages injected into the log. Here we see the default messages that get logged including the timing for starting and finishing the test, as well as the setup and tear down fixtures.

```

build/log/Test_That_Should_Fail.log
1752616167.762041000 Beginning log for Test_That_Should_Fail at 2025-07-15 21:49:27
1752616167.762044000 Starting Set_Up for test Test_That_Should_Fail

```

```

1752616167.762044000 Finishing Set_Up for test Test_That_Should_Fail
1752616167.762045000 Starting Tear_Down for test Test_That_Should_Fail
1752616167.762046000 Finishing Tear_Down for test Test_That_Should_Fail
1752616167.762046000 Ending log for Test_That_Should_Fail at 2025-07-15 21:49:27 and to

```

This is a simple but easy way for the user to mark where issues are occurring, or even print out their own data to the log for faster debugging of both the unit test code and the production code.

4.3.4 Unit Test Coverage Report

It is often desirable to know how much of the implementation code that a unit test actually tests. One metric for quantifying this is through a “coverage” report, which shows which lines of code the unit tests traversed, and which lines of code were not executed. For Adamant embedded systems, a 100% line coverage metric is required, and justification must be provided if this goal cannot be achieved. Note that Adamant uses the [gcovr](#) tool to generate reports.

Once we have a passing set of unit tests, creating a coverage report is not difficult. From the `test_better` directory created in the last section we can run:

```
> redo coverage # create unit test coverage report
```

which produces the output:

```
(INFO) Reading coverage data...
(INFO) Writing coverage report...
```

GCC Code Coverage Report				
Directory: doc/example_architecture/simple_package				
File	Lines	Exec	Cover	Missing
simple_package.adb	2	2	100%	
test_better3/build/obj/Linux_Coverage/b__test.adb	221	219	99%	242,298
test_better3/build/src/simple_package_tests-implementation-suite.adb	7	7	100%	
test_better3/build/src/simple_package_tests.adb	31	31	100%	
test_better3/build/src/simple_package_tests.ads	6	6	100%	
test_better3/simple_package_tests-implementation.adb	23	23	100%	
test_better3/simple_package_tests-implementation.ads	3	3	100%	
test_better3/test.adb	4	4	100%	
TOTAL	297	295	99%	

which shows that we have 100% coverage on `simple_package.adb`, which is the package we are interested in fully testing. Note, this print out can also be found in the autogenerated text file `build/coverage/coverage.txt`.

In addition to a text file, the `redo coverage` command also creates a user friendly HTML

output that shows each files and color-coded lines of code executed. This can be viewed by opening `build/coverage/test.html` in your favorite web browser. Below are a few examples of what the HTML output looks like:

Summary Page:

GCC Code Coverage Report		Exec	Total	Coverage
Directory: /	Date: 2020-04-20 21:31:32	Lines: 231	233	99.1 %
Legend: low: < 75.0 % medium: >= 75.0 % high: >= 90.0 %		Branches: 16	32	50.0 %
File		Lines		Branches
<code>simple_package.adb</code>		100.0 %	2 / 2	50.0 % 1 / 2
<code>test_better3/build/obj/Linux_Coverage/b_test.adb</code>		98.9 %	181 / 183	50.0 % 7 / 14
<code>test_better3/build/src/simple_package_tests-implementation-suite.adb</code>		100.0 %	7 / 7	- % 0 / 0
<code>test_better3/build/src/simple_package_tests.adb</code>		100.0 %	8 / 8	50.0 % 2 / 4
<code>test_better3/build/src/simple_package_tests.ads</code>		100.0 %	3 / 3	- % 0 / 0
<code>test_better3/simple_package_tests-implementation.adb</code>		100.0 %	23 / 23	50.0 % 4 / 8
<code>test_better3/simple_package_tests-implementation.ads</code>		100.0 %	3 / 3	- % 0 / 0
<code>test_better3/test.adb</code>		100.0 %	4 / 4	50.0 % 2 / 4

Generated by: [GCOVR \(Version 4.2\)](#)

Specific File Page:

GCC Code Coverage Report		Exec	Total	Coverage
Directory: /	File: <code>simple_package.adb</code>	Lines: 2	2	100.0 %
Date: 2020-04-20 21:31:32		Branches: 1	2	50.0 %
Line Branch Exec Source				
1				package body Simple_Package is
2				
3				
4		5		function Add_Two_Numbers(left : in Integer; right : in Integer) return Integer is
5				
6	X✓	5		begin
7				return left + right;
8				end Add_Two_Numbers;
9				end Simple_Package;

Generated by: [GCOVR \(Version 4.2\)](#)

4.3.5 Unit Test Documentation

Adamant provides automatic generation of documentation for any unit test model. Currently two versions of documentation can be created: HTML, which is useful for presentation in meetings or for quick reference, and PDF (via L^AT_EX), which is useful for formal documentation.

To build HTML documentation run:

```
> redo build/html/simple_package_tests.html
```

which when opened with your favorite web browser looks something like this:

Simple_Package_Tests Unit Test Suite

Description: This is a set of unit tests for the Simple_Package package.

Test Name	Description
Test_That_Should_Pass	This test should pass.
Test_That_Should_Fail	This test should not pass.

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/simple_package/test_better3/simple_package.tests.yaml on 2020-04-21 14:14.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

To build the PDF documentation run:

```
> redo build/pdf/simple_package_tests.pdf
```

Which produces a PDF output that looks like the following.

This is a set of unit tests for the Simple_Package package.

- **Test_That_Should_Pass** - This test should pass.
 - **Test_That_Should_Fail** - This test should not pass.
-

4.4 Object-oriented Design

The following section introduces you to object-oriented design in Ada and the standard practices and conventions used in Adamant. In addition, we walk through how to unit test an object-oriented package.

4.4.1 Creating an Object-oriented Package

The previous sections discussed writing, compiling, and unit testing a simple Ada package in Adamant. The source code for Adamant heavily utilizes object-oriented design principles, where programming operations are mostly concerned with the interaction and behavior of "objects". This section shows how to create a very simple "class"-like entity in Ada, known as a *tagged type*, which supports common object-oriented patterns such as overriding, polymorphism, dynamic dispatching, inheritance, mix-ins, etc.

Note that this section does not delve into the specifics of object-oriented programming in Ada. Instead it attempts to show simply what a simple tagged type might look like, and how unit testing it is slightly different from the example presented in the previous sections. For details on the object-oriented programming in Ada see the [Ada Programming Wiki](#) or [AdaCore Learn](#).

First let's create a directory for the package as before, and add this directory to the *build path*.

```
> mkdir oo_package  
> cd oo_package  
> touch .all_path # add directory to the build path
```

Below is the source code for an object-oriented tagged type that we can put in this directory: *oo_package.ads*:

```

1 package Oo_Package is
2
3     -- Declare a tagged type. In Adamant, this is usually named
4     -- "Instance".
5     type Instance is tagged private;
6
7     -- Declare an initialization procedure (a primitive operation)
8     -- that sets the state of a variable called "N" inside of the tagged type.
9     procedure Init (Self : in out Instance; N : in Integer);
10
11    -- Create a primitive for our tagged type that adds N to a number.
12    -- N is a state of instance that is stored upon initialization.
13    function Add_N (Self : in Instance; Left : in Integer) return Integer;
14
15    -- Everything below this is private, and is only accessible
16    -- to child packages of this package.
17    private
18
19        -- This is the private implementation of type instance. Instance
20        -- is a record that contains a single private member variable, n.
21    type Instance is tagged record
22        N : Integer := 0;
23    end record;
24
25 end Oo_Package;

```

oo_package.adb:

```

1 package body Oo_Package is
2
3     procedure Init (Self : in out Instance; N : in Integer) is
4     begin
5         -- Set our internal state variable to the input:
6         Self.N := N;
7     end Init;
8
9     function Add_N (Self : in Instance; Left : in Integer) return Integer is
10    begin
11        return Left + Self.N;
12    end Add_N;
13
14 end Oo_Package;

```

We can see that the package contains a tagged type named `Instance`. `Instance` is the standard name used for a tagged type within Adamant when a package contains a single tagged type, which is common.

Below the tagged type is two public subprograms, or “primitives”. These subprograms are considered primitive operations of the tagged type `Instance` because `Instance` is passed as an argument to them. Primitives are similar to “methods” in Java or a “member functions” in C++. The first subprogram is a procedure called `Init` which initializes the object, which is always named `self` in Adamant. The second subprogram implements an adding function similar to what was presented in the previous sections. This time, however, a private variable, `n`, stored within the tagged type record is added to the input argument instead. The variable `n` is part of the private definition of type `Instance` and thus cannot be accessed outside of the `Oo_Package`, except within a child package, as we will see in the next section.

4.4.2 “White-box” Unit Testing

In this section we unit test the object-oriented package we created in the previous section. Tagged types are unique in that they act as objects that encapsulate private data members. These data members are usually not accessible to the outside world, meaning the only way to test the object-oriented package is by calling its public methods, also known as “black-box” testing. However, using the patterns seen in this section, we also perform “white-box” testing, allowing us to safely check the values of the private members within the object, without compromising the type safety of the deployment code by making the type definition public.

First, let’s create an Adamant unit test YAML file as before in a directory called `test/`.

```
> mkdir test # Make test/ within oo_package/
> cd test
```

In this directory we can create a test model file called `oo_package.tests.yaml` with the following contents:

```
1  ---
2  # Optional - description of unit test suite
3  description: This is a set of unit tests for the Oo_Package package.
4  # Required - list of tests
5  tests:
6      # Required - name of test
7      - name: Test_Init
8          # Optional - description of test
9          description: This test makes sure the initialization function works as
10             ↳ expected.
11      - name: Test_Add
12          description: This test makes sure the addition function works as expected.
```

We declare two tests, one to test each of the primitive operations of our tagged type.

Note that within our unit tests we are not able to access the private member variable `n` within `Oo_Package.Instance` unless we create a special “child” package that can provide us visibility. In Ada, child packages have access the private variables and subprograms found in their parent package. Note that a child package is not the same as type inheritance, which is a distinct concept in Ada. Child packages ONLY provide a mechanism for package extension and control of private sections.

Let’s create a child package called `Oo_Package.Tester` that has a function that gives us visibility into the private variable `n`. Note the file and package naming conventions seen below. This pattern is required by Ada for proper compilation of a child package.

`oo_package-tester.ads:`

```
1  package Oo_Package.Tester is
2
3      -- This child package has access to the parent package's private members.
4      -- This is useful for unit testing since it is nice to inspect an object's
5      -- internal private variables during testing, even though you don't want this
6      -- feature during the deployment of your software.
7
8      -- This function retrieves the private value of "n" from within the object's
9      -- record and returns it.
10     function Get_N (Self : in Instance) return Integer;
11
12 end Oo_Package.Tester;
```

`oo_package-tester.adb:`

```

1 package body Oo_Package.Tester is
2
3     function Get_N (Self : in Instance) return Integer is
4     begin
5         return Self.N;
6     end Get_N;
7
8 end Oo_Package.Tester;

```

Excellent. We can now use this fancy new package and function in our unit tests. To write our unit tests we follow the same procedure as presented previously. First, we make the unit test templates, copy them into our test directory, and then modify them to taste.

```

> redo build/template/oo_package_tests-implementation.adb
> redo build/template/oo_package_tests-implementation.ads
> redo build/template/test.adb
> cp build/template/* . # copy the template files into test_better/

```

The unit test files with complete unit test code are shown below.

oo_package_tests-implementation.ads:

```

1 -----
2 -- Oo_Package Tests Spec
3 -----
4
5 with Oo_Package;
6
7 -- This is a set of unit tests for the Oo_Package package.
8 package Oo_Package_Tests.Implementation is
9     -- Test data and state:
10    type Instance is new Oo_Package_Tests.Base_Instance with private;
11    private
12        -- Fixture procedures:
13        overriding procedure Set_Up_Test (Self : in out Instance);
14        overriding procedure Tear_Down_Test (Self : in out Instance);
15
16        -- This test makes sure the initialization function works as expected.
17        overriding procedure Test_Init (Self : in out Instance);
18        -- This test makes sure the addition function works as expected.
19        overriding procedure Test_Add (Self : in out Instance);
20
21        -- Test data and state:
22        type Instance is new Oo_Package_Tests.Base_Instance with record
23            Oo : Oo_Package.Instance;
24        end record;
25    end Oo_Package_Tests.Implementation;

```

oo_package_tests-implementation.adb:

```

1 -----
2 -- Oo_Package Tests Body
3 -----
4
5 with Basic_Assertions; use Basic_Assertions;
6 with Oo_Package.Tester;
7
8 package body Oo_Package_Tests.Implementation is
9
10    -----
11    -- Fixtures:

```

```

12 -----
13
14 overriding procedure Set_Up_Test (Self : in out Instance) is
15 begin
16     -- Initialize the object oriented package by calling the Init function:
17     Self.Oo.Init (N => -15);
18 end Set_Up_Test;
19
20 overriding procedure Tear_Down_Test (Self : in out Instance) is
21 begin
22     null;
23 end Tear_Down_Test;
24
25 -----
26 -- Tests:
27 -----
28
29 overriding procedure Test_Init (Self : in out Instance) is
30 begin
31     -- This is a white-box test. We need to look inside of the
32     -- Instance's private state and make sure that the value of
33     -- "n" got set to what we expect, 15.
34     Integer Assert.Eq (Oo_Package.Tester.Get_N (Self.Oo), -15);
35 end Test_Init;
36
37 overriding procedure Test_Add (Self : in out Instance) is
38 begin
39     -- The result should equal -8;
40     Integer Assert.Eq (Self.Oo.Add_N (7), -8);
41
42     -- Make sure the the operation above did not alter the
43     -- object's internal state. Only the Init function should
44     -- be able to do that.
45     Integer Assert.Eq (Oo_Package.Tester.Get_N (Self.Oo), -15);
46 end Test_Add;
47
48 end Oo_Package_Tests.Implementation;

```

There are a few important differences here compared to the previous sections. First, an object of type `Oo_Package.Instance` is declared in the record of the unit test packed record itself named `Oo`. This makes the object part of the unit test class, and thus callable from the fixtures and unit test procedures. Second, we use the `Set_Up_Test` fixture this time to call the `Init` method on the `Oo_Package.Instance` object. This fixture will get called before both unit tests are run.

In the first unit test, all we do is call the special function we implemented in the child package `oo_package-tester.ads`, which gives us visibility into the private variable `N`. This test makes sure that `N` was initialized properly via the `Init` function. In the second unit test we call the addition function and make sure it returns the expected result. We also call the child subprogram `Get_N` again to make sure that the internal state of the `Oo_Package.Instance` object did not change as a result of the call to `Self.Oo.Add_N`, which should not modify the object.

Everything works as expected and produces the test output:

```

OK Test_Init
OK Test_Add

Total Tests Run:    2
Successful Tests:  2

```

Failed Assertions: 0
Unexpected Errors: 0

5 Packed Types

Often in embedded systems, communication protocols and hardware devices require a very specific binary format for transferred bytes or bits to properly function. However, technical differences in processors and compilers can often make code built to talk to these devices difficult to implement in a cross-platform and portable way. For instance, processors used in the design of embedded systems can be either big or little endian, depending on the chosen platform. Embedded systems often communicate with other embedded processors or end-user systems which have an endianness different from their own. If not carefully handled, these differences in endianness can be one of the largest sources of defects encountered while debugging embedded software. Besides endianness issues, the “packed”-ness, that is, the amount of padding supplied around individual types, of a struct or array in processor memory differs based on the compiler used. The differences in endianness and “packed”-ness can make writing portable software difficult without a layer of abstraction to handle the differences.

Adamant aims to solve these problem through the use of *packed types*. In Adamant, packed types are modeled using a YAML specification. From the specification, the following type definitions are generated:

- **T** - A big endian packed representation of the type with no padding. These are the most commonly seen type in Adamant.
- **T_Le** - A little endian packed representation of the type with no padding. This is usually only used to communicate with little endian specific external hardware.
- **U** - A *native* version of the type whose bit layout has been chosen by the compiler. This is usually the most performant version of the type, but the bit layout changes based on the compiler used, and thus is not portable. This type will be in endianness of the target processor and will often include padding to align values to machine-defined boundaries for performance. This version is often used when performance is a concern, but the layout of the bits does not matter.

The T and T_Le types can be thought of as the *serialized* version of the record, where the bit layout and endianness is known, no matter the compiler or processor that the code is running on. The U version of the type can be thought of as the *native* version of the type, which has been optimized by the compiler for the target processor. Adamant provides subprograms to convert between these 3 types so that a user can easily pack, unpack, or swap the endianness of the data at will. Subprogsams to reinterpret packed types as an array of bytes (serialization) or to reinterpret an array of bytes as a packed type (deserialization) are also provided.

For each of the types above, Adamant also provides facilities to validate that the values in the type are within range, pretty print the type, check the value of the type during testing via assertions, and produce HTML and PDF documentation for the type. These features are discussed in the following subsections.

5.1 Packed Records

Using packed records is the preferred method of declaring data types in Adamant. A packed record is a structure-like datatype which can contain an arbitrary number of fields of any type, including arrays, packed arrays, or even other packed records.

To define a packed record, a developer should create a YAML model file of the form *name.record.yaml*, where *name* is the desired name of the packed record. Let’s create a record with the following fields:

- **Value_1** - A five bit modular type that wraps around at 32.
- **Value_2** - A 3 bit signed integer type with a valid range from -3 to 2.
- **Value_3** - An 8-bit enumeration type with valid states: Red, Green, Blue, Purple, Pink, and

Chartreuse

- **Value_4** - A 32-bit floating point type.

Below is the example packed record definition called *example_record*, stored in a file called *example_record.record.yaml*:

```

1  ---
2  # Optional - description of the packed record
3  description: This is an example packed record.
4  # Optional - preamble which provides "in-line" Ada code
5  preamble: |
6      type Five_Bit_Integer is mod 2**5;
7      subtype Three_Bit_Signed_Integer is Integer range -3 .. 2;
8      type My_Color is (Red, Green, Blue, Purple, Pink, Chartreuse);
9      # Optional - Include any dependencies we might need
10     # * In this case we don't have any... but here is what it might
11     # look like:
12     # with:
13     #   - Basic_Types
14     # List the record fields
15     fields:
16         # Required - the name of the field
17         - name: Value_1
18             # Optional - description of the field
19             description: This is the first value.
20             # Required - the type of the field
21             type: Five_Bit_Integer
22             # Required (most of the time) - provide the number of bits
23             # that this type will consume when packed. This field is
24             # required unless the type is itself another packed record
25             format: U5
26             # Optional - provide a default value
27             default: "0"
28         - name: Value_2
29             description: This is the second value.
30             type: Three_Bit_Signed_Integer
31             format: I3
32         - name: Value_3
33             type: My_Color
34             format: E8
35         - name: Value_4
36             description: This is the fourth value.
37             type: Short_Float
38             format: F32

```

This YAML model file above contains comments explaining whether or not each field is optional or required. The main portion of the model is the packed record *fields* which require a *name*, *type*, and *format* specification. The type of the field can be any Ada type, including arrays, records, enumerations, packed arrays, or other packed records. The *format* field is where a developer can describe out how the field is represented in memory in its packed form. This field takes an enumeration of the form *U*, *I*, *F* or *E* (for unsigned, signed, floating point, or enumeration) followed by the number of bits used to represent it. A table of example memory representations is laid out below:

Table 2: Example Packed Record Format Specifications

Format	Description
U8	Unsigned 8-bit integer
I16	Signed 16-bit integer
U1	Unsigned single bit

I13	Signed 13-bit integer
F32	32-bit floating point value
F64	64-bit floating point value
E3	3-bit enumeration type
U8x32	32-bytes of unsigned integers (often used for arrays)
E5x10	an array of ten 5-bit enumerations

Ada uses the *format* specification to understand the number of bits taken up by the type. If the number of bits specified in the representation field is not compatible with the type specified in the type field, the Ada compiler will warn or error, telling the user to fix the inconsistency. The enumeration portion of the *format* is needed to make the record visible to a ground system, such as Hydra. Adamant currently supports the display of many representations in Hydra including all signed and unsigned integers from 1 to 64 bits and floating point numbers of 32 and 64 precision and enumeration types. There is no support in Adamant or Hydra to display values with representations over 64 bits in length. The record should be further divided or an array *format* should be used in this case. Note that an entire record's length must end on a 1 byte (8-bit) boundary, or a error message will be produced. Add additional fields of pad bytes to make the total length of the record end on an 8-bit boundary.

Note that packed records can also specify fields with types that are themselves other packed records (or packed arrays, discussed in Section 5.2). In this case, a *format* specification for that field is not necessary. The concept of nesting packed records within other packed records is discussed further in Section 5.1.10. Including records or arrays defined in raw Ada packages (ie. not modeled using Adamant) are not allowed to be included in packed records, since they are, by default, unpacked in representation and their lengths and bit layout is not known to the Adamant modeling system.

5.1.1 Packed Record Ada Specification

The *example_record.record.yaml* file produces a single Ada specification file which declares a package containing the packed and unpacked representations of the record. The file can be generated and compiled by running the following command from the same directory as *example_record.record.yaml*:

```
> cd to/dir/with/example_record
> redo build/obj/Linux/example_record.o
```

Two source code files are autocoded: *build/src/example_record.ads* and *build/src/example_record.adb*. Below is the specification file, *build/src/example_record.ads*.

```

1 -----
2 -- Example_Record Record Spec
3 --
4 -- Generated from example_record.record.yaml on 2025-07-15 21:46.
5 -----
6
7 -- Standard Includes:
8 with Basic_Types;
9 with System;
10 with Serializer_Types; use Serializer_Types;
11 with Serializer;
12
13 -- This is an example packed record.
14 package Example_Record is
15
16   -- Preamble code:
17   type Five_Bit_Integer is mod 2**5;
18   subtype Three_Bit_Signed_Integer is Integer range -3 .. 2;
19   type My_Color is (Red, Green, Blue, Purple, Pink, Chartreuse);
```

```

20
21 -- Packed type size (in bits):
22 Size : constant Positive := 48;
23
24 -- Packed type size rounded up to nearest byte.
25 Size_In_Bytes : constant Positive := (Size - 1) /
26   Basic_Types.Byte'Object_Size + 1;
27
28 -- The total number of fields contained in the packed record, this includes
29 -- any fields of packed records included directly as a member in this
29 -- packed record.
30 Num_Fields : constant Positive := 4;
31
32 -- Unpacked type:
33 type U is record
34   -- This is the first value.
35   Value_1 : Five_Bit_Integer := 0;
36   -- This is the second value.
37   Value_2 : Three_Bit_Signed_Integer;
38   Value_3 : My_Color;
39   -- This is the fourth value.
40   Value_4 : Short_Float;
41 end record;
42
43 -- Access type for U
44 type U_Access is access all U;
45
46 -- Packed type definition.
47 type T is new U
48   with Bit_Order => System.High_Order_First,
49     Scalar_Storage_Order => System.High_Order_First,
50     Size => Size,
51     Object_Size => Size,
52     Value_Size => Size,
53     Alignment => 1,
54     Volatile => False;
55
56 -- Packed type layout:
57 for T use record
58   -- This is the first value.
59   Value_1 at 0 range 0 .. 4;
60   -- This is the second value.
61   Value_2 at 0 range 5 .. 7;
62   Value_3 at 0 range 8 .. 15;
63   -- This is the fourth value.
64   Value_4 at 0 range 16 .. 47;
65 end record;
66
67 -- Access type for T
68 type T_Access is access all T;
69
70 -- Packed type definition with little endian definition.
71 type T_Le is new U
72   with Bit_Order => System.Low_Order_First,
73     Scalar_Storage_Order => System.Low_Order_First,
74     Size => Size,
75     Object_Size => Size,
76     Value_Size => Size,
77     Alignment => 1,
78     Volatile => False;
79

```

```

80  -- Packed type layout:
81  for T_Le use record
82      -- This is the first value.
83      Value_1 at 0 range 0 .. 4;
84      -- This is the second value.
85      Value_2 at 0 range 5 .. 7;
86      Value_3 at 0 range 8 .. 15;
87      -- This is the fourth value.
88      Value_4 at 0 range 16 .. 47;
89  end record;
90
91  -- Access type for T_Le
92  type T_Le_Access is access all T_Le;
93
94  -- Volatile packed type definition:
95  -- Note: This type is volatile. You should use this type to specify that the
96  -- variable in question may suddenly change in value. For example, this may
97  -- occur due to a device writing to a shared buffer. When this pragma is
98  -- used,
99  -- the compiler must suppress any optimizations that would interfere with the
100 -- correct reading of the volatile variables. For example, two successive
101 -- readings of the same variable cannot be optimized to just one or
102 -- reordered.
103 -- Important: You should use the Register type for accessing hardware
104 -- registers.
105 type Volatile_T is new T
106     with Bit_Order => System.High_Order_First,
107         Scalar_Storage_Order => System.High_Order_First,
108         Size => Size,
109         Object_Size => Size,
110         Value_Size => Size,
111         Alignment => 1,
112         Volatile => True;
113
114  -- Access type for Volatile_T
115  type Volatile_T_Access is access all Volatile_T;
116
117  -- Volatile little endian packed type definition:
118  -- Note: This type is volatile. You should use this type to specify that the
119  -- variable in question may suddenly change in value. For example, this may
120  -- occur due to a device writing to a shared buffer. When this pragma is
121  -- used,
122  -- the compiler must suppress any optimizations that would interfere with the
123  -- correct reading of the volatile variables. For example, two successive
124  -- readings of the same variable cannot be optimized to just one or
125  -- reordered.
126  -- Important: You should use the Register type for accessing hardware
127  -- registers.
128 type Volatile_T_Le is new T_Le
129     with Bit_Order => System.Low_Order_First,
130         Scalar_Storage_Order => System.Low_Order_First,
131         Size => Size,
132         Object_Size => Size,
133         Value_Size => Size,
134         Alignment => 1,
135         Volatile => True;
136
137  -- Access type for Volatile_T_Le
138  type Volatile_T_Le_Access is access all Volatile_T_Le;
139
140  -- Not supported. This type is 48 bits in size.

```

```

135  -- Type conversion functions between packed an unpacked representations:
136  function Pack (Src : in U) return T with Inline => True;
137  function Unpack (Src : in T) return U with Inline => True;
138  function Pack (Src : in U) return T_Le with Inline => True;
139  function Unpack (Src : in T_Le) return U with Inline => True;
140
141  -- Endianness conversion functions
142  function Swap_Endianness (Src : in T) return T_Le with Inline => True;
143  function Swap_Endianness (Src : in T_Le) return T with Inline => True;
144
145  -- Serializing functions for entire record:
146  package Serialization is new Serializer (T);
147  package Serialization_Le is new Serializer (T_Le);
148
149  -- The length in bytes of the serialized type.
150  Max_Serialized_Length : Natural renames Serialization.Serialized_Length; --
151  -- in bytes
152  -- The length in bytes of the serialized type.
153  Min_Serialized_Length : Natural renames Serialization.Serialized_Length; --
154  -- in bytes
155  -- Convenience function which always returns Success and the length defined
156  -- above ^. This
157  -- is useful when you want to instantiate a generic which requires a function
158  -- of the
159  -- definition below, but its for a statically sized type.
160  function Serialized_Length (Src : in T; Num_Bytes_Serialized : out Natural)
161  -- return Serialization_Status
162  -- with Inline => True;
163  function Serialized_Length_Le (Src : in T_Le; Num_Bytes_Serialized : out
164  -- Natural) return Serialization_Status
165  -- with Inline => True;
166
167  -- Get the size of the already serialized type inside of the byte array. If
168  -- the byte array is too small then
169  -- a serialization error is returned.
170  function Serialized_Length (Src : in Basic_Types.Byte_Array;
171  -- Num_Bytes_Serialized : out Natural) return Serialization_Status
172  -- with Inline => True;
173  function Serialized_Length_Le (Src : in Basic_Types.Byte_Array;
174  -- Num_Bytes_Serialized : out Natural) return Serialization_Status
175  -- with Inline => True;
176
177  -----
178  -- Types related to record's internal fields:
179  -----
180  -- Field type sizes (in bits):
181  Value_1_Size : constant Positive := 5;
182  Value_2_Size : constant Positive := 3;
183  Value_3_Size : constant Positive := 8;
184  Value_4_Size : constant Positive := 32;
185
186  -- Packed field sizes in bytes rounded up to nearest byte boundary.
187  Value_1_Size_In_Bytes : constant Positive := (Value_1_Size - 1) /
188  -- Basic_Types.Byte'Object_Size + 1;
189  Value_2_Size_In_Bytes : constant Positive := (Value_2_Size - 1) /
190  -- Basic_Types.Byte'Object_Size + 1;
191  Value_3_Size_In_Bytes : constant Positive := (Value_3_Size - 1) /
192  -- Basic_Types.Byte'Object_Size + 1;
193  Value_4_Size_In_Bytes : constant Positive := (Value_4_Size - 1) /
194  -- Basic_Types.Byte'Object_Size + 1;

```

```

183
184 -----
185 -- Named subtypes for each field for convenience:
186 -----
187
188 subtype Value_1_Type is Five_Bit_Integer;
189 subtype Value_2_Type is Three_Bit_Signed_Integer;
190 subtype Value_3_Type is My_Color;
191 subtype Value_4_Type is Short_Float;
192
193 end Example_Record;

```

A record is produced with 4 fields as expected. The U type will respect the Ada *type* given in the YAML file, and any range or default value associated with it. The representation of this type in memory is completely up to the compiler. The unpacked type should be used whenever the type is used in heavy calculations, since the compiler will be able to optimize the program for speed instead of spending time worrying about storage space and endianness. If the representation or size of the type is of concern, like when sending the data through a connector or to an external system, the T type should be used. This is a specialization of the U type which specifies exactly how the record should be laid out in memory. The size of each field is calculated, and a big endian representation is requested.

Some examples of how to use the specification are presented in the following two sections.

5.1.2 Packed Record Type Conversion

Converting a record between the T, T_Le, and U types can be done via the Pack, Unpack, and Swap_Endianness functions, as shown in the following example:

```

1 with Example_Record;
2
3 procedure Main is
4   use Example_Record;
5   -- A packed big-endian version of the type.
6   Packed_Be_Type : Example_Record.T := (Value_1 => 2, Value_2 => -1, Value_3 =>
7     Green, Value_4 => 0.5);
8   -- A packed little-endian version of the type.
9   Packed_Le_Type : Example_Record.T_Le;
10  -- An unpacked version of the type.
11  Unpacked_Type : Example_Record.U;
12
13 begin
14   -- Convert from big-endian packed to unpacked:
15   Unpacked_Type := Unpack (Packed_Be_Type);
16   -- Convert from unpacked to big-endian packed:
17   Packed_Be_Type := Pack (Unpacked_Type);
18   -- Convert from big-endian to little-endian:
19   Packed_Le_Type := Swap_Endianness (Packed_Be_Type);
20   -- Convert from little-endian to big-endian:
21   Packed_Be_Type := Swap_Endianness (Packed_Le_Type);
22   -- Convert from little-endian packed to unpacked:
23   Unpacked_Type := Unpack (Packed_Le_Type);
24
25 end Main;

```

5.1.3 Packed Record Serialization

Also included in the specification is the `Serialization` package which can be used to convert the packed record to and from an array of bytes, which is a very common operation in embedded systems. Here is an example:

```

1  with Example_Record;
2
3  procedure Main is
4    use Example_Record;
5    -- A packed version of the type.
6    Packed_Type : Example_Record.T := (Value_1 => 2, Value_2 => -1, Value_3 =>
7      ↳ Green, Value_4 => 0.5);
8    -- A byte array that is the same size as the type.
9    Byte_Array : Example_Record.Serialization.Byte_Array;
10   begin
11     -- Copy from the packed record to a byte array.
12     Byte_Array := Example_Record.Serialization.To_Byte_Array (Packed_Type);
13     -- Copy from a byte array to a packed record.
14     Packed_Type := Example_Record.Serialization.From_Byte_Array (Byte_Array);
15   end Main;

```

Note also that with every packed record there are a series of `Serialized_Length` subprograms. These subprograms allow the user to determine the length of a type (in bytes). While the length of the packed record presented in this section is static, meaning the function always returns the same value, variable length packed records return different values based on the type's content. See Section 5.1.11 for more details.

5.1.4 Packed Record Validation

Using the serialization and deserialization functions shown in the previous section can be error prone. Consider the field in our record `Value_2`, which only has a valid integer range between -2 and 3. It is possible that when calling the deserialization function `From_Byte_Array`, the incoming byte array might contain a bit representation that puts the `Value_2` field out of range, ie. 5. The call to `From_Byte_Array` will not fail, but any subsequent operation using the `Packed_Type` variable will fail with an Ada constraint error, due to the range check failing on `Value_2`.

Adamant packed records have a method for mitigating this using an autocoded child package which performs validation on the type. To build the validation child package run the following:

```
> redo build/obj/Linux/example_record-validation.o
```

This generates the source files `build/src/example_record-validation.ads` and `build/src/example_record-validation.adb`. The autocoded specification is shown below.

```

1  -----
2  -- Example_Record Record Validation Spec
3  --
4  -- Generated from example_record.record.yaml on 2025-07-15 21:46.
5  -----
6
7  with Interfaces;
8
9  -- Record validation package for Example_Record
10 package Example_Record.Validation is
11
12   -- We assume that for this autocoder to work, the "field" number for the
13   -- packed array takes less than 32-bits to represent:
14   use Interfaces;
15   pragma Compile_Time_Error (4 > Interfaces.Unsigned_32'Last,
16     "The autocoded validation functions assume a field size that can fit in
17     ↳ 32-bits. This record's field size is too large. Please write this file
18     ↳ by hand.");
19
20   -- Return True if the packed record is valid. The function performs

```

```

19   -- range checks on all the fields of the record. If a field is invalid,
20   -- False is returned and the Errant_Field parameter is filled in with a
21   -- Natural specifying which field was out of range.
22   function Valid (R : in U; Errant_Field : out Interfaces.Unsigned_32) return
23     Boolean;
24   function Valid (R : in T; Errant_Field : out Interfaces.Unsigned_32) return
25     Boolean;
26   function Valid (R : in T_Le; Errant_Field : out Interfaces.Unsigned_32)
27     return Boolean;
28
29   -- Return a field (provided by a field number) as a polymorphic type.
30   -- This is useful for returning any field in a record in a very generic
31   -- way. Fields bigger than the polymorphic type will only have their
32   -- least significant bits returned. This function should be used in tandem
33   -- with the Valid functions above to create useful error messages for an
34   -- invalid
35   -- type:
36   function Get_Field (Src : in T; Field : in Interfaces.Unsigned_32) return
37     Basic_Types.Poly_Type;
38   function Get_Field (Src : in T_Le; Field : in Interfaces.Unsigned_32) return
39     Basic_Types.Poly_Type;
40
41 end Example_Record.Validation;

```

Note that a Valid function which returns a Boolean is available to validate each version of the packed record. These functions can be called to determine if the type is within range, and thus safe to use, before using the type elsewhere in the system. The Adamant component autocoder uses the feature extensively to validate incoming commands and parameters prior to their use within user code.

Below is an example of using this package.

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Example_Record.Validation, Interfaces;
3
4  procedure Main is
5    use Example_Record, Interfaces;
6    -- A packed version of the type.
7    Packed_Type_1 : constant Example_Record.T := (Value_1 => 2, Value_2 => -1,
8      Value_3 => Green, Value_4 => 0.5);
9    -- Deserialize a byte array of all 0xFF into the record.
10   Packed_Type_2 : constant Example_Record.T :=
11     Example_Record.Serialization.From_Bye_Array ([255, 255, 255, 255, 255,
12       255]);
13   Errant_Field : Unsigned_32 := 0;
14
15 begin
16   if Example_Record.Validation.Valid (Packed_Type_1, Errant_Field) then
17     null; -- Should go here, since record is valid.
18   else
19     pragma Assert (False, "Record 1 is invalid at field " & Unsigned_32'Image
20                   & (Errant_Field) & "!");
21   end if;
22
23   if Example_Record.Validation.Valid (Packed_Type_2, Errant_Field) then
24     null; -- Should not go here, since record is not valid.
25   else
26     Put_Line ("Record 2 is invalid at field " & Unsigned_32'Image
27                   & (Errant_Field) & "!");
28   end if;
29 end Main;

```

which, when run, produces the output:

```
Record 2 is invalid at field 3!
```

which shows that a byte value of 255 is out of range for the enumeration type used for Value_3.

Note that Adamant currently struggles to support validation for certain Ada types including System.Address and all access types. The inclusion of these in packed records is rare in practice so this does not pose a critical issue. If the validation autocode for a certain field does not compile you have two options: 1) Don't use the autocoded package and instead hand-code your own or 2) If you know that the type will always be valid then use the Skip_Validation field modifier to specify that you want the autocoder to skip generating validation code for that field. Below is a packed record *memory_region.record.yaml* that uses Skip_Validation to avoid validating a troublesome address type.

```
1  ---
2  description: A memory region described by a system address and length (in
3  ← bytes).
4  fields:
5  - name: Address
6  description: The starting address of the memory region.
7  type: System.Address
8  format: U32
9  skip_validation: True
10 byte_image: True
11 - name: Length
12 description: The number of bytes at the given address to associate with this
13 ← memory region.
14 type: Natural
15 format: U32
```

If not specified the Skip_Validation defaults to False. The unusual Byte_Image modifier is also seen in this YAML file. See the Section 5.1.5 for more details.

5.1.5 Packed Record String Representation

During testing, it is often nice to print the value of a complex datatype in a human readable form. Adamant has the ability to generate human readable string representations for packed records. The string representation code for the example_record is included in two files, a specification *build/src/example_record-representation.ads* and *build/src/example_record-representation.adb* which can be generated and compiled using the following commands:

```
> redo build/obj/Linux/example_record-representation.o
```

This generates the source files *build/src/example_record-representation.ads* and *build/src/example_record-representation.adb*. The autocoded specification is shown below.

```
1 -----
2 -- Example_Record Record Representation Spec
3 --
4 -- Generated from example_record.yaml on 2025-07-15 21:46.
5 -----
6 --
7 -- Standard includes:
8 with String_Util; use String_Util;
9 --
10 -- String representation package for Example_Record
11 package Example_Record.Representation is
12 --
13 --
14 -- Common to string functions:
```

```

15 -----
16 -- Return string showing bytes in record:
17 function To_Byte_String is new String_Util.To_Byte_String (U);
18 function To_Byte_String is new String_Util.To_Byte_String (T);
19 function To_Byte_String is new String_Util.To_Byte_String (T_Le);
20
21 -- Display record as string:
22 function Image (R : in U) return String;
23 function Image (R : in T) return String;
24 function Image (R : in T_Le) return String;
25
26 -----
27 -- Less commonly used to string functions:
28 -----
29 -- Return string representation of record fields in form (field 1, field 2,
30 -- etc.)
31 function To_Tuple_String (R : in U) return String;
32 function To_Tuple_String (R : in T) return String;
33 function To_Tuple_String (R : in T_Le) return String;
34
35 -- Return string representation of record fields and bytes
36 function Image_With_Prefix (R : in U; Prefix : in String := "") return
37 -- String;
38 function Image_With_Prefix (R : in T; Prefix : in String := "") return
39 -- String;
40 function Image_With_Prefix (R : in T_Le; Prefix : in String := "") return
41 -- String;
42
43 -----
44 -- To string functions for record field type:
45 -----
46 function Value_1_To_Byte_String is new String_Util.To_Byte_String
47 -- (Five_Bit_Integer);
48 function Value_1_Image (R : in Five_Bit_Integer) return String renames
49 -- Five_Bit_Integer'Image;
50 function Value_1_To_Tuple_String (R : in Five_Bit_Integer) return String
51 -- renames Five_Bit_Integer'Image;
52 function Value_1_Image_With_Prefix (R : in Five_Bit_Integer) return String
53 -- renames Five_Bit_Integer'Image;
54
55 function Value_2_To_Byte_String is new String_Util.To_Byte_String
56 -- (Three_Bit_Signed_Integer);
57 function Value_2_Image (R : in Three_Bit_Signed_Integer) return String
58 -- renames Three_Bit_Signed_Integer'Image;
59 function Value_2_To_Tuple_String (R : in Three_Bit_Signed_Integer) return
60 -- String renames Three_Bit_Signed_Integer'Image;
61 function Value_2_Image_With_Prefix (R : in Three_Bit_Signed_Integer) return
62 -- String renames Three_Bit_Signed_Integer'Image;
63
64 function Value_3_To_Byte_String is new String_Util.To_Byte_String (My_Color);
65 function Value_3_Image (R : in My_Color) return String renames
66 -- My_Color'Image;
67 function Value_3_To_Tuple_String (R : in My_Color) return String renames
68 -- My_Color'Image;
69 function Value_3_Image_With_Prefix (R : in My_Color) return String renames
70 -- My_Color'Image;
71
72 function Value_4_To_Byte_String is new String_Util.To_Byte_String
73 -- (Short_Float);
74 function Value_4_Image (R : in Short_Float) return String renames
75 -- Short_Float'Image;

```

```

59      function Value_4_To_Tuple_String (R : in Short_Float) return String renames
60        Short_Float'Image;
61      function Value_4_Image_With_Prefix (R : in Short_Float) return String renames
62        Short_Float'Image;
63
64  end Example_Record.Representation;

```

There are many subprograms provided, but the most commonly used are `To_Byte_String`, `Image`, and `To_Tuple_String`. The first `To_Byte_String` function returns a string that shows the bytes that make up the packed record in hex. The second `Image` functions returns the most human readable representation of the packed record, printing each field name following by its value on a separate line. The `To_Tuple_String` is useful for logging, as the entire record is always printed, comma separated, on a single line.

Below is an example of using the package.

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Example_Record.Representation;
3
4  procedure Main is
5    use Example_Record;
6    -- A packed version of the type.
7    Packed_Type_1 : constant Example_Record.T := (Value_1 => 2, Value_2 => -1,
8      Value_3 => Green, Value_4 => 0.5);
9  begin
10   -- Print the packed record to the terminal in human readable form:
11   Put_Line ("Human readable:");
12   Put_Line (Example_Record.Representation.Image (Packed_Type_1));
13   Put_Line ("");
14   -- Print the packed record to the terminal in log friendly form:
15   Put_Line ("Log friendly:");
16   Put_Line (Example_Record.Representation.To_Tuple_String (Packed_Type_1));
17   Put_Line ("");
18 end Main;

```

which, when run, produces the output:

```

Human readable:
[17 01 3F 00 00 00]
Value_1 : Five_Bit_Integer => 2
Value_2 : Three_Bit_Signed_Integer => -1
Value_3 : My_Color => GREEN
Value_4 : Short_Float => 5.00000E-01

Log friendly:
(Value_1 => 2, Value_2 => -1, Value_3 => GREEN, Value_4 => 5.00000E-01)

```

Note that Adamant currently struggles to support pretty print strings for certain Ada types including `System.Address` and all access types. The inclusion of these in packed records is rare in practice so this does not pose a critical issue. If the string representation autocode for a certain field type is not to your liking, or does not compile you have two options: 1) Don't use the autocoded package and instead hand-code your own, see the `Sys_Time` type for an example of this, or 2) Use the `Byte_Image` field modifier to specify that you want the representation of the type to simply be an array of hex bytes. Since any type can be represented by its underlying machine byte values, option 2 will always compile, but may not be as human readable. Below is a packed record `memory_region.record.yaml` that uses `Byte_Image` to display a troublesome address type.

```

1  ---
2  description: A memory region described by a system address and length (in
3   → bytes).
4  fields:
5    - name: Address
6      description: The starting address of the memory region.
7      type: System.Address
8      format: U32
9      skip_validation: True
10     byte_image: True
11    - name: Length
12      description: The number of bytes at the given address to associate with this
13       → memory region.
14      type: Natural
15      format: U32

```

If not specified the `Byte_Image` defaults to `False`. The unusual `Skip_Validation` modifier is also seen in this YAML file. See the Section 5.1.4 for more details.

5.1.6 Packed Record Unit Test Assertions

Adamant also provides an assertion package for each packed record. The assertion package is useful during unit testing, as it allows you to compare two packed types. If the packed types do not meet the comparison criteria, then a human readable error message is printed. To build the assertion package run.

```
> redo build/obj/Linux/example_record-assertion.o
```

The `build/src/example_record-assertion.ads` and `build/src/example_record-assertion.adb` files are auto-coded. The specification is shown below.

```

1  -----
2  -- Example_Record Record Assertion Spec
3  --
4  -- Generated from example_record.record.yaml on 2025-07-15 21:46.
5  -----
6
7  -- Standard includes:
8  with Smart Assert;
9  with Example_Record.Representation;
10 with GNAT.Source_Info;
11
12 package Example_Record.Assertion is
13
14   package Sinfo renames GNAT.Source_Info;
15
16   -- Special assertion package for the variable length type. This package
17   -- does assertions, only comparing used data, ignoring data that is not
18   -- being used. For example, a type with a buffer that is only half filled
19   -- with valid data, as prescribed by that types variable length field, would
20   -- only be compared up to that length. Unused data in the buffer is ignored.
21   package Example_Record_U Assert is
22     procedure Eq (T1 : in U; T2 : in U; Epsilon : in Long_Float := 0.0;
23      → Message : in String := ""; Filename : in String := Sinfo.File; Line :
24      → in Natural := Sinfo.Line);
25     procedure Neq (T1 : in U; T2 : in U; Epsilon : in Long_Float := 0.0;
26      → Message : in String := ""; Filename : in String := Sinfo.File; Line :
27      → in Natural := Sinfo.Line);
28   end Example_Record_U Assert;
29

```

```

26 package Example_Record_Assert is
27     procedure Eq (T1 : in T; T2 : in T; Epsilon : in Long_Float := 0.0;
28         --> Message : in String := ""; Filename : in String := Sinfo.File; Line :
29         --> in Natural := Sinfo.Line);
30     procedure Neq (T1 : in T; T2 : in T; Epsilon : in Long_Float := 0.0;
31         --> Message : in String := ""; Filename : in String := Sinfo.File; Line :
32         --> in Natural := Sinfo.Line);
33 end Example_Record_Assert;
34
35 package Example_Record_Le_Assert is
36     procedure Eq (T1 : in T_Le; T2 : in T_Le; Epsilon : in Long_Float := 0.0;
37         --> Message : in String := "";Filename : in String := Sinfo.File; Line :
38         --> in Natural := Sinfo.Line);
39     procedure Neq (T1 : in T_Le; T2 : in T_Le; Epsilon : in Long_Float := 0.0;
40         --> Message : in String := "";Filename : in String := Sinfo.File; Line :
41         --> in Natural := Sinfo.Line);
42 end Example_Record_Le_Assert;
43
44 -- This package compares data without any Epsilon, using the
45 -- Smart Assert.Basic package.
46 package Example_Record_U_Assert_All is new Smart_Assert.Basic
47     -- (Example_Record.U, Example_Record.Representation.Image);
48 package Example_Record_Assert_All is new Smart_Assert.Basic
49     -- (Example_Record.T, Example_Record.Representation.Image);
50 package Example_Record_Le_Assert_All is new Smart_Assert.Basic
51     -- (Example_Record.T_Le, Example_Record.Representation.Image);
52
53 -- Specialized smart assert package for the fields in this record:
54 package Value_1_Assert is new Smart_Assert.Basic (Five_Bit_Integer,
55     -- Example_Record.Representation.Value_1_Image);
56 package Value_2_Assert is new Smart_Assert.Basic (Three_Bit_Signed_Integer,
57     -- Example_Record.Representation.Value_2_Image);
58 package Value_3_Assert is new Smart_Assert.Basic (My_Color,
59     -- Example_Record.Representation.Value_3_Image);
60 package Value_4_Assert is new Smart_Assert.Float (Short_Float,
61     -- Example_Record.Representation.Value_4_Image);
62
63 end Example_Record.Assertion;

```

The package uses the Adamant Smart_Assert package and can be used as follows:

```

1  with Example_Record.Assertion;
2
3  procedure Main is
4      use Example_Record;
5      use Example_Record.Assertion;
6      -- A packed version of the type.
7      Packed_Type_1 : constant Example_Record.T := (Value_1 => 2, Value_2 => -1,
8          -- Value_3 => Green, Value_4 => 0.5);
9      Packed_Type_2 : constant Example_Record.T := (Value_1 => 5, Value_2 => -1,
10         -- Value_3 => Green, Value_4 => 0.4);
11
12 begin
13     -- See if the records are not equal (should pass):
14     Example_Record_Assert.Neq (Packed_Type_1, Packed_Type_2);
15
16     -- See if the records are equal (should fail):
17     Example_Record_Assert.Eq (Packed_Type_1, Packed_Type_2);
18
19 end Main;

```

which, when run, produces the output:

```

Assertion:
2
=
5
failed.
Message: Assertion failed for field Value_1 while comparing Example_Record below.
Assertion:
2
=
5
failed.
Message: Assertion failed for field Value_1 while comparing Example_Record below.
Assertion:
[17 01 3F 00 00 00]
Value_1 : Five_Bit_Integer => 2
Value_2 : Three_Bit_Signed_Integer => -1
Value_3 : My_Color => GREEN
Value_4 : Short_Float => 5.00000E-01
=
[2F 01 3E CC CC CD]
Value_1 : Five_Bit_Integer => 5
Value_2 : Three_Bit_Signed_Integer => -1
Value_3 : My_Color => GREEN
Value_4 : Short_Float => 4.00000E-01
failed.

raised ADA ASSERTIONS ASSERTION_ERROR : at main.adb:14

```

5.1.7 Packed Record Documentation

Adamant provides automatic generation of documentation for any packed record definition. Currently two versions of documentation can be created: HTML, which is useful for presentation in meetings or for quick reference, and PDF (via L^AT_EX), which is useful for formal documentation.

To build HTML documentation run:

```
> redo build/html/example_record.html
```

which when opened with your favorite web browser looks something like this:

Example_Record Packed Record : 48 bits

Description: This is an example packed record.

Preamble:

```
type Five_Bit_Integer is mod 2**5;
subtype Three_Bit_Signed_Integer is Integer range -3 .. 2;
type My_Color is (Red, Green, Blue, Purple, Pink, Chartreuse);
```

Name	Type	Range	Size (Bits)	Start Bit	End Bit	Description
Value_1	Five_Bit_Integer	0 to 31	5	0	4	This is the first value.
Value_2	Three_Bit_Signed_Integer	-3 to 2	3	5	7	This is the second value.
Value_3	My_Color	0 => Red 1 => Green 2 => Blue 3 => Purple 4 => Pink 5 => Chartreuse	8	8	15	
Value_4	Short_Float	-3.40282e+38 to 3.40282e+38	32	16	47	This is the fourth value.

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/example_record.record.yaml on 2020-04-20 22:04.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

To build the PDF documentation run:

```
> redo build/pdf/example_record.pdf
```

Which produces a PDF output that looks like the following.

This is an example packed record. *Preamble (inline Ada definitions):*

```
1 type Five_Bit_Integer is mod 2**5;
2 subtype Three_Bit_Signed_Integer is Integer range -3 .. 2;
3 type My_Color is (Red, Green, Blue, Purple, Pink, Chartreuse);
```

Table 3: Example_Record Packed Record : 48 bits

Name	Type	Range	Size (Bits)	Start Bit	End Bit
Value_1	Five_Bit_Integer	0 to 31	5	0	4
Value_2	Three_Bit_Signed_Integer	-3 to 2	3	5	7
Value_3	My_Color	0 => Red 1 => Green 2 => Blue 3 => Purple 4 => Pink 5 => Chartreuse	8	8	15
Value_4	Short_Float	-3.40282e+38 to 3.40282e+38	32	16	47

Field Descriptions:

- **Value_1** - This is the first value.
 - **Value_2** - This is the second value.
 - **Value_3** - *No description provided.*
 - **Value_4** - This is the fourth value.
-

5.1.8 Packed Record Python Class

For each packed type, Adamant provides a python class which supplies analogous operations to the Ada serialization, representation, and assertion packages presented in the previous sections. For space-based applications, these python classes are meant as “ground” tools, or tools which assist with some task related to the deployed embedded system such as testing, parsing data, sending commands, etc. See Section 8 for more details. To build the python class for a record run:

```
> redo build/py/example_record.py
```

The resulting autocode is shown below:

```

1 ######
2 # Example_Record Record
3 #
4 # Generated from example_record.record.yaml on 2025-07-15 21:46.
5 #####
6
7 from base_classes.packed_type_base import PackedTypeBase
8 from bitstring import BitArray
9
10 # Internal packed type imports:
11
12
13 # This is an example packed record.
14 class Example_Record(PackedTypeBase):
15
16     def __init__(
17         self,
18         Value_1=0,
19         Value_2=None,
20         Value_3=None,
21         Value_4=None
22     ):
23         # Fields:
24         # Check and set Value_1:
25         # This is the first value.
26         if Value_1 is not None:
27             assert isinstance(Value_1, int), \
28                 ("Expected type for field "
29                  "'Value_1' "
30                  "to be "
31                  "'int' "
32                  "and instead found '"
33                  + str(type(Value_1)))
34             assert Value_1 >= 0, \
35                 ("Value_1 "
36                  "is unsigned and must be positive. "
37                  "Received value of '"
38                  + str(Value_1) + "' "
39                  "is not allowed.")
40             self.Value_1 = \

```

```

41         Value_1
42     # Check and set Value_2:
43     # This is the second value.
44     if Value_2 is not None:
45         assert isinstance(Value_2, int), \
46             ("Expected type for field "
47             "'Value_2' "
48             "to be "
49             "'int' "
50             "and instead found '"
51             + str(type(Value_2)))
52     self.Value_2 = \
53         Value_2
54     # Check and set Value_3:
55     if Value_3 is not None:
56         assert isinstance(Value_3, int), \
57             ("Expected type for field "
58             "'Value_3' "
59             "to be "
60             "'int' "
61             "and instead found '"
62             + str(type(Value_3)))
63     self.Value_3 = \
64         Value_3
65     # Check and set Value_4:
66     # This is the fourth value.
67     if Value_4 is not None:
68         assert isinstance(Value_4, float), \
69             ("Expected type for field "
70             "'Value_4' "
71             "to be "
72             "'float' "
73             "and instead found '"
74             + str(type(Value_4)))
75     self.Value_4 = \
76         Value_4
77
78     # Sizes:
79     self._size = 48    # in bits
80     self._size_in_bytes = int(int(self._size - 1)/8 + 1)    # in bytes
81     self._min_serialized_length_in_bits = self._size    # in bits
82     self._max_serialized_length_in_bits = self._size    # in bits
83     self._min_serialized_length = self._size_in_bytes    # in bytes
84     self._max_serialized_length = self._size_in_bytes    # in bytes
85
86     # Other attributes:
87     self.num_fields = 4
88
89     def __eq__(self, other):
90         return isinstance(other, self.__class__) and \
91             self.Value_1 == \
92             other.Value_1 and \
93             self.Value_2 == \
94             other.Value_2 and \
95             self.Value_3 == \
96             other.Value_3 and \
97             self.Value_4 == \
98             other.Value_4
99
100    def serialized_length(self):    # in bytes
101        return self._size_in_bytes

```

```

102
103     def _from_byte_array(self, stream):
104         # Extract each field from the bitstream:
105         _Value_1 = stream.read("uint:5")
106         _Value_2 = stream.read("int:3")
107         _Value_3 = stream.read("uint:8")
108         _Value_4 = stream.read("floatbe:32")
109         # Fill in self:
110         Example_Record.__init__(
111             self=self,
112             Value_1=_Value_1,
113             Value_2=_Value_2,
114             Value_3=_Value_3,
115             Value_4=_Value_4
116         )
117
118     def _to_byte_array(self):
119         bits = BitArray()
120         bits += BitArray(
121             "uint:5=" +
122             str(self.Value_1
123
124                 if self.Value_1 is not None else "0")
125         )
126         bits += BitArray(
127             "int:3=" +
128             str(self.Value_2
129
130                 if self.Value_2 is not None else "0")
131         )
132         bits += BitArray(
133             "uint:8=" +
134             str(self.Value_3
135
136                 if self.Value_3 is not None else "0")
137         )
138         bits += BitArray(
139             "floatbe:32=" +
140             str(self.Value_4
141
142                 if self.Value_4 is not None else "0")
143         )
144         # Make sure the bit array is the correct size:
145         assert len(bits) <= self._max_serialized_length_in_bits, \
146             "Expected: " + str(len(bits)) + " <= " +
147             str(self._max_serialized_length_in_bits)
148         assert len(bits) >= self._min_serialized_length_in_bits, \
149             "Expected: " + str(len(bits)) + " >= " +
150             str(self._min_serialized_length_in_bits)
151     return bits
152
153     def to_string(self, prefix=""):
154         strn = prefix + self.to_byte_string() + "\n"
155         strn += prefix + "Value_1 : Five_Bit_Integer => " + \
156             str(self.Value_1)
157         strn += "\n"
158         strn += prefix + "Value_2 : Three_Bit_Signed_Integer => " + \
159             str(self.Value_2)
160         strn += "\n"
161         strn += prefix + "Value_3 : My_Color => " + \
162             str(self.Value_3)

```

```

161     strn += "\n"
162     strn += prefix + "Value_4 : Short_Float => " + \
163         str(self.Value_4)
164     strn += "\n"
165     return strn[:-1]
166
167     def to_tuple_string(self):
168         strn = "("
169         strn += "Value_1 => " + \
170             str(self.Value_1)
171         strn += ", "
172         strn += "Value_2 => " + \
173             str(self.Value_2)
174         strn += ", "
175         strn += "Value_3 => " + \
176             str(self.Value_3)
177         strn += ", "
178         strn += "Value_4 => " + \
179             str(self.Value_4)
180         strn += ","
181         return strn[:-2] + ")"

```

The python classes use the `bitstring` python module which allows easily manipulation of an object that acts like an array of bits. The python class contains the following methods - note that some of the methods below are actually implemented in the python packed type base class `PackedTypeBase`, from which the autocode inherits:

- **`__init__`** - Create a packed record. The record field values are optional upon initialization, and will be set to the default or `None` if there is no default. During initialization, the class performs extensive type checking of the record fields to make sure that their values adhere to ranges of the equivalent Ada types.
- **`__eq__`** - Compares two packed records for equality.
- **`serialized_length`** - Equivalent to the Ada `Serialized_Length` function, returns the length of the packed type.
- **`to_byte_array` and `from_byte_array`** - Equivalent to the Ada serialization/deserialization functions `Serialization.To_Byte_Array` and `Serialization.From_Byte_Array`; converts the python class to a serialized byte array whose bits correspond to the layout of the packed big-endian type (`T`) in Ada, or takes a byte array of this form and turns it into a python class.
- **`create_from_byte_array`** - Creates a new packed record object from a serialized byte array.
- **`__repr__`, `__str__` and `to_string`** - Equivalent to the Ada `Representation.Image` function; creates a human readable string version of the packed record.
- **`to_tuple_string`** - Equivalent to the Ada `Representation.To_Tuple_String` function; creates a log-friendly string of the packed record.

An example of how this class can be used is shown below in a file called `test.py`:

```

1 #!/usr/bin/env python3
2
3 # Build our dependencies using the build system.
4 # This is necessary because some of the dependencies we
5 # have are autogenerated.
6 from util import pydep
7
8 pydep.build_py_deps()
9

```

```

10 # Now that the autocode has been built, import
11 # the autocoded module.
12 from example_record import Example_Record
13
14 # Main:
15 if __name__ == "__main__":
16     print("create and print example record:")
17     example1 = Example_Record(Value_1=2, Value_2=-1, Value_3=1, Value_4=0.5)
18     print(str(example1))
19     print()
20     print(str(example1.to_tuple_string()))
21     print()
22     print(str(example1.to_byte_array().hex()))
23     print()
24     print(str(example1.to_byte_string()))
25     print()
26
27     print("serializing and deserializing example record:")
28     data = example1.to_byte_array()
29     example2 = Example_Record.create_from_byte_array(data)
30     assert example1 == example2
31     print("done.")

```

Note the call at the top of the script to `pydep.build_py_deps()`. This Adamant python module does two useful things: 1) it searches the python program for any autocoded dependencies and uses `redo` to build them and 2) adds any autocoded dependencies automatically to the `PYTHONPATH` so they can be imported without error. It is important to run this function as the first line of any Adamant python script that will use autocoded python classes.

The script can be run with the command:

```
> python test.py
```

and produces the following output:

```

create and print example record:
[17 01 3F 00 00 00]
Value_1 : Five_Bit_Integer => 2
Value_2 : Three_Bit_Signed_Integer => -1
Value_3 : My_Color => 1
Value_4 : Short_Float => 0.5

(Value_1 => 2, Value_2 => -1, Value_3 => 1, Value_4 => 0.5)

17013f000000

[17 01 3F 00 00 00]

serializing and deserializing example record:
done.

```

Python classes can also be autogenerated for variable-length packed records and packed arrays, presented in the following sections.

5.1.9 Packed Record MATLAB Class

For each packed type, Adamant provides a MATLAB class which supplies analogous operations to the Ada and python serialization, representation, and assertion packages presented in the previous

sections. For space-based applications, these MATLAB classes are meant as “ground” tools, or tools which assist with some task related to the deployed embedded system such as testing, parsing data, sending commands, etc. See Section 8 for more details. To build the MATLAB class for a record run:

```
> redo build/m/Example_Record.m
```

The resulting autocode is shown below:

```

1 %%%%%%
2 %% Example_Record Record
3 %%
4 %% Generated from example_record.record.yaml on 2025-07-15 21:58.
5 %%%%%%
6
7 classdef Example_Record
8     % This is an example packed record.
9
10    properties
11        Value_1
12        Value_2
13        Value_3
14        Value_4
15        size
16        size_in_bytes
17        min_serialized_length_in_bits
18        max_serialized_length_in_bits
19        min_serialized_length
20        max_serialized_length
21        num_fields
22    end
23
24    methods
25        function self = Example_Record(Value_1, Value_2, Value_3, Value_4)
26            % Constructor
27
28            % If no arguments are provided, the return an empty (zero version) of the
29            % type:
30            if nargin == 0
31                self = Example_Record.create_empty();
32                return
33            end
34
35            % If more than zero arguments provided, make sure an argument was provided
36            % for each
37            % field in the packed type, and no more.
38            if nargin ~= 4
39                error("Example_Record expects 4 arguments but only got " + nargin +
40                    ".");
41            end
42
43            %
44            % Construct and instance of the class by filling in the object's
45            % properties.
46
47            % Check and set Value_1:
48            % This is the first value.
49            if ~isnumeric(Value_1)
50                error("Expected type for field 'Value_1' to be numeric and instead found
51                    item of type '" + class(Value_1) + "'.");
52            end
53
54    end
55
```

```

48     end
49     if Value_1 - round(Value_1) ~= 0
50         error("Expected type for field 'Value_1' to be an integer type and
51             instead found item of type '" + class(Value_1) + "'.");
52     end
53     if Value_1 > intmax('uint8')
54         error("Expected type for field 'Value_1' to be an integer type with a
55             max value of " + intmax('uint8') + " but a value of '" + Value_1 +
56             "' was provided.");
57     end
58     self.Value_1 = uint8(Value_1);
59
60     % Check and set Value_2:
61     % This is the second value.
62     if ~isnumeric(Value_2)
63         error("Expected type for field 'Value_2' to be numeric and instead found
64             item of type '" + class(Value_2) + "'.");
65     end
66     if Value_2 - round(Value_2) ~= 0
67         error("Expected type for field 'Value_2' to be an integer type and
68             instead found item of type '" + class(Value_2) + "'.");
69     end
70     if Value_2 > intmax('int8')
71         error("Expected type for field 'Value_2' to be an integer type with a
72             max value of " + intmax('int8') + " but a value of '" + Value_2 + "'"
73             " was provided.");
74     end
75     self.Value_2 = int8(Value_2);
76
77     % Check and set Value_3:
78     if ~isnumeric(Value_3)
79         error("Expected type for field 'Value_3' to be numeric and instead found
80             item of type '" + class(Value_3) + "'.");
81     end
82     self.Value_3 = int8(Value_3);
83
84     % Check and set Value_4:
85     % This is the fourth value.
86     if ~isnumeric(Value_4)
87         error("Expected type for field 'Value_4' to be numeric and instead found
88             item of type '" + class(Value_4) + "'.");
89     end
90     self.Value_4 = single(Value_4);
91
92     % Set the size properties
93     self.size = 48;
94     self.size_in_bytes = Example_Record.length();
95     self.min_serialized_length_in_bits = self.size; % in bits
96     self.max_serialized_length_in_bits = self.size; % in bits
97     self.min_serialized_length = self.size_in_bytes; % in bytes
98     self.max_serialized_length = self.size_in_bytes; % in bytes
99
100    % Other attributes:
101    self.num_fields = 4;
102
103
104    function pred = eq(self, other)

```

```

99      % Equality operator, ie. "=="  

100     pred = length(class(self)) == length(class(other));  

101     if pred  

102         pred = all(class(self) == class(other)) && ...  

103             all(self.Value_1 == other.Value_1) && ...  

104             all(self.Value_2 == other.Value_2) && ...  

105             all(self.Value_3 == other.Value_3) && ...  

106             all(self.Value_4 == other.Value_4);  

107     end  

108 end  

109  

110 function pred = ne(self, other)  

111     % Inequality operator, ie. "~="  

112     pred = ~eq(self, other);  

113 end  

114  

115 function len_in_bytes = serialized_length(self) % in bytes  

116     % Return the length of this object when serialized into an array of bytes  

117     len_in_bytes = self.size_in_bytes;  

118 end  

119  

120 function self = from_byte_array(~, bytes)  

121     % Deserialize the object from a flat array of bytes into the class  

122     % properties. This  

123     % one operates on self.  

124     self = Example_Record.create_from_byte_array(bytes);  

125 end  

126  

127 function bytes = to_byte_array(self)  

128     % Serialize the object properties into a flat array of bytes  

129  

130     % Create a Bit_Array class to serialize our data into a byte array:  

131     ba = Bit_Array();  

132     ba = ba.zeros(self.size_in_bytes);  

133  

134     % Serialize Value_1:  

135     ba = ba.set(1, 5, self.Value_1);  

136  

137     % Serialize Value_2:  

138     ba = ba.set(6, 8, self.Value_2);  

139  

140     % Serialize Value_3:  

141     ba = ba.set(9, 16, self.Value_3);  

142  

143     % Serialize Value_4:  

144     ba = ba.set(17, 48, self.Value_4);  

145  

146     % Return the byte array.  

147     bytes = ba.bytes();  

148 end  

149  

150 function strn = to_byte_string(self)  

151     % Return a print out of the bytes in the serialized class.  

152     bytes = self.to_byte_array();  

153     strn = "[";  

154     for idx=1:length(bytes)  

155         strn = strn + sprintf("%02x ", bytes(idx));  

156     end  

157     strn = strtrim(strn) + "]";  

158 end

```

```

159 function strn = to_string(self, prefix)
160     % Return a human readable string representation of the object.
161     if nargin < 2
162         prefix = "";
163     end
164
165     % Add the byte string to start:
166     strn = self.to_byte_string();
167
168     % Append human readable items:
169     strn = prefix + strn + newline + ...
170             prefix + "Value_1 : Five_Bit_Integer => " + ...
171             self.Value_1 + ...
172             newline + ...
173             prefix + "Value_2 : Three_Bit_Signed_Integer => " + ...
174             self.Value_2 + ...
175             newline + ...
176             prefix + "Value_3 : My_Color => " + ...
177             self.Value_3 + ...
178             newline + ...
179             prefix + "Value_4 : Short_Float => " + ...
180             self.Value_4 + ...
181             '';
182
183 end
184
185 function strn = to_tuple_string(self)
186     % Return a comma separated representation of the object.
187     strn = '(' + ...
188             "Value_1 => " + ...
189             self.Value_1 + ...
190             ', ' + ...
191             "Value_2 => " + ...
192             self.Value_2 + ...
193             ', ' + ...
194             "Value_3 => " + ...
195             self.Value_3 + ...
196             ', ' + ...
197             "Value_4 => " + ...
198             self.Value_4 + ...
199             ')';
200
201 end
202
203 methods(Static)
204     function self = create_from_byte_array(bytes)
205         % Deserialize the object from a flat array of bytes into the class
206         % properties.
207
208         % Check type and size of incoming byte array:
209         if class(bytes) ~= "uint8"
210             error("type of 'bytes' must be an array of 'uint8', instead bytes is of
211                   type '" + class(bytes) + "'");
212         end
213
214         if length(bytes) < Example_Record.min_length()
215             error("size of 'bytes' (" + length(bytes) + ") too small to extract
216                   Example_Record (must be at least " + Example_Record.min_length() +
217                   ")");
218         end
219
220         % Create a Bit_Array class to deserialize the bytes into the
221         % class properties:

```

```

216     ba = Bit_Array(bytes);
217
218     % Deserialize Value_1:
219     Value_1_temp = ba.extract(1, 5, 'uint8');
220
221     % Deserialize Value_2:
222     Value_2_temp = ba.extract(6, 8, 'int8');
223
224     % Deserialize Value_3:
225     Value_3_temp = ba.extract(9, 16, 'int8');
226
227     % Deserialize Value_4:
228     Value_4_temp = ba.extract(17, 48, 'single');
229
230     % Fill in self:
231     self = Example_Record( ...
232         Value_1_temp, ...
233         Value_2_temp, ...
234         Value_3_temp, ...
235         Value_4_temp ...
236     );
237 end
238
239 function self = create_empty()
240     % Create an empty version of the packed type.
241     self = Example_Record.create_from_byte_array(zeros(1,
242         ↳ Example_Record.min_length() + uint64(0), 'uint8'));
243 end
244
245 function size_in_bytes = length()
246     % Get the size (maximum) of the packed type in bytes
247     size_in_bytes = Example_Record.max_length();
248 end
249
250 function size_in_bytes = max_length()
251     % Get the size (maximum) of the packed type in bytes
252     size_in_bytes = idivide(uint64(48 - 1), uint64(8)) + uint64(1); % in bytes
253 end
254
255 function size_in_bytes = min_length()
256     % Get the size (minimum) of the packed type in bytes
257     size_in_bytes = idivide(uint64(48 - 1), uint64(8)) + uint64(1); % in bytes
258 end
259 end

```

The MATLAB classes use the `Bit_Array` MATLAB class, found in `gnd/matlab/Bit_Array.m`, which allows easily manipulation of an object that acts like an array of bits. Each autocoded MATLAB class uses `Bit_Array` to provide the following methods:

- **Example_Record** - The constructor used to create a packed record object. The record field values are optional required initialization. The class performs extensive type checking of the record fields to make sure that their values adhere to ranges of the equivalent Ada types, where possible.
- **eq** - Compares two packed records for equality.
- **ne** - Opposite of `eq`.
- **serialized_length** - Equivalent to the Ada `Serialized_Length` function, returns the length of the packed type.

- **to_byte_array** and **from_byte_array** - Equivalent to the Ada serialization/deserialization functions `Serialization.To_Byte_Array` and `Serialization.From_Byte_Array`; converts the MATLAB class to a serialized byte array whose bits correspond to the layout of the packed big-endian type (`T`) in Ada, or takes a byte array of this form and turns it into a MATLAB class.
- **create_from_byte_array** - Creates a new packed record object from a serialized byte array.
- **create_empty** - Creates a new packed record object where the default values of all fields is zero, or the lowest legal value where zero is not allowed.
- **to_string** - Equivalent to the Ada `Representation.Image` function; creates a human readable string version of the packed record.
- **to_tuple_string** - Equivalent to the Ada `Representation.To_Tuple_String` function; creates a log-friendly string of the packed record.

An example of how this class can be used is shown below in a file called `test.m`:

```

1 disp("create and disp example record:");
2 example1 = Example_Record(2, -1, 1, 0.5);
3 disp(example1);
4 disp("");
5 disp(example1.to_tuple_string());
6 disp("");
7 disp(example1.to_string());
8 disp("");
9 disp(example1.to_byte_string());
10 disp("");
11
12 ("serializing and deserializing example record:");
13 data = example1.to_byte_array();
14 example2 = Example_Record.create_from_byte_array(data);
15 assert(example1 == example2);
16 disp("done.");

```

The script can be run from the MATLAB command prompt:

```

>> test
create and disp example record:
Example_Record with properties:

                           Value_1: 2
                           Value_2: -1
                           Value_3: 1
                           Value_4: 0.5
                               size: 48
                           size_in_bytes: 6
min_serialized_length_in_bits: 48
max_serialized_length_in_bits: 48
    min_serialized_length: 6
    max_serialized_length: 6
        num_fields: 4
(Value_1 = 2, Value_2 = -1, Value_3 = 1, Value_4 = 0.5)
[17 01 3f 00 00 00]
Value_1 : Five_Bit_Integer = 2
Value_2 : Three_Bit_Signed_Integer = -1
Value_3 : My_Color = 1
Value_4 : Short_Float = 0.5
[17 01 3f 00 00 00]

```

done.

MATLAB classes can also be autogenerated for variable-length packed records and packed arrays, presented in the following sections.

Note that it is up to the user to make sure their MATLAB path is configured to include the appropriate directories for their program. Sometimes it is convenient to have all autocoded MATLAB copied to a single directory. Adamant provides some scripts to achieve this in *gnd/matlab/build_all_matlab_autocode.sh* and *gnd/matlab/copy_all_matlab_autocode.sh*.

5.1.10 Nested Packed Records

Packed records or packed arrays (discussed in Section 5.2) can also contain fields that are themselves packed records (or arrays). These types can be nested arbitrarily deep, but there are some limitations on nesting that will be discussed in this section.

See an example below, a packed record that includes two fields that are of packed record type:

```
1 ---  
2 description: This is an example packed record that demonstrates the nesting of  
3   ↳ other packed records within.  
4 fields:  
5   - name: Simple_Field  
6     description: A simple float, we must provide a format  
7     type: Short_Float  
8     # Required  
9     format: F32  
10    - name: Nested_Field_1  
11      description: This field is of type Example_Record, which is itself a packed  
12        ↳ record definition.  
13      type: Example_Record.T  
14        # ^ No format specification is necessary, since the  
15        # bit representation of a packed record is known  
16    - name: Nested_Field_2  
17      description: Another field of type Example Record.  
18      type: Example_Record.T
```

Note that the **format** field is not defined when the type of the field is a packed record, since the bit representation can be determined by the model for that packed record.

Nesting of packed types carries with it the following limitations/constraints:

1. Only T or T_Le types are allowed to be included as a packed record field type, since they represent “packed” types with known endianness and bit representation. U types are not allowed to be declared in a packed record or packed array since the bit representation is determined by the compiler. Similarly, arrays or records declared in a raw Ada package (ie. not modeled using Adamant) are also not allowed, since these, by default, are also unpacked types.
2. If a big-endian T type is used as a field type, then any other nested packed types must also be big-endian T types. Mixing endian is not currently supported. In this case, a T_Le type is not generated for the packed record, only a T and U type will be provided. Consequently, no Swap_Endianness function is provided either. The converse is true if little-endian T_Le types are nested in a packed record.

5.1.11 Variable Length Packed Records

The packed types that have been presented so far are all static in length. To be more specific, when these types are put onto an Adamant queue, they always take up the same amount of memory. For memory efficiency, Adamant also supports variable length packed records. In particular these records

have two main features that are common in many communication protocols:

- **Array of Values** - An array of values, often bytes, that are usually filled with multi-purpose data
- **Length** - A number, often in a protocol header, which indicates how many values in the array are valid and currently in use

One common protocol type used in space-based applications that follows this paradigm is a CCSDS packet, which contains an array of data bytes, whose length is specified by a specific value in the CCSDS header. Adamant uses variable length fields for many common types including commands, events, parameters, faults, and packets. Find these definitions in the `src/types` directory to explore good examples of variable length packed records.

Adamant variable length records offer an advantage when they are put onto queues. When enqueueing a variable length type, only the valid portion of the data is stored on the queue, not the maximum size of the type. So for instance, if the maximum size of a variable length packed record is 1000 bytes, but only 12 bytes are being specified as "used" by the length field, then only 12 bytes (plus some overhead) will be used on an Adamant queue to store the type. Note that when declaring a variable length type on the stack, the maximum size of the type will still be allocated. Care must still be taken in defining a reasonably sized maximum array length so as to not overflow the stack.

Adamant variable length packed records are optimized for speed, at the cost of flexibility. As a result they must adhere to the following constraints:

1. There may only be a single variable length field, ie. *array of values*, in a packed record.
2. The variable length field of the record must be the last field specified in the packed record.
3. All fields in a variable length record must be byte aligned.

Note that item 3 can be overcome by encapsulating any non-byte aligned fields in their own packed record (which as a whole must be byte aligned), and then including that packed record as the type for a field within variable length packed record. See the CCSDS Primary Header definition within Adamant for an example of this.

To define a variable length packed record we introduce a new specification `variable_length` and `variable_length_offset` to the packed record YAML definition. Below is a simple example:

```
1  ---
2  description: A simple variable sized record.
3  with:
4    - Interfaces
5  preamble: |
6    -- Define a byte with a restricted range.
7    subtype Limited_Bit is Interfaces.Unsigned_8 range 0 .. 250;
8    -- Define an array with index ranging from -15 to 4.
9    type Buffer_Type is array (Integer range -15 .. 4) of Limited_Bit;
10   fields:
11     - name: Length
12       type: Interfaces.Unsigned_8
13       format: U8
14     - name: Buffer
15       type: Buffer_Type
16       # Define the proper array format, 20 unsigned bytes
17       format: U8x20
18       # Define the default value as all zeros.
19       default: "[others => 0]"
```

```

20  # Define the length field that controls the "used" length of this field
21  variable_length: Length
22  # Optional - provide a signed integer that will be added to the length
23  # field to determine the actual value to use as the "used" length
24  # of the variable field. This will almost always be zero, but some
25  # protocols, notably CCSDS, require this value to be 1.
26  variable_length_offset: 0

```

The `variable_length` specification in the `Buffer` field indicates that 1) the `Buffer` field is of variable length and 2) that the `Length` field will be used to determine that length during runtime. The `variable_length_offset` field is a signed integer that will be added to the `Length` field during runtime to modify how the value is treated. For most sane protocols this value will always be 0, which is the default, and need not be specified. Some protocols, notably CCSDS, deviates from this and requires a value of 1 be specified for the `variable_length_offset`.

Note that the variable length field may be any type as long as it is an array. This can include arrays of non-byte aligned elements (so long as the whole array is byte aligned), arrays with elements larger than a single byte (such as a float), packed arrays (discussed in the previous sections), and even packed arrays whose element type is a packed record. The length field always specifies the number of used elements in the array in units of the array element type. It does not specify the number of valid bytes that are used unless, of course, the array element type is specifically a byte.

For the most part, variable length packed records have the same facilities available to them as their non-variable length counterparts including a serialization package, validation package, representation package, and an assertion package. There are subtle differences in the subprogram interface for some of the utilities, so see the specification of the autogenerated code to make sure that you are making the right calls, and providing proper error handling. In particular, serialization and deserialization of a variable length packed record can fail if the length field is larger than the array it is describing. In this case, any serialization and deserialization calls return a status to the caller that must be handled appropriately. Similarly, for assertions, there exists an assertion call to compare an entire variable length record against another up to the maximum size, and another which only compares the “used” variable length data, and ignores the data beyond.

The mechanism for determining a variable length record’s length is the autocoded `Serialized_Length` subprograms provided in the specification. These subprograms allow the user to determine the length of a type (in bytes). While the length of most packed records are static, meaning the function will always return the same value, variable length packed records will return different values based on the type’s content, ie. the `variable_length` field. These `Serialized_Length` subprograms are important because they are required by all generic packages within adamant that deal with variable length packed records, including Adamant’s variable queue package used for component queues, see Section 6.3.1.

5.1.12 C/C++ Packed Records

Adamant includes support for mixing C and C++ with Ada. See Section 9.1. For any packed record or packed array (see section 5.2), a `.C` child package can be generated which provides a version of the packed type suitable for passing to a binded C or C++ function. To create this package run:

```
> redo build/obj/Linux/example_record-c.o
```

This generates the source file `build/src/example_record-c.ads`. The autogenerated specification is shown below.

```

1 -----
2 -- Example_Record Record C/C++ Interface Spec
3 --
4 -- Generated from example_record.record.yaml on 2025-07-15 21:58.
5 -----

```

```

6
7 -- This is an example packed record.
8 package Example_Record.C is
9
10    -- Unpacked C/C++ compatible type:
11    type U_C is record
12        -- This is the first value.
13        Value_1 : aliased Five_Bit_Integer := 0;
14        -- This is the second value.
15        Value_2 : aliased Three_Bit_Signed_Integer;
16        Value_3 : aliased My_Color;
17        -- This is the fourth value.
18        Value_4 : aliased Short_Float;
19    end record
20    with Convention => C_Pass_By_Copy;
21
22    -- Access type for U_C.
23    type U_C_Access is access all U_C;
24
25    -- Functions for converting between the Ada and C version of the
26    -- unpacked and packed types:
27    function To_Ada (Src : in U_C) return U;
28    function To_C (Src : in U) return U_C;
29    function Pack (Src : in U_C) return T;
30    function Unpack (Src : in T) return U_C;
31    function Pack (Src : in U_C) return T_Le;
32    function Unpack (Src : in T_Le) return U_C;
33
34 end Example_Record.C;

```

The type is declared with the name `U_C` as in unpacked C/C++ native representation. Notice that all the field members include the `aliased` keyword to indicate that any field in the record might be dereferenced at any time, following C/C++ semantics. Appropriate pragmas are also included to specify that the value will be passed to C/C++ functions in the proper way.

Two functions are also provided: `To_C` to convert an unpacked Ada type, `U`, to the unpacked C/C++ type, `U_C`, and `To_Ada` to convert an unpacked C/C++ type to the unpacked Ada type. These functions can be used before or after calling into a binded C/C++ function. Doing so encapsulates the usage of the C and C++ type to only where it is needed. This allows the user to utilize all the features of Adamant packed types in the rest of the Ada-based code.

5.2 Packed Arrays

Packed arrays are similar to their packed record counterparts, and contain all of the features presented in the previous packed record sections. Specifying a packed arrays requires providing an element type and a length. The element type may be of any type including packed arrays, packed records, or types that don't end on a byte boundary. The resulting autocoded packed type will be of static length and of the type provided. The size of the array in memory will be packed such that it is the size of the format specified (in bits) multiplied by the length.

One useful utility of packed arrays might be to declare an array of 10-bit items that is packed tight with no padding. This is the idea we will use for our example. We want to declare an array of 10-bit integers with a valid range between 0 and 999 that is 8 elements long. Because the array is made up of 10-bit numbers, the packed version of the array should only take up 10-bytes in memory. Note, that some versions of the Ada compiler will require packed arrays to end on a 4-byte boundary. A compiler error will be thrown in this case, and you may have to modify your packed array YAML to comply.

To define a packed array, a developer should create a YAML model file of the form `name.array.yaml`,

where *name* is the desired name of the packed array. Below is the example packed array definition called *example_array*, stored in a file called *example_array.array.yaml*:

```

1  ---
2  # Optional - description of the packed record
3  description: This is an example packed array.
4  # Optional - preamble which provides "in-line" Ada code
5  preamble: subtype Short_Int is Natural range 1 .. 999;
6  # Required - the type of an array element
7  type: Short_Int
8  # Required (most of the time) - provide the number of bits
9  # that the element type will consume when packed. This field is
10 # required unless the type is itself another packed record
11 format: U10
12 # Required - the length of the packed array
13 length: 8

```

This YAML model file above contains comments explaining whether or not each field is optional or required. The main portion of the model is the packed array *type*, *length*, and *format* specification. The type of the field can be any Ada type, including arrays, records, enumerations, packed records, or other packed arrays. The *format* field is where a developer can describe out how the field will be represented in memory in its packed form, and is identical to the format specification for packed records. The *format* specification is not necessary if the array type is a packed record or packed array since the length of the type in memory is known from that type's model, see Section 5.1.10. See the previous sections for more details on *format*. The *length* field specifies how many elements the packed array contains. Unconstrained packed arrays are not currently supported in Adamant.

5.2.1 Packed Array Ada Specification

The *example_array.array.yaml* file produces a single Ada specification file which declares a package containing the packed and unpacked representations of the array. The file can be generated and compiled by running the following command from the same directory as *example_array.array.yaml*:

```
> cd to/dir/with/example_array
> redo build/obj/Linux/example_array.o
```

This will generate the source files *build/src/example_array.ads* and *build/src/example_array.adb*. The autocoded specification is shown below.

```

1 -----
2 -- Example_Array Array Spec
3 --
4 -- Generated from example_array.array.yaml on 2025-07-15 21:49.
5 --
6
7 -- Standard Includes:
8 with System;
9 with Serializer;
10 with Basic_Types;
11
12 -- This is an example packed array.
13 package Example_Array is
14
15   -- Preamble code:
16   subtype Short_Int is Natural range 1 .. 999;
17
18   -- Packed type size (in bits):
19   Length : constant Natural := 8;
20   Element_Size : constant Positive := 10;
21   Size : constant Positive := Element_Size * Length;

```

```

22
23    -- Packed type size rounded up to nearest byte.
24    Element_Size_In_Bytes : constant Positive := (Element_Size - 1) /
25        Basic_Types.Byte'Object_Size + 1;
26    Size_In_Bytes : constant Positive := (Size - 1) / 8 + 1;
27
28    -- Array index type:
29    subtype Unconstrained_Index_Type is Natural;
30    subtype Constrained_Index_Type is Unconstrained_Index_Type range 0 .. Length
31        - 1;
32
33    -- Unconstrained base type:
34    type Unconstrained is array (Unconstrained_Index_Type range <>) of Short_Int;
35
36    -- Unpacked array type:
37    subtype U is Unconstrained (Constrained_Index_Type);
38
39    -- Access type for U
40    type U_Access is access all U;
41
42    -- Packed type definition.
43    type T is new U
44        with Component_Size => Element_Size,
45            Scalar_Storage_Order => System.High_Order_First,
46            Object_Size => Size,
47            Volatile => False,
48            Volatile_Components => False;
49
50
51    -- Access type for T
52    type T_Access is access all T;
53
54    -- Packed type definition with little endian definition.
55    type T_Le is new U
56        with Component_Size => Element_Size,
57            Scalar_Storage_Order => System.Low_Order_First,
58            Object_Size => Size,
59            Volatile => False,
60            Volatile_Components => False;
61
62    -- Access type for T_Le
63    type T_Le_Access is access all T_Le;
64
65    -- Volatile packed type definition:
66    -- Note: This type is volatile. You should use this type to specify that the
67    -- variable in question may suddenly change in value. For example, this may
68    -- occur due to a device writing to a shared buffer. When this pragma is
69    -- used,
70    -- the compiler must suppress any optimizations that would interfere with the
71    -- correct reading of the volatile variables. For example, two successive
72    -- readings of the same variable cannot be optimized to just one or
73    -- reordered.
74    -- Important: You should use the Register type for accessing hardware
75    -- registers.
76    type Volatile_T is new T
77        with Component_Size => Element_Size,
78            Scalar_Storage_Order => System.High_Order_First,
79            Object_Size => Size,
80            Volatile => True,
81            Volatile_Components => True;
82
83
84    -- Access type for Volatile_T

```

```

78 type Volatile_T_Access is access all Volatile_T;
79
80 -- Volatile little endian packed type definition:
81 -- Note: This type is volatile. You should use this type to specify that the
82 -- variable in question may suddenly change in value. For example, this may
83 -- occur due to a device writing to a shared buffer. When this pragma is
84 -- used,
85 -- the compiler must suppress any optimizations that would interfere with the
86 -- correct reading of the volatile variables. For example, two successive
87 -- readings of the same variable cannot be optimized to just one or
88 -- reordered.
89 -- Important: You should use the Register type for accessing hardware
90 -- registers.
91 type Volatile_T_Le is new T_Le
92   with Component_Size => Element_Size,
93        Scalar_Storage_Order => System.Low_Order_First,
94        Object_Size => Size,
95        Volatile => True,
96        Volatile_Components => True;
97
98 -- Access type for Volatile_T_Le
99 type Volatile_T_Le_Access is access all Volatile_T_Le;
100
101 --
102 -- Atomic type definitions. These are only supported for array types that
103 -- have components that are exactly 32 bits in size and are not themselves
104 -- packed types.
105 --
106 -- Not supported. This type has array components that are 10 bits in size.
107
108 -- Type conversion functions between packed an unpacked representations:
109 function Pack (Src : in U) return T with Inline => True;
110 function Unpack (Src : in T) return U with Inline => True;
111 function Pack (Src : in U) return T_Le with Inline => True;
112 function Unpack (Src : in T_Le) return U with Inline => True;
113
114 -- Endianness conversion functions
115 function Swap_Endianness (Src : in T) return T_Le with Inline => True;
116 function Swap_Endianness (Src : in T_Le) return T with Inline => True;
117
118 -- Serializing functions for entire packed array:
119 package Serialization is new Serializer (T);
120 package Serialization_Le is new Serializer (T_Le);
121
122 -- Packed type definition for array element:
123 subtype Element_Packed is Short_Int
124   with Object_Size => Element_Size_In_Bytes * 8,
125        Value_Size => Element_Size_In_Bytes * 8;
126
127 -- Serializing functions for an element of the array:
128 package Element_Serialization is new Serializer (Element_Packed);
129
130 -----
131 -- Named subtypes for array element for convenience:
132 -----
133 subtype Element_Type is Short_Int;
134 end Example_Array;

```

An array is produced with the specifications we described. The `U` type will respect the Ada *type*

given in the YAML file, and any range associated with it. The representation of this type in memory is completely up to the compiler. The unpacked type should be used whenever the type is used in heavy calculations, since the compiler will be able to optimize the program for speed instead of spending time worrying about storage space and endianness. If the representation or size of the type is of concern, like when sending the data through a connector or to a different processing unit, the T type should be used. This is a specialization of the U type which specifies exactly how the array should be laid out in memory. The size of each field is calculated, and a big endian representation is requested.

5.2.2 Packed Array Type Conversion

Packed array type conversion is similar to packed record type conversion. See Section 5.1.2 for details.

5.2.3 Packed Array Serialization

Packed array serialization is similar to packed record serialization. See Section 5.1.3 for details.

5.2.4 Packed Array Validation

Packed array validation is similar to packed record validation. See Section 5.1.4 for details.

5.2.5 Packed Array String Representation

During testing, it is often nice to print the value of a complex datatype in a human readable form. Adamant has the ability to generate human readable string representations for packed arrays. The string representation code for the example_array is included in two files, a specification *example_array-representation.ads* and *example_array-representation.adb* which can be generated and compiled using the following commands:

```
> redo build/obj/Linux/example_array-representation.o
```

This generates the source files *build/src/example_array-representation.ads* and *build/src/example_array-representation.adb*. The autocoded specification is shown below.

```

1 -----
2 -- Example_Array Array Representation Spec
3 --
4 -- Generated from example_array.array.yaml on 2025-07-15 21:49.
5 -----
6
7 -- Standard includes:
8 with String_Util;
9
10 -- String representation package for Example_Array
11 package Example_Array.Representation is
12
13 -----
14 -- Common to string functions:
15 -----
16 -- Return string showing bytes in array:
17 function To_Byte_String is new String_Util.To_Byte_String (U);
18 function To_Byte_String is new String_Util.To_Byte_String (T);
19 function To_Byte_String is new String_Util.To_Byte_String (T_Le);
20
21 -- Display array as string:
22 function Image (R : in U) return String;
23 function Image (R : in T) return String;
24 function Image (R : in T_Le) return String;
25
```

```

26 -----
27 -- Less commonly used to string functions:
28 -----
29 -- Return string representation of array components in form (element 1,
30 -- element 2, etc.)
31 function To_Tuple_String (R : in U) return String;
32 function To_Tuple_String (R : in T) return String;
33 function To_Tuple_String (R : in T_Le) return String;
34
35 -- Return string representation of array elements and bytes
36 function Image_With_Prefix (R : in U; Prefix : in String) return String;
37 function Image_With_Prefix (R : in T; Prefix : in String) return String;
38 function Image_With_Prefix (R : in T_Le; Prefix : in String) return String;
39
40 -----
41 -- To string functions for array element type:
42 -----
43 function Element_To_Byte_String is new String_Util.To_Byte_String
44   (Short_Int);
45 function Element_Image (R : in Short_Int) return String renames
46   Short_Int'Image;
47 function Element_To_Tuple_String (R : in Short_Int) return String renames
48   Short_Int'Image;
49 function Element_Image_With_Prefix (R : in Short_Int) return String renames
50   Short_Int'Image;
51
52 end Example_Array.Representation;

```

There are many subprograms provided, but the most commonly used are `To_Byte_String`, `Image`, and `To_Tuple_String`. The first `To_Byte_String` function returns a string that shows the bytes that make up the packed array in hex. The second `Image` functions returns the most human readable representation of the packed array, printing each element name following by its value on a separate line. The `To_Tuple_String` is useful for logging, as the entire array is always be printed, comma separated, on a single line.

Below is an example of using the package.

```

1  with Ada.Text_IO; use Ada.Text_IO;
2  with Example_Array.Representation;
3
4  procedure Main is
5    -- A packed version of the type.
6    Packed_Type_1 : constant Example_Array.T := [11, 22, 33, 44, 55, 66, 77, 88];
7  begin
8    -- Print the packed array to the terminal in human readable form:
9    Put_Line ("Human readable:");
10   Put_Line (Example_Array.Representation.Image (Packed_Type_1));
11   Put_Line ("");
12
13   -- Print the packed array to the terminal in log friendly form:
14   Put_Line ("Log friendly:");
15   Put_Line (Example_Array.Representation.To_Tuple_String (Packed_Type_1));
16   Put_Line ("");
17 end Main;

```

which, when run, produces the output:

```

Human readable:
[02 C1 60 84 2C 0D C4 21 34 58 00 00 00 00 00 00]
Example_Array : array Short_Int (1 .. 8) => [

```

```

0 => 11,
1 => 22,
2 => 33,
3 => 44,
4 => 55,
5 => 66,
6 => 77,
7 => 88
]

Log friendly:
[11, 22, 33, 44, 55, 66, 77, 88]

```

Note that Adamant currently struggles to support pretty print strings for certain Ada types including `System.Address` and all access types. The inclusion of these in packed arrays is rare in practice so this does not pose a critical issue. If the string representation autocode for an element type is not to your liking, or does not compile you have two options: 1) Don't use the autocoded package and instead hand-code your own or 2) Use the `Byte_Image` field modifier to specify that you want the representation of the type to simply be an array of hex bytes. Since any type can be represented by its underlying machine byte values, option 2 will always compile, but may not be as human readable. See Section 5.1.5 for an example of using this modifier with packed records.

5.2.6 Packed Array Unit Test Assertions

Packed array unit test assertions are similar to packed record unit test assertions. See Section 5.1.6 for details.

5.2.7 Packed Array Documentation

Adamant provides automatic generation of documentation for any packed array definition. Currently two versions of documentation can be created: HTML, which is useful for presentation in meetings or for quick reference, and PDF (via L^AT_EX), which is useful for formal documentation.

To build HTML documentation run:

```
> redo build/html/example_array.html
```

which when opened with your favorite web browser looks something like this:

Example_Array Packed Array : 80 bits

Description: This is an example packed array.

Preamble:

```
subtype Short_Int is Natural range 1 .. 999;
```

Type	Range	Element Size (Bits)	Length	Total Size (Bits)
Short_Int	1 to 999	10	8	80

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/example_array.array.yaml on 2020-04-20 22:04.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

To build the PDF documentation run:

```
> redo build/pdf/example_array.pdf
```

Which produces a PDF output that looks like the following.

This is an example packed array. *Preamble (inline Ada definitions):*

```
1 subtype Short_Int is Natural range 1 .. 999;
```

Table 4: Example_Array Packed Array : 80 bits

Type	Range	Element Size (Bits)	Length	Total Size (Bits)
Short_Int	1 to 999	10	8	80

5.2.8 Packed Array Python Class

Packed array python classes are similar to packed record python classes. See Section 5.1.8 for details.

5.2.9 Packed Array MATLAB Class

Packed array MATLAB classes are similar to packed record MATLAB classes. See Section 5.1.9 for details.

5.2.10 Nested Packed Arrays

Packed arrays with elements of type packed arrays or packed records are allowed. This is specified similarly to the way it is done for packed records. See Section 5.1.10 for details.

5.2.11 C/C++ Packed Arrays

Packed arrays can be converted to a C/C++ type in a similar way to packed records. See Section 5.1.12 for details.

5.3 Enumerations

As seen in the previous sections, Ada enumeration types can be used in both packed records and packed arrays. Adamant also provides a mechanism to model enumerations in YAML, which can be used to autocode enumerations. These autocoded enumerations themselves are not packed types, but can easily be included in packed records as field types or as the element type within a packed array. The benefit to modeling an enumeration is that Adamant can provide autocoded assertion packages, representation packages, and documentation. Enumeration modeling should be used for enumerations that will be widely used, such as a protocol header flag, or those that are important and require extensive documentation, such as a system mode state.

In adamant, enumerations are modeled in suites, so many enumeration types can be described in the same YAML file. To create an enumeration suite called *example_enums*, we can write a file called *example_enums.enums.yaml*:

```
1 ---
2 # Optional - Description of the enumeration suite
3 description: This is a simple set of useless enumerations
4 enums:
```

```

5   - name: First_Enum
6     description: This is the first enumeration type.
7     literals:
8       # Required - The name of the first literal
9       - name: Yellow
10      # Optional - The integral value of the first literal
11      value: 10
12      # Optional - A description of what the first state means.
13      description: The yellow state.
14     - name: Green
15       description: The green state.
16     - name: Blue
17       description: The blue state.
18     - name: Purple
19       value: 0
20       description: The purple state.
21   - name: Second_Enum
22     description: This is the second enumeration type.
23     literals:
24       - name: Car
25       - name: Boat
26       - name: House
27       - name: Spaceship

```

This YAML model file above contains comments explaining whether or not each field is optional or required. Two enumerations are described in this file: `First_Enum` and `Second_Enum` both of which have different states, also called literals. For each enumeration and each literal a description can be provided. An integral value can be assigned to the literal if necessary. If unspecified, Adamant assigns the lowest possible remaining unique positive integer (or zero) to each literal.

5.3.1 Enumeration Specification

The `example_enums.enums.yaml` file produces a single Ada specification file which declares a package containing the enumeration definitions. The file can be generated and compiled by running the following command from the same directory as `example_enums.enums.yaml`:

```
> cd to/dir/with/example_enums
> redo build/obj/Linux/example_enums.0
```

Below is the specification file produced, `build/src/example_enums.ads`.

```

1 -----
2 -- Example_Enums Enums Spec
3 --
4 -- Generated from example_enums.enums.yaml on 2025-07-15 21:47.
5 -----
6
7 -- This is a simple set of useless enumerations
8 package Example.Enums is
9
10   --
11   -- Enumeration types:
12   --
13
14   -- First_Enum Definition:
15   -- This is the first enumeration type.
16   package First_Enum is
17     -- Enumeration type definition:
18     type E is (
19       Purple, -- The purple state.

```

```

20             Green, -- The green state.
21             Blue, -- The blue state.
22             Yellow -- The yellow state.
23         );
24         -- Enumeration type values:
25         for E use (
26             Purple => 0,
27             Green => 1,
28             Blue => 2,
29             Yellow => 10
30         );
31     end First_Enum;
32
33     -- Second_Enum Definition:
34     -- This is the second enumeration type.
35     package Second_Enum is
36         -- Enumeration type definition:
37         type E is (
38             Car, --
39             Boat, --
40             House, --
41             Spaceship ---
42         );
43         -- Enumeration type values:
44         for E use (
45             Car => 0,
46             Boat => 1,
47             House => 2,
48             Spaceship => 3
49         );
50     end Second_Enum;
51
52 end Example.Enums;

```

A package is produced with two internal packages named `First_Enum` and `Second_Enum` which both define an enumerations named `E` that describe the literals expressed in the YAML. Note that values are assigned to each literal explicitly. If literal values are left unspecified in the YAML file, Adamant assigns the lowest possible remaining unique positive integer (or zero) to each literal, as can be seen in the `First_Enum` package.

5.3.2 Enumeration String Representation

During testing, it is often nice to print the value of an enumeration in a human readable form. Adamant has the ability to generate human readable string representations for enumerations. The string representation code for the `example_enums` is included in two files, a specification `build/src/example_enums-representation.ads` and `build/src/example_enums-representation.adb` which can be generated and compiled using the following commands:

```
> redo build/obj/Linux/example_enums-representation.o
```

This generates the source files `build/src/example_enums-representation.ads` and `build/src/example_enums-representation.adb`. The autocoded specification is shown below.

```

1 -----
2 -- Example.Enums Enums Representation Spec
3 --
4 -- Generated from example_enums.enums.yaml on 2025-07-15 21:47.
5 -----
6
7 -- String representation package for Example.Enums

```

```

8 package Example.Enums.Representation is
9
10    -- First_Enum representation functions:
11    function First_Enum_Image (Enum : in Example.Enums.First_Enum.E) return
12        String;
13    function First_Enum_To_Tuple_String (Enum : in Example.Enums.First_Enum.E)
14        return String renames Example.Enums.First_Enum.E'Image;
15    function First_Enum_Image_With_Prefix (Enum : in Example.Enums.First_Enum.E;
16        Prefix : String := "") return String;
17
18    -- Second_Enum representation functions:
19    function Second_Enum_Image (Enum : in Example.Enums.Second_Enum.E) return
20        String;
21    function Second_Enum_To_Tuple_String (Enum : in Example.Enums.Second_Enum.E)
22        return String renames Example.Enums.Second_Enum.E'Image;
23    function Second_Enum_Image_With_Prefix (Enum : in
24        Example.Enums.Second_Enum.E; Prefix : String := "") return String;
25
26 end Example.Enums.Representation;

```

There are many subprograms provided, but the most commonly used are the *_Image and the *_To_Tuple_String. Both are pretty similar for enumerations, and the differences can be seen in the example program below.

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Example.Enums.Representation;
3
4 procedure Main is
5     use Example.Enums;
6     use Example.Enums.Representation;
7     use Example.Enums.First_Enum;
8     -- A enum instantiation.
9     Enum_1 : constant First_Enum.E := Blue;
10 begin
11     -- Print the enum to the terminal in human readable form:
12     Put_Line ("Human readable:");
13     Put_Line (First_Enum_Image (Enum_1));
14     Put_Line ("");
15
16     -- Print the enum to the terminal in log friendly form:
17     Put_Line ("Log friendly:");
18     Put_Line (First_Enum_To_Tuple_String (Enum_1));
19     Put_Line ("");
20 end Main;

```

which, when run, produces the output:

```
Human readable:
BLUE (2)
```

```
Log friendly:
BLUE
```

5.3.3 Enumeration Unit Test Assertions

Adamant also provides an assertion package for each enumeration suite. The assertion package is useful during unit testing, as it allows you to compare two enumerations. If the enumerations do not

meet the comparison criteria, then a human readable error message is printed. To build the assertion package run.

```
> redo build/obj/Linux/example_enums-assertion.o
```

This generates the source files *build/src/example_enums-assertion.ads* and *build/src/example_enums-assertion.adb*. The autocoded specification is shown below.

```

1-----  

2-- Example_Enums Enums Assertion Spec  

3--  

4-- Generated from example_enums.enums.yaml on 2025-07-15 21:49.  

5-----  

6  

7-- Standard includes:  

8with Smart Assert;  

9with Example_Enums.Representation;  

10  

11package Example_Enums.Assertion is  

12  

13    -- Basic assertion packages for enumeration First_Enum:  

14    package First_Enum Assert is new Smart Assert.Basic  

15        (Example_Enums.First_Enum.E,  

16         Example_Enums.Representation.First_Enum_Image);  

17  

18    -- Basic assertion packages for enumeration Second_Enum:  

19    package Second_Enum Assert is new Smart Assert.Basic  

20        (Example_Enums.Second_Enum.E,  

21         Example_Enums.Representation.Second_Enum_Image);  

22  

23end Example_Enums.Assertion;
```

The package uses the Adamant *Smart Assert* package and can be used as follows:

```

1with Example_Enums.Assertion;  

2  

3procedure Main is  

4    use Example_Enums;  

5    use Example_Enums.First_Enum;  

6    use Example_Enums.Assertion;  

7    -- A enum instantiation.  

8    Enum_1 : constant First_Enum.E := Green;  

9begin  

10    -- See if the enums are not equal (should pass):  

11    First_Enum Assert.Neq (Enum_1, Blue);  

12  

13    -- See if the enums are equal (should not pass):  

14    First_Enum Assert.Eq (Enum_1, Purple);  

15end Main;
```

which, when run, produces the output:

```
Assertion:  
GREEN (1)  
=  
PURPLE (0)  
failed.  
  
raised ADA ASSERTIONS ASSERTION_ERROR : at main.adb:14
```

5.3.4 Enumeration Documentation

Adamant provides automatic generation of documentation for any enumeration definition. Currently two versions of documentation can be created: HTML, which is useful for presentation in meetings or for quick reference, and PDF (via L^AT_EX), which is useful for formal documentation.

To build HTML documentation run:

```
> redo build/html/exampleEnums.html
```

which when opened with your favorite web browser looks something like this:

Example.Enums Enumerations

Description: This is a simple set of useless enumerations

First_Enum Literals

Description: This is the first enumeration type.

Name	Value	Description
Purple	0	The purple state.
Green	1	The green state.
Blue	2	The blue state.
Yellow	10	The yellow state.

Second_Enum Literals

Description: This is the second enumeration type.

Name	Value	Description
Car	0	
Boat	1	
House	2	
Spaceship	3	

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/exampleEnums.enums.yaml on 2020-04-20 22:04.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

To build the PDF documentation run:

```
> redo build/pdf/exampleEnums.pdf
```

Which produces a PDF output that looks like the following.

This is a simple set of useless enumerations **First_Enum**: This is the first enumeration type.

Table 5: First_Enum Literals:

Name	Value	Description
Purple	0	The purple state.
Green	1	The green state.
Blue	2	The blue state.
Yellow	10	The yellow state.

Second_Enum: This is the second enumeration type.

Table 6: Second _Enum Literals:

Name	Value	Description
Car	0	
Boat	1	
House	2	
Spaceship	3	

5.3.5 Using Enumerations In Packed Types

An Adamant-modeled enumeration can be used in a packed record or packed array just like a standard Ada-defined enumeration. Below is an example. We modified the *example_record* to contain two new fields, one that holds the enumeration, and another that holds some padding bits to make sure the packed record is byte aligned. The new file is called *enum_record.record.yaml* and is shown here:

```

1  ---
2  # Optional - description of the packed record
3  description: This is an example packed record.
4  # Optional - preamble which provides "in-line" Ada code
5  preamble: |
6      type Four_Bit_Integer is mod 2**4;
7      type Five_Bit_Integer is mod 2**5;
8      subtype Three_Bit_Signed_Integer is Integer range -3 .. 2;
9      type My_Color is (Red, Green, Blue, Purple, Pink, Chartreuse);
10     # Optional - Include any dependencies we might need
11     # * In this case we don't have any... but here is what it might
12     # look like:
13     # with:
14     #   - Basic_Types
15     # List the record fields
16 fields:
17     # Required - the name of the field
18     - name: Value_1
19         # Optional - description of the field
20         description: This is the first value.
21         # Required - the type of the field
22         type: Five_Bit_Integer
23         # Required (most of the time) - provide the number of bits
24         # that this type will consume when packed. This field is
25         # required unless the type is itself another packed record
26         format: U5
27         # Optional - provide a default value
28         default: "0"
29     - name: Value_2
30         description: This is the second value.
31         type: Three_Bit_Signed_Integer

```

```

32     format: U3
33 - name: Value_3
34   type: My_Color
35   format: E8
36 - name: Value_4
37   description: This is the fourth value.
38   type: Short_Float
39   format: F32
40 - name: Value_5
41   description: This is the fifth value it and contains an Adamant-modeled
42     → enumeration type.
42   type: Example.Enums.Second_Enum.E
43   default: Example.Enums.Second_Enum.Spaceship
44   format: E4
45 - name: Reserved
46   description: Reserved bits to make sure we end on a byte boundary.
47   type: Four_Bit_Integer
48   format: U4
49

```

Note how the format used for the Adamant-modeled enumeration is E4. The E must be used when the type is an enumeration, or a modeling error will be thrown.

5.3.6 Enumeration Python Class

For each enumeration in an enumeration model, Adamant provides a python enumeration class. For space-based applications, these python classes are meant as “ground” tools, or tools which assist with some task related to the deployed embedded system such as testing, parsing data, sending commands, etc. See Section 8 for more details. To build the python class for an enumeration model run:

```
> redo build/py/example_enum.py
```

The resulting autocode is shown below:

```

1 ######
2 # Example.Enums Enums
3 #
4 # Generated from example_enums.enums.yaml on 2025-07-15 21:58.
5 #####
6
7 from base_classes.enum_base import EnumBase
8
9 # This is a simple set of useless enumerations
10
11
12 # First_Enum Definition:
13 # This is the first enumeration type.
14 class First_Enum(EnumBase):
15     # The purple state.
16     Purple = 0
17     # The green state.
18     Green = 1
19     # The blue state.
20     Blue = 2
21     # The yellow state.
22     Yellow = 10
23
24
25 # Second_Enum Definition:
26 # This is the second enumeration type.

```

```

27 class Second_Enum(EnumBase):
28     Car = 0
29     Boat = 1
30     House = 2
31     Spaceship = 3

```

The resulting enumeration classes inherit from the Adamant base class `EnumBase`, which in turn inherits from the python `enum.Enum` class. The python class contains the numeric definitions for each of the enumeration literals. It also includes the following helpful methods:

- **`to_string`** - Equivalent to the `Ada.Representation.Image` function; creates a human readable version of the enumeration.
- **`to_tuple_string`** - Equivalent to the `Ada.Representation.To_Tuple_String` function; creates a log-friendly string of the enumeration.

An example of how these classes can be used is shown below in a file called `test.py`:

```

1  #!/usr/bin/env python3
2
3  # Build our dependencies using the build system.
4  # This is necessary because some of the dependencies we
5  # have are autogenerated.
6  from util import pydep
7
8  pydep.build_py_deps()
9
10 from test_enums import First_Enum, Second_Enum
11
12 # Main:
13 if __name__ == "__main__":
14     print("create enums:")
15     fe = First_Enum(1)
16     print(str(fe))
17     print(str(fe.to_tuple_string()))
18     print(str(fe.to_string()))
19     try:
20         fe = First_Enum(-1)
21         assert False, "should not get here."
22     except ValueError:
23         pass
24     se = Second_Enum(10)
25     print()
26     print(str(se))
27     print(str(se.to_tuple_string()))
28     print(str(se.to_string()))
29     try:
30         se = Second_Enum(12)
31         assert False, "should not get here."
32     except ValueError:
33         pass
34     print("passed.")
35     print()

```

The test code tests that initializing an enumeration class with a value outside of the enumeration range fails with a python `ValueError` exception.

Note the call at the top of the script to `pydep.build_py_deps()`. This Adamant python module does two useful things: 1) it searches the python program for any autocoded dependencies and uses `redo` to build them and 2) adds any autocoded dependencies automatically to the `PYTHONPATH`

so they can be imported without error. It is important to run this function as the first line of any Adamant python script that will use autocoded python classes.

The script can be run with the command:

```
> python test.py
```

and produces the following output:

```
create enums:  
First_Enum.Green  
Green  
Green (1)  
  
Second_Enum.Yellow  
Yellow  
Yellow (10)  
passed.
```

Python classes can also be autogenerated for packed records and packed arrays, see Section 5.1.8.

5.3.7 Enumeration MATLAB Class

For each enumeration in an enumeration model, Adamant provides a MATLAB enumeration class. For space-based applications, these MATLAB classes are meant as “ground” tools, or tools which assist with some task related to the deployed embedded system such as testing, parsing data, sending commands, etc. See Section 8 for more details. To build the MATLAB class for an enumeration model run:

```
> redo build/m/Example.Enums
```

This will build a .m file for each enumeration defined in *example_enums.enums.yaml*, since MATLAB requires that each enum class definition be contained within its own .m file. The resulting autogenerated code for *build/m/Example.Enums/First_Enum.m* and *build/m/Example.Enums/Second_Enum.m* shown below:

First_Enum.m

```
1 %%%%%%%%
2 %% Example.Enums Enums
3 %%
4 %% Generated from example_enums.enums.yaml on 2025-07-15 21:58.
5 %%%%%%%%
6 classdef First_Enum < uint64
7 % First_Enum Definition:
8 % This is the first enumeration type.
9 enumeration
10 Purple (0) % The purple state.
11 Green (1) % The green state.
12 Blue (2) % The blue state.
13 Yellow (10) % The yellow state.
14 end
15 end
```

Second_Enum.m

```
1 %%%%%%
2 %% Example_Enums Enums
3 %%
4 %% Generated from example_enums.enums.yaml on 2025-07-15 21:58.
5 %%%%%%
6 classdef Second_Enum < uint64
7     % Second_Enum Definition:
8     % This is the second enumeration type.
9     enumeration
10        Car (0)
11        Boat (1)
12        House (2)
13        Spaceship (3)
14    end
15 end
```

The MATLAB class contain the numeric definitions for each of the enumeration literals.

An example of how these classes can be used is shown below in a file called *test.m*:

```
1 disp("create enums:");
2 fe = First_Enum(1);
3 disp(fe);
4 fe = First_Enum(10);
5 disp(fe);
6 disp("");
7 se = Second_Enum(0);
8 disp(se);
9 se = Second_Enum(3);
10 disp(se);
11 disp("passed.");
12 disp("")
```

The script can be run with the command:

```
>> test
create enums:
    Green
    Yellow
    Car
    Spaceship
passed.
```

MATLAB classes can also be autogenerated for packed records and packed arrays, see Section 5.1.9.

5.4 Packed Registers

Adamant packed records and arrays allow a developer to specify the exact bit representation of a type within the Ada language. This feature makes it ideal for describing hardware register definitions and memory mapped IO. However, using regular packed record types .T (for big endian machines) and .T_Le (for little endian machines) is often not enough to ensure correct interaction with the hardware. On many hardware platforms, registers must be read in their entirety (ie. all 32-bits) with a single CPU instruction without interruptions from other tasks in the system, and without reads and writes being removed by compiler optimizations. Adamant provides a method for circumventing these issues using special Volatile, Atomic, and Register packed records and packed record sets, which are described in the subsequent sections.

5.4.1 Volatile, Atomic, and Volatile Full Access in Ada

The Ada language provides a few facilities for specifying how an object or type should be read or written to memory. These come in the form of representation pragmas (or aspects) that specify the properties of the type of object. The important facilities used within Adamant are `Volatile`, `Atomic`, and `Volatile_Full_Access`. Descriptions of each is provided below:

- **`Volatile`** - `Volatile` is a representation pragma that can be used with types and variables to specify that the variable in question may suddenly change in value. For example, this may occur due to a device writing to a shared buffer. When this pragma is used, the compiler must suppress any optimizations that would interfere with the correct reading of the volatile variables. For example, two successive readings of the same variable cannot be optimized to just one or reordered.
- **`Atomic`** - `Atomic` is a representation pragma that can be used with types and variables to specify that the code generated must read and write the type or variable from memory atomically, i.e. as a single/non-interruptible operation. It implies pragma `Volatile`, the difference is that pragma `Atomic` is stronger: the compilation must fail if the variable cannot be updated atomically. In some cases it can be used for tasking, but for communication of Ada tasks it is safer to use protected types or other synchronization primitives. Note: It is almost ALWAYS wrong to use `Atomic` variables for tasking. If you choose to do so, proceed with caution.
- **`Volatile_Full_Access`** - This is similar in effect to pragma `Volatile`, except that any reference to the object is guaranteed to be done only with instructions that read or write ALL the bits of the object. Furthermore, if the object is of a composite type, then any reference to a subcomponent of the object is guaranteed to read and/or write ALL the bits of the object. The intention is that this be suitable for use with memory-mapped I/O devices and register on some machines. Note that there are two important respects in which this is different from pragma `Atomic`. First a reference to a `Volatile_Full_Access` object is not a sequential action in the RM 9.10 sense and, therefore, does not create a synchronization point. Second, in the case of pragma `Atomic`, there is no guarantee that all the bits will be accessed if the reference is not to the whole object; the compiler is allowed (and generally will) access only part of the object in this case. In Adamant, `Volatile_Full_Access` is used for register type definitions.

In general, `Volatile_Full_Access` should be used for all hardware interaction, particularly if you need to guarantee that an entire register is read from/written to (instead of just a portion). `Volatile` should be used when you want to access a memory region that could be written to / read from by some external source, such as an FPGA shared buffer. `Atomic` should be used rarely, but is sometimes useful if a variable needs to be shared between interrupt handlers, where using a protected object may impose too much of a performance penalty. You should almost always prefer the use of protected objects over `Atomic` variables.

These pragmas can be applied to types, ie.

```
-- Volatile integer:
type V_Int is new Integer
  with Volatile => True;
-- Array with atomic components:
type A_Array is new array(Natural range 1 .. 10) of Positive
  with Volatile_Components => True;
```

and also to objects, ie.

```
-- Integer register:
a : Integer
  with Volatile_Full_Access => True;
for a'Address use Some_Hw_Addr;
```

```
-- Volatile integer:
b : Integer := 0
  with Volatile => True;
```

The following section will show how to use Adamant packed records and arrays with these pragmas already predefined.

5.4.2 Packed Register Type

In Adamant, when you create a packed record or array a volatile version of that type is always created called `.Volatile_T` (big endian) or `.Volatile_T_Le` (little endian). You may have noticed these definitions in section 5.1.1. These can be used to specify a volatile version of the record, which when accessed in a volatile manner.

Atomic packed records, `.Atomic_T` or `.Atomic_T_Le`, and `Volatile_Full_Access` packed records, `.Register_T` or `.Register_T_Le`, are only defined when the size of the packed record is exactly 8, 16, or 32 bits in length. The reason for this is that Ada can only ensure that at type is atomic or `Volatile_Full_Access` if the type is of one of those sizes. Since registers used on embedded targets for Adamant are almost always 32 bits in length, a packed record register definition is always created with a `.Register_T` and `.Register_T_Le` type that should be used to access any hardware defined by that type.

A typical 32-bit hardware register definition is provided below:

```
1  ---
2  description: This is an example packed register.
3  preamble: |
4    type Enable_Disable_Type is (Disable, Enable);
5    for Enable_Disable_Type use (Disable => 0, Enable => 1);
6    type Fourteen_Bit_Uint is mod 2**14;
7  # List the record fields. The whole register must be 32-bits in size.
8  fields:
9    - name: Hw_1_Enabled
10      description: Single bit to represent if some fictional hardware 1 is enabled
11        ↳ or disabled.
12      type: Enable_Disable_Type
13      format: E1
14    - name: Hw_2_Enabled
15      description: Single bit to represent if some fictional hardware 2 is enabled
16        ↳ or disabled.
17      type: Enable_Disable_Type
18      format: E1
19    - name: Threshold
20      description: This is a 14-bit unsigned integer.
21      type: Fourteen_Bit_Uint
22      format: U14
23    - name: Unused
24      description: The second half of the register is not used, but must be
                    ↳ defined since registers are 32-bits in size.
          type: Interfaces.Unsigned_16
          format: U16
```

Since the size is 32 bits in length, the special `Atomic` and `Register` types are created with the expected Ada defined representation pragmas (aspects).

```
1  -----
2  -- Example_Register Record Spec
3  --
4  -- Generated from example_register.record.yaml on 2025-07-15 21:58.
5  -----
```

```

6
7  -- Standard Includes:
8  with Basic_Types;
9  with System;
10 with Serializer_Types; use Serializer_Types;
11 with Serializer;

12
13 -- Record Component Includes:
14 with Interfaces;

15
16 -- This is an example packed register.
17 package Example_Register is

18
19   -- Preamble code:
20   type Enable_Disable_Type is (Disable, Enable);
21   for Enable_Disable_Type use (Disable => 0, Enable => 1);
22   type Fourteen_Bit_Uint is mod 2**14;

23
24   -- Packed type size (in bits):
25   Size : constant Positive := 32;

26
27   -- Packed type size rounded up to nearest byte.
28   Size_In_Bytes : constant Positive := (Size - 1) /
29   -- Basic_Types.Byte'Object_Size + 1;

30   -- The total number of fields contained in the packed record, this includes
31   -- any fields of packed records included directly as a member in this
32   -- packed record.
33   Num_Fields : constant Positive := 4;

34
35   -- Unpacked type:
36   type U is record
37     -- Single bit to represent if some fictional hardware 1 is enabled or
38     -- disabled.
39     Hw_1_Enabled : Enable_Disable_Type;
40     -- Single bit to represent if some fictional hardware 2 is enabled or
41     -- disabled.
42     Hw_2_Enabled : Enable_Disable_Type;
43     -- This is a 14-bit unsigned integer.
44     Threshold : Fourteen_Bit_Uint;
45     -- The second half of the register is not used, but must be defined since
46     -- registers are 32-bits in size.
47     Unused : Interfaces.Unsigned_16;
48   end record;

49   -- Access type for U
50   type U_Access is access all U;

51
52   -- Packed type definition.
53   type T is new U
54     with Bit_Order => System.High_Order_First,
55           Scalar_Storage_Order => System.High_Order_First,
56           Size => Size,
57           Object_Size => Size,
58           Value_Size => Size,
59           Alignment => 1,
60           Volatile => False;

61   -- Packed type layout:
62   for T use record
63     -- Single bit to represent if some fictional hardware 1 is enabled or
64     -- disabled.

```

```

64      Hw_1_Enabled at 0 range 0 .. 0;
65      -- Single bit to represent if some fictional hardware 2 is enabled or
66      -- disabled.
67      Hw_2_Enabled at 0 range 1 .. 1;
68      -- This is a 14-bit unsigned integer.
69      Threshold at 0 range 2 .. 15;
70      -- The second half of the register is not used, but must be defined since
71      -- registers are 32-bits in size.
72      Unused at 0 range 16 .. 31;
73  end record;
74
75  -- Access type for T
76  type T_Access is access all T;
77
78  -- Packed type definition with little endian definition.
79  type T_Le is new U
80    with Bit_Order => System.Low_Order_First,
81        Scalar_Storage_Order => System.Low_Order_First,
82        Size => Size,
83        Object_Size => Size,
84        Value_Size => Size,
85        Alignment => 1,
86        Volatile => False;
87
88  -- Packed type layout:
89  for T_Le use record
90    -- Single bit to represent if some fictional hardware 1 is enabled or
91    -- disabled.
92    Hw_1_Enabled at 0 range 0 .. 0;
93    -- Single bit to represent if some fictional hardware 2 is enabled or
94    -- disabled.
95    Hw_2_Enabled at 0 range 1 .. 1;
96    -- This is a 14-bit unsigned integer.
97    Threshold at 0 range 2 .. 15;
98    -- The second half of the register is not used, but must be defined since
99    -- registers are 32-bits in size.
100   Unused at 0 range 16 .. 31;
101 end record;
102
103 -- Access type for T_Le
104 type T_Le_Access is access all T_Le;
105
106 -- Volatile packed type definition:
107 -- Note: This type is volatile. You should use this type to specify that the
108 -- variable in question may suddenly change in value. For example, this may
109 -- occur due to a device writing to a shared buffer. When this pragma is
110 -- used,
111 -- the compiler must suppress any optimizations that would interfere with the
112 -- correct reading of the volatile variables. For example, two successive
113 -- readings of the same variable cannot be optimized to just one or
114 -- reordered.
115 -- Important: You should use the Register type for accessing hardware
116 -- registers.
117 type Volatile_T is new T
118  with Bit_Order => System.High_Order_First,
119        Scalar_Storage_Order => System.High_Order_First,
120        Size => Size,
121        Object_Size => Size,
122        Value_Size => Size,
123        Alignment => 1,
124        Volatile => True;

```

```

119
120    -- Access type for Volatile_T
121    type Volatile_T_Access is access all Volatile_T;
122
123    -- Volatile little endian packed type definition:
124    -- Note: This type is volatile. You should use this type to specify that the
125    -- variable in question may suddenly change in value. For example, this may
126    -- occur due to a device writing to a shared buffer. When this pragma is
127    -- used,
128    -- the compiler must suppress any optimizations that would interfere with the
129    -- correct reading of the volatile variables. For example, two successive
130    -- readings of the same variable cannot be optimized to just one or
131    -- reordered.
132    -- Important: You should use the Register type for accessing hardware
133    -- registers.
134    type Volatile_T_Le is new T_Le
135        with Bit_Order => System.Low_Order_First,
136            Scalar_Storage_Order => System.Low_Order_First,
137            Size => Size,
138            Object_Size => Size,
139            Value_Size => Size,
140            Alignment => 1,
141            Volatile => True;
142
143    -- Access type for Volatile_T_Le
144    type Volatile_T_Le_Access is access all Volatile_T_Le;
145
146    --
147    -- Atomic and Register type definitions. These are only supported for types
148    -- that are
149    -- either 8, 16, or 32 bits in size.
150    --
151    -- Atomic packed type definition:
152    -- Note: This type is atomic. Use this type to specify that the code
153    -- generated must read and write the type or variable from memory atomically,
154    -- i.e. as a single/non-interruptible operation. It implies pragma Volatile,
155    -- the difference is that pragma Atomic is stronger: the compilation must
156    -- fail if the variable cannot be updated atomically.
157    -- Important: Atomic types create a synchronization point and can be used for
158    -- very limited intertask communication. However, protected objects should
159    -- almost
160    -- always be preferred.
161    type Atomic_T is new T
162        with Bit_Order => System.High_Order_First,
163            Scalar_Storage_Order => System.High_Order_First,
164            Size => Size,
165            Object_Size => Size,
166            Value_Size => Size,
167            Volatile => True,
168            Atomic => True;
169
170    -- Access type for Atomic_T
171    type Atomic_T_Access is access all Atomic_T;
172
173    -- Atomic little endian packed type definition:
174    -- Note: This type is atomic. Use this type to specify that the code
175    -- generated must read and write the type or variable from memory atomically,
176    -- i.e. as a single/non-interruptible operation. It implies pragma Volatile,
177    -- the difference is that pragma Atomic is stronger: the compilation must
178    -- fail if the variable cannot be updated atomically.
179    -- Important: Atomic types create a synchronization point and can be used for

```

```

175      -- very limited intertask communication. However, protected objects should
176      -- almost
177      -- always be preferred.
178      type Atomic_T_Le is new T_Le
179          with Bit_Order => System.Low_Order_First,
180              Scalar_Storage_Order => System.Low_Order_First,
181              Size => Size,
182              Object_Size => Size,
183              Value_Size => Size,
184              Volatile => True,
185              Atomic => True;
186
187      -- Access type for Atomic_T_Le
188      type Atomic_T_Le_Access is access all Atomic_T_Le;
189
190      -- Register packed type definition:
191      -- Note: This type is Volatile_Full_Access. This is similar in effect to
192      -- pragma Volatile, except that any reference to the object is guaranteed to
193      -- be done only with instructions that read or write all the bits of the
194      -- object.
195      -- Furthermore, if the object is of a composite type, then any reference to a
196      -- subcomponent of the object is guaranteed to read and/or write all the bits
197      -- of the object.
198      -- Important: You should use a Register type for accessing memory mapped IO
199      -- or
200      -- hardware registers.
201      type Register_T is new T
202          with Bit_Order => System.High_Order_First,
203              Scalar_Storage_Order => System.High_Order_First,
204              Size => Size,
205              Object_Size => Size,
206              Value_Size => Size,
207              Volatile => True,
208              Volatile_Full_Access => True;
209
210      -- Access type for Register_T
211      type Register_T_Access is access all Register_T;
212
213      -- Register little endian packed type definition:
214      -- Note: This type is Volatile_Full_Access. This is similar in effect to
215      -- pragma Volatile, except that any reference to the object is guaranteed to
216      -- be done only with instructions that read or write all the bits of the
217      -- object.
218      -- Furthermore, if the object is of a composite type, then any reference to a
219      -- subcomponent of the object is guaranteed to read and/or write all the bits
220      -- of the object.
221      -- Important: You should use a Register type for accessing memory mapped IO
222      -- or
223      -- hardware registers.
224      type Register_T_Le is new T_Le
225          with Bit_Order => System.Low_Order_First,
226              Scalar_Storage_Order => System.Low_Order_First,
227              Size => Size,
228              Object_Size => Size,
229              Value_Size => Size,
230              Volatile => True,
231              Volatile_Full_Access => True;
232
233      -- Access type for Register_T_Le
234      type Register_T_Le_Access is access all Register_T_Le;

```

```

231 -- Type conversion functions between packed an unpacked representations:
232 function Pack (Src : in U) return T with Inline => True;
233 function Unpack (Src : in T) return U with Inline => True;
234 function Pack (Src : in U) return T_Le with Inline => True;
235 function Unpack (Src : in T_Le) return U with Inline => True;
236
237 -- Endianness conversion functions
238 function Swap_Endianness (Src : in T) return T_Le with Inline => True;
239 function Swap_Endianness (Src : in T_Le) return T with Inline => True;
240
241 -- Serializing functions for entire record:
242 package Serialization is new Serializer (T);
243 package Serialization_Le is new Serializer (T_Le);
244
245 -- The length in bytes of the serialized type.
246 Max_Serialized_Length : Natural renames Serialization.Serialized_Length; --
247 -- in bytes
247 -- The length in bytes of the serialized type.
248 Min_Serialized_Length : Natural renames Serialization.Serialized_Length; --
249 -- in bytes
250
251 -- Convenience function which always returns Success and the length defined
252 -- above ^. This
253 -- is useful when you want to instantiate a generic which requires a function
254 -- of the
255 -- definition below, but its for a statically sized type.
256 function Serialized_Length (Src : in T; Num_Bytes_Serialized : out Natural)
257 -- return Serialization_Status
258 -- with Inline => True;
259 function Serialized_Length_Le (Src : in T_Le; Num_Bytes_Serialized : out
260 -- Natural) return Serialization_Status
261 -- with Inline => True;
262
263 -- Get the size of the already serialized type inside of the byte array. If
264 -- the byte array is too small then
265 -- a serialization error is returned.
266 function Serialized_Length (Src : in Basic_Types.Byte_Array;
267 -- Num_Bytes_Serialized : out Natural) return Serialization_Status
268 -- with Inline => True;
269 function Serialized_Length_Le (Src : in Basic_Types.Byte_Array;
270 -- Num_Bytes_Serialized : out Natural) return Serialization_Status
271 -- with Inline => True;
272
273 -----
274 -- Types related to record's internal fields:
275 -----
276 -- Field type sizes (in bits):
277 Hw_1_Enabled_Size : constant Positive := 1;
278 Hw_2_Enabled_Size : constant Positive := 1;
279 Threshold_Size : constant Positive := 14;
280 Unused_Size : constant Positive := 16;
281
282 -- Packed field sizes in bytes rounded up to nearest byte boundary.
283 Hw_1_Enabled_Size_In_Bytes : constant Positive := (Hw_1_Enabled_Size - 1) /
284 -- Basic_Types.Byte'Object_Size + 1;
285 Hw_2_Enabled_Size_In_Bytes : constant Positive := (Hw_2_Enabled_Size - 1) /
286 -- Basic_Types.Byte'Object_Size + 1;
287 Threshold_Size_In_Bytes : constant Positive := (Threshold_Size - 1) /
288 -- Basic_Types.Byte'Object_Size + 1;
289 Unused_Size_In_Bytes : constant Positive := (Unused_Size - 1) /
290 -- Basic_Types.Byte'Object_Size + 1;

```

```

279
280 -----
281 -- Named subtypes for each field for convenience:
282 -----
283
284 subtype Hw_1_Enabled_Type is Enable_Disable_Type;
285 subtype Hw_2_Enabled_Type is Enable_Disable_Type;
286 subtype Threshold_Type is Fourteen_Bit_Uint;
287 subtype Unused_Type is Interfaces.Unsigned_16;
288
289 end Example_Register;

```

Notice the new types `.Atomic_T`, `.Atomic_T_Le`, `.Register_T`, and `.Register_T_Le`. The register type can be used as follows:

```

1  with Example_Register;
2  with System.Storage_Elements; use System.Storage_Elements;
3
4 procedure Main is
5   use Example_Register;
6   -- Define a hardware register on a little endian machine:
7   Reg : Example_Register.Register_T_Le;
8   for Reg'Address use To_Address (Integer_Address (16#0060_0014#));
9   -- Define a register copy which is of the normal type:
10  Reg_Copy : Example_Register.T_Le;
11 begin
12   -- Read entire register atomically:
13   Reg_Copy := T_Le (Reg);
14
15   -- Write entire register atomically:
16   Reg_Copy.Threshold := 17;
17   Reg := Register_T_Le (Reg_Copy);
18
19   -- Read and write register but only access certain components. The
20   -- compiler will ensure that the ENTIRE register is read/written
21   -- during the following operations.
22   if Reg.Hw_1_Enabled = Enable and then Reg.Hw_2_Enabled = Enable then
23     Reg.Threshold := 22;
24   end if;
25 end Main;

```

Notice that we have defined the register type for a little endian machine and tied it to a specific hardware address. We then access the register by reading the entire register into a copy, modifying that copy, and then writing the register back into place. Generally, this is the pattern that should be followed when working with hardware registers, as it is most efficient and readable. However, the second set of code also works due to the `Volatile_Full_Access` type specification. Two bits are read from the register and then a 14-bit number is written. With a normal packed record, this operation would fail since likely the compiler would not read or write the entire 32-bit register during the operation, resulting in unintended behavior within the hardware. However, with the `.Register_T_Le` type, the entire register is read for each of the two comparisons and a read-modify-write sequence is used to set the 14-bit threshold. While this works as intended, the first pattern is usually preferred as it will result in more efficient code.

5.4.3 Packed Register Set

The previous section demonstrated how to define and access a hardware register using Adamant packed records. Complex hardware devices are often defined as a set of registers. While a set of registers can be defined as a regular packed record in Adamant, you will not gain access to the `Volatile_Full_Access` pragma for each of the individual registers which could result in errant reads/writes of the registers. The proper way to define a register set in Adamant is to create an

encompassing packed record which holds each of the individual register components. In other words, we are creating a packed record with many nested packed records inside, each which represent an individual register. See Section 5.1.10 for more information on nesting packed records.

The following defines a register set which consists of three registers of the type `Example_Register.Register_T_Le`. Mixed endianness is not currently allowed within a single packed record. The Register types are used to ensure that the compiler produces code for accessing each individual register by reading/writing all bits of the register atomically.

```

1  ---
2  description: This is an example register set.
3  # The list of registers. In this case, each register
4  # has the same bit layout.
5  fields:
6  - name: Reg_1
7  description: The first register.
8  type: Example_Register.Register_T_Le
9  - name: Reg_2
10 description: The second register.
11 type: Example_Register.Register_T_Le
12 - name: Reg_3
13 description: The third register.
14 type: Example_Register.Register_T_Le

```

Note that when you define a packed record with a `Volatile`, `Atomic`, or `Register` type, ALL components of that packed record must be either `Volatile`, `Atomic`, or `Register`. It does not make sense to define a record with only a portion of the record being `Volatile`, `Atomic`, or `Register` type.

The resulting generated code is different from that normally produced for a packed record. Below is the output `example_register_set.ads`:

```

1  -----
2  -- Example_Register_Set Record Spec
3  --
4  -- Generated from example_register_set.record.yaml on 2025-07-15 21:58.
5  -----
6
7  -- Standard Includes:
8  with Basic_Types;
9  with System;
10 with Serializer_Types; use Serializer_Types;
11 with Serializer;
12
13 -- Record Component Includes:
14 with Example_Register;
15
16 -- This is an example register set.
17 package Example_Register_Set is
18
19   -- Packed type size (in bits):
20   Size : constant Positive := 96;
21
22   -- Packed type size rounded up to nearest byte.
23   Size_In_Bytes : constant Positive := (Size - 1) /
24     ↳ Basic_Types.Byte'Object_Size + 1;
25
26   -- The total number of fields contained in the packed record, this includes
27   -- any fields of packed records included directly as a member in this
28   -- packed record.
29   Num_Fields : constant Positive := 12;

```

```

29
30    -- Unpacked type:
31    type U is record
32        -- The first register.
33        Reg_1 : Example_Register.U;
34        -- The second register.
35        Reg_2 : Example_Register.U;
36        -- The third register.
37        Reg_3 : Example_Register.U;
38    end record;
39
40    -- Access type for U
41    type U_Access is access all U;
42
43    -- Packed type definition with little endian definition.
44    type T_Le is record
45        Reg_1 : Example_Register.T_Le;
46        Reg_2 : Example_Register.T_Le;
47        Reg_3 : Example_Register.T_Le;
48    end record
49    with Bit_Order => System.Low_Order_First,
50        Scalar_Storage_Order => System.Low_Order_First,
51        Size => Size,
52        Object_Size => Size,
53        Value_Size => Size,
54        Alignment => 1,
55        Volatile => False;
56
57    -- Packed type layout:
58    for T_Le use record
59        -- The first register.
60        Reg_1 at 0 range 0 .. 31;
61        -- The second register.
62        Reg_2 at 0 range 32 .. 63;
63        -- The third register.
64        Reg_3 at 0 range 64 .. 95;
65    end record;
66
67    -- Access type for T_Le
68    type T_Le_Access is access all T_Le;
69
70    --
71    -- Note: This register type is created because the model is defined with all
72    -- register fields.
73    --
74
75    -- Register packed little endian type:
76    type Register_T_Le is record
77        -- The first register.
78        Reg_1 : Example_Register.Register_T_Le;
79        -- The second register.
80        Reg_2 : Example_Register.Register_T_Le;
81        -- The third register.
82        Reg_3 : Example_Register.Register_T_Le;
83    end record
84    with Bit_Order => System.Low_Order_First,
85        Scalar_Storage_Order => System.Low_Order_First,
86        Size => Size,
87        Object_Size => Size,
88        Value_Size => Size,
89        Alignment => 4, -- Cannot be aligned at 1, must be aligned at word
90        -- boundary

```

```

89      Volatile => True;
90
91      -- Register packed type layout:
92      for Register_T_Le use record
93          -- The first register.
94          Reg_1 at 0 range 0 .. 31;
95          -- The second register.
96          Reg_2 at 0 range 32 .. 63;
97          -- The third register.
98          Reg_3 at 0 range 64 .. 95;
99      end record;
100
101     -- Access type for Register_T_Le
102     type Register_T_Le_Access is access all Register_T_Le;
103
104     -- Type conversion functions between packed an unpacked representations:
105     function Pack (Src : in U) return T_Le;
106     function Unpack (Src : in T_Le) return U;
107
108     -- Serializing functions for entire record:
109     package Serialization_Le is new Serializer (T_Le);
110
111     -- The length in bytes of the serialized type.
112     Max_Serialized_Length : Natural renames Serialization_Le.Serialized_Length;
113     -- in bytes
114     -- The length in bytes of the serialized type.
115     Min_Serialized_Length : Natural renames Serialization_Le.Serialized_Length;
116     -- in bytes
117
118     -- Convenience function which always returns Success and the length defined
119     -- above ^. This
120     -- is useful when you want to instantiate a generic which requires a function
121     -- of the
122     -- definition below, but its for a statically sized type.
123     function Serialized_Length_Le (Src : in T_Le; Num_Bytes_Serialized : out
124     -- Natural) return Serialization_Status
125         with Inline => True;
126
127     -- Get the size of the already serialized type inside of the byte array. If
128     -- the byte array is too small then
129     -- a serialization error is returned.
130     function Serialized_Length_Le (Src : in Basic_Types.Byte_Array;
131     -- Num_Bytes_Serialized : out Natural) return Serialization_Status
132         with Inline => True;
133
134     -----
135     -- Types related to record's internal fields:
136     -----
137     -- Field type sizes (in bits):
138     Reg_1_Size : constant Positive := Example_Register.Size;
139     Reg_2_Size : constant Positive := Example_Register.Size;
140     Reg_3_Size : constant Positive := Example_Register.Size;
141
142     -- Packed field sizes in bytes rounded up to nearest byte boundary.
143     Reg_1_Size_In_Bytes : constant Positive := Example_Register.Size_In_Bytes;
144     Reg_2_Size_In_Bytes : constant Positive := Example_Register.Size_In_Bytes;
145     Reg_3_Size_In_Bytes : constant Positive := Example_Register.Size_In_Bytes;
146
147     -----
148     -- Named subtypes for each field for convenience:
149     -----

```

```

143 subtype Reg_1_Type_U is Example_Register.U;
144 subtype Reg_1_Type_T_Le is Example_Register.T_Le;
145 subtype Reg_2_Type_U is Example_Register.U;
146 subtype Reg_2_Type_T_Le is Example_Register.T_Le;
147 subtype Reg_3_Type_U is Example_Register.U;
148 subtype Reg_3_Type_T_Le is Example_Register.T_Le;
149
150 end Example_Register_Set;

```

First, note that there is U and T_Le types declared which include .U and .T_Le versions of the underlying registers. These may be useful in some situations. However, a Register_T_Le is also created that includes the Register_T_Le version of each field. This represents a register set that adheres to the Volatile_Full_Access semantics when the individual fields are read from or written to.

The register set can be used like any normal packed record, except that accessing the individual records is safe for the hardware due to the underlying Volatile_Full_Access type specification. The following example expands on the example provided in the previous section for a single register. This time we access different registers found within the register set.

```

1 with Example_Register;
2 with Example_Register_Set;
3 with System.Storage_Elements; use System.Storage_Elements;
4
5 procedure Main is
6   use Example_Register;
7   -- Define hardware register set:
8   Registers : Example_Register_Set.Register_T_Le;
9   for Registers'Address use To_Address (Integer_Address (16#0060_0014#));
10  -- Define a register copy:
11  Reg_Copy : Example_Register.T_Le;
12 begin
13   -- Read the entire first register atomically:
14   Reg_Copy := T_Le (Registers.Reg_1);
15
16   -- Write entire register atomically:
17   Reg_Copy.Threshold := 17;
18   Registers.Reg_1 := Register_T_Le (Reg_Copy);
19
20   -- Read and write registers but only access certain components. The
21   -- compiler will ensure that the ENTIRE register is read/written
22   -- during the following operations.
23   if Registers.Reg_2.Hw_1_Enabled = Enable and then
24     Registers.Reg_3.Hw_2_Enabled = Enable
25   then
26     Registers.Reg_1.Threshold := 22;
27   end if;
28 end Main;

```

5.4.4 Packed Register Arrays

If all the registers in a register set are of the same type you can use a packed array instead of a packed record to describe the hardware. For example, to describe a set of 10 32-bit registers that should be interpreted as unsigned integers you can define the following packed array.

```

1 ---
2 description: This is an array of 10 32-bit unsigned registers.
3 type: Interfaces.Unsigned_32

```

```
4  format: U32
5  length: 10
```

This model produces the following generated code:

```
1  -----
2  -- Example_Register_Array Array Spec
3  --
4  -- Generated from example_register_array.array.yaml on 2025-07-15 21:58.
5  -----
6
7  -- Standard Includes:
8  with System;
9  with Serializer;
10 with Basic_Types;
11
12 -- Type Includes:
13 with Interfaces;
14
15 -- This is an array of 10 32-bit unsigned registers.
16 package Example_Register_Array is
17
18     -- Packed type size (in bits):
19     Length : constant Natural := 10;
20     Element_Size : constant Positive := 32;
21     Size : constant Positive := Element_Size * Length;
22
23     -- Packed type size rounded up to nearest byte.
24     Element_Size_In_Bytes : constant Positive := (Element_Size - 1) /
25         Basic_Types.Byte'Object_Size + 1;
26     Size_In_Bytes : constant Positive := (Size - 1) / 8 + 1;
27
28     -- Array index type:
29     subtype Unconstrained_Index_Type is Natural;
30     subtype Constrained_Index_Type is Unconstrained_Index_Type range 0 .. Length
31         - 1;
32
33     -- Unconstrained base type:
34     type Unconstrained is array (Unconstrained_Index_Type range <>) of
35         Interfaces.Unsigned_32;
36
37     -- Unpacked array type:
38     subtype U is Unconstrained (Constrained_Index_Type);
39
40     -- Access type for U
41     type U_Access is access all U;
42
43     -- Packed type definition.
44     type T is new U
45         with Component_Size => Element_Size,
46             Scalar_Storage_Order => System.High_Order_First,
47             Size => Size,
48             Object_Size => Size,
49             Volatile => False,
50             Volatile_Components => False;
51
52     -- Access type for T
53     type T_Access is access all T;
54
55     -- Packed type definition with little endian definition.
56     type T_Le is new U
```

```

54      with Component_Size => Element_Size,
55          Scalar_Storage_Order => System.Low_Order_First,
56          Size => Size,
57          Object_Size => Size,
58          Volatile => False,
59          Volatile_Components => False;
60
61      -- Access type for T_Le
62      type T_Le_Access is access all T_Le;
63
64      -- Volatile packed type definition:
65      -- Note: This type is volatile. You should use this type to specify that the
66      -- variable in question may suddenly change in value. For example, this may
67      -- occur due to a device writing to a shared buffer. When this pragma is
68      -- used,
69      -- the compiler must suppress any optimizations that would interfere with the
70      -- correct reading of the volatile variables. For example, two successive
71      -- readings of the same variable cannot be optimized to just one or
72      -- reordered.
73      -- Important: You should use the Register type for accessing hardware
74      -- registers.
75      type Volatile_T is new T
76          with Component_Size => Element_Size,
77              Scalar_Storage_Order => System.High_Order_First,
78              Size => Size,
79              Object_Size => Size,
80              Volatile => True,
81              Volatile_Components => True;
82
83      -- Access type for Volatile_T
84      type Volatile_T_Access is access all Volatile_T;
85
86      -- Volatile little endian packed type definition:
87      -- Note: This type is volatile. You should use this type to specify that the
88      -- variable in question may suddenly change in value. For example, this may
89      -- occur due to a device writing to a shared buffer. When this pragma is
90      -- used,
91      -- the compiler must suppress any optimizations that would interfere with the
92      -- correct reading of the volatile variables. For example, two successive
93      -- readings of the same variable cannot be optimized to just one or
94      -- reordered.
95      -- Important: You should use the Register type for accessing hardware
96      -- registers.
97      type Volatile_T_Le is new T_Le
98          with Component_Size => Element_Size,
99              Scalar_Storage_Order => System.Low_Order_First,
100             Size => Size,
101             Object_Size => Size,
102             Volatile => True,
103             Volatile_Components => True;
104
105     -- Access type for Volatile_T_Le
106     type Volatile_T_Le_Access is access all Volatile_T_Le;
107
108     -- Atomic type definitions. These are only supported for array types that
109     -- have components that are exactly 32 bits in size and are not themselves
110     -- packed types.
111     --
112
113     -- Atomic packed type definition:

```

```

109  -- Note: This type is atomic. Use this type to specify that the code
110  -- generated must read and write the type or variable from memory atomically,
111  -- i.e. as a single/non-interruptible operation. It implies pragma Volatile,
112  -- the difference is that pragma Atomic is stronger: the compilation must
113  -- fail if the variable cannot be updated atomically.
114  -- Important: Atomic types create a synchronization point and can be used for
115  -- very limited intertask communication. However, protected objects should
116  -- almost
117  -- always be preferred.
118  type Atomic_T is new T
119      with Component_Size => Element_Size,
120          Scalar_Storage_Order => System.High_Order_First,
121          Size => Size,
122          Object_Size => Size,
123          Volatile => True,
124          Volatile_Components => True,
125          Atomic_Components => True;
126
127  -- Access type for Atomic_T
128  type Atomic_T_Access is access all Atomic_T;
129
130  -- Atomic little endian packed type definition:
131  -- Note: This type is atomic. Use this type to specify that the code
132  -- generated must read and write the type or variable from memory atomically,
133  -- i.e. as a single/non-interruptible operation. It implies pragma Volatile,
134  -- the difference is that pragma Atomic is stronger: the compilation must
135  -- fail if the variable cannot be updated atomically.
136  -- Important: Atomic types create a synchronization point and can be used for
137  -- very limited intertask communication. However, protected objects should
138  -- almost
139  -- always be preferred.
140  type Atomic_T_Le is new T
141      with Component_Size => Element_Size,
142          Scalar_Storage_Order => System.Low_Order_First,
143          Size => Size,
144          Object_Size => Size,
145          Volatile => True,
146          Volatile_Components => True,
147          Atomic_Components => True;
148
149  -- Access type for Atomic_T_Le
150  type Atomic_T_Le_Access is access all Atomic_T_Le;
151
152  -- Register packed type definition:
153  -- Note: This type is the same as Atomic for arrays with component sizes of
154  -- 32-bits.
155  -- Important: You should use a Register type for accessing memory mapped IO
156  -- or
157  -- hardware registers.
158  type Register_T is new T
159      with Component_Size => Element_Size,
160          Scalar_Storage_Order => System.High_Order_First,
161          Size => Size,
162          Object_Size => Size,
163          Volatile => True,
164          Volatile_Components => True,
165          Atomic_Components => True;
166
167  -- Access type for Register_T
168  type Register_T_Access is access all Register_T;

```

```

166  -- Register little endian packed type definition:
167  -- Note: This type is the same as Atomic for arrays with component sizes of
168  -- 32-bits.
169  -- Important: You should use a Register type for accessing memory mapped IO
170  -- or
171  -- hardware registers.
172  type Register_T_Le is new T_Le
173      with Component_Size => Element_Size,
174          Scalar_Storage_Order => System.Low_Order_First,
175          Size => Size,
176          Object_Size => Size,
177          Volatile => True,
178          Volatile_Components => True,
179          Atomic_Components => True;
180
181
182  -- Access type for Register_T_Le
183  type Register_T_Le_Access is access all Register_T_Le;
184
185
186  -- Type conversion functions between packed an unpacked representations:
187  function Pack (Src : in U) return T with Inline => True;
188  function Unpack (Src : in T) return U with Inline => True;
189  function Pack (Src : in U) return T_Le with Inline => True;
190  function Unpack (Src : in T_Le) return U with Inline => True;
191
192  -- Endianness conversion functions
193  function Swap_Endianness (Src : in T) return T_Le with Inline => True;
194  function Swap_Endianness (Src : in T_Le) return T with Inline => True;
195
196  -- Serializing functions for entire packed array:
197  package Serialization is new Serializer (T);
198  package Serialization_Le is new Serializer (T_Le);
199
200
201  -- Packed type definition for array element:
202  subtype Element_Packed is Interfaces.Unsigned_32
203      with Object_Size => Element_Size_In_Bytes * 8,
204          Value_Size => Element_Size_In_Bytes * 8;
205
206
207  -- Serializing functions for an element of the array:
208  package Element_Serialization is new Serializer (Element_Packed);
209
210
211  -----
212  -- Named subtypes for array element for convenience:
213  -----
214
215
216  subtype Element_Type is Interfaces.Unsigned_32;
217
218 end Example_Register_Array;

```

Because the array components are 32-bits in size, the autocode includes an `Atomic_T`, `Atomic_T_Le`, `Register_T`, and `Register_T_Le` types which can be used to provide `Atomic` type access. Note that for arrays of 32-bit components an `Atomic` component type is equivalent to `Volatile_Full_Access` array component type. Ada does not support specifying the latter, but Adamant provides a type for the `Atomic` type with the name `Register`.

You can also declare a register array that is made up of packed register record elements. For example, the following array can be defined:

```

1  ---
2  description: This is an array of 3 32-bit registers with a packed record
2  -- definition.
3  type: Example_Register.Register_T_Le

```

```
4 length: 10
```

In this case we have an array of 10 elements, each of Register_T_Le type. The following autocode is produced:

```
1 -----
2 -- Example_Packed_Register_Array Array Spec
3 --
4 -- Generated from example_packed_register_array.array.yaml on 2025-07-15 21:58.
5 -----
6 
7 -- Standard Includes:
8 with System;
9 with Serializer;
10 
11 -- Type Includes:
12 with Example_Register;
13 
14 -- This is an array of 3 32-bit registers with a packed record definition.
15 package Example_Packed_Register_Array is
16 
17     -- Packed type size (in bits):
18     Length : constant Natural := 10;
19     Element_Size : constant Positive := Example_Register.Size;
20     Size : constant Positive := Element_Size * Length;
21 
22     -- Packed type size rounded up to nearest byte.
23     Element_Size_In_Bytes : constant Positive := Example_Register.Size_In_Bytes;
24     Size_In_Bytes : constant Positive := (Size - 1) / 8 + 1;
25 
26     -- Array index type:
27     subtype Unconstrained_Index_Type is Natural;
28     subtype Constrained_Index_Type is Unconstrained_Index_Type range 0 .. Length
29         - 1;
30 
31     -- Unconstrained base type:
32     type Unconstrained is array (Unconstrained_Index_Type range <>) of
33         Example_Register.U;
34 
35     -- Unpacked array type:
36     subtype U is Unconstrained (Constrained_Index_Type);
37 
38     -- Access type for U
39     type U_Access is access all U;
40 
41     -- Packed type definition with little endian definition.
42     type T_Le is array (Constrained_Index_Type) of Example_Register.T_Le
43         with Component_Size => Element_Size,
44             Scalar_Storage_Order => System.Low_Order_First,
45             Size => Size,
46             Object_Size => Size,
47             Volatile => False,
48             Volatile_Components => False;
49 
50     -- Access type for T_Le
51     type T_Le_Access is access all T_Le;
52 
53     --
54     -- Note: This register type is created because the model is defined with all
55         -- register fields.
56     --
```

```

54
55    -- Register little endian packed type definition:
56    -- Note: This type is register. You should use this type to specify that the
57    -- variable in question may suddenly change in value. For example, this may
58    -- occur due to a device writing to a shared buffer. When this pragma is
59    -- used,
60    -- the compiler must suppress any optimizations that would interfere with the
61    -- correct reading of the register variables. For example, two successive
62    -- readings of the same variable cannot be optimized to just one or
63    -- reordered.
64    -- Important: You should use the Register type for accessing hardware
65    -- registers.
66
67    type Register_T_Le is array (Constrained_Index_Type) of
68        Example_Register.Register_T_Le
69        with Component_Size => Element_Size,
70            Scalar_Storage_Order => System.Low_Order_First,
71            Size => Size,
72            Object_Size => Size,
73            Volatile => True,
74            Volatile_Components => True;
75
76
77    type Register_T_Le_Access is access all Register_T_Le;
78    -- Type conversion functions between packed an unpacked representations:
79    function Pack (Src : in U) return T_Le;
80    function Unpack (Src : in T_Le) return U;
81
82
83    -- Serializing functions for entire packed array:
84    package Serialization_Le is new Serializer (T_Le);
85
86    -- Serializing functions for an element of the array:
87    package Element_Serialization renames Example_Register.Serialization;
88
89    end Example_Packed_Register_Array;

```

This is treated similar to a packed register set, described in the previous section. Because all the array components are of type Register, a type Register_T and Register_T_Le is provided.

The code below illustrates using these arrays to read/write registers.

```

1  with Example_Register;
2  with Example_Packed_Register_Array;
3  with System.Storage_Elements; use System.Storage_Elements;
4
5  procedure Main is
6      use Example_Register;
7      -- Define hardware register array:
8      Registers : Example_Packed_Register_Array.Register_T_Le;
9      for Registers'Address use To_Address (Integer_Address (16#0070_0014#));
10     -- Define register copy:
11     Reg_Copy : Example_Register.T_Le;
12 begin
13     -- Read the entire third register atomically:
14     Reg_Copy := T_Le (Registers (3));
15

```

```
16    -- Write entire register fifth atomically:  
17    Registers (5) := Register_T_Le (Reg_Copy);  
18  
19    -- Read and write registers but only access certain components. The  
20    -- compiler will ensure that the ENTIRE register is read/written  
21    -- during the following operations.  
22    if Registers (1).Hw_1_Enabled = Enable and then  
23        Registers (4).Hw_2_Enabled = Enable  
24    then  
25        Registers (7).Threshold := 22;  
26    end if;  
27 end Main;
```

6 Components

Adamant is a component-based framework. Component-based frameworks approach application design as a set of reusable functional or logical entities called components, which expose well-defined interfaces for inter-component communication. The motivation behind Adamant's component-based architecture is thoroughly discussed in the [Architectural Description Document](#). An understanding of the concepts in that document is required before diving into building your own component, which is the purpose of this section.

The beginning of this section outlines the different component patterns that are common within Adamant constructed software. The following list is copied from the Archectural Description Document and specifies the User Guides section where each component pattern is presented.

- **Passive without Queue**, Section 6.1 - This type of component will only execute when a caller invokes one of its connectors. This type of component is typically used to provide a synchronous service to other components, such as getting the system time, or updating a parameter database.
- **Passive with Queue**, Section 6.3 - Like the *Passive without Queue* component, this component will also only execute when called. However, because it has a queue, this component is typically used to respond to asynchronous messages at a scheduled time. Execution is commonly invoked through a "schedule" connector, which tells the component to service its queue and possibly do other periodic work. This design is ideal for components that need to be executed with cyclic real-time deadlines, such as a control loop.
- **Active with Queue**, Section 6.4 - The default behavior of components that follow this pattern is to block on their queue and only wake up and do work when a message is received asynchronously. This type of component is ideal for components that run at a specified priority and receive asynchronous messages, such as a logger. This default task behavior can be overridden if necessary by the developer, but this should be avoided if possible, since the component diagram will not give insight into this custom behavior.
- **Active without Queue**, Section 6.5 - Unlike *Active with Queue* components, there is no default behavior for the *Active without Queue* task. In this case, the developer must define the code which executes on the component's thread. This component pattern is uncommon, and not recommended, since the component's behavior cannot be readily ascertained from its diagram. However, this design might be useful for components that run in the background, but receive no asynchronous input, such as a memory scrubber.

Each pattern builds on the previous pattern. It is recommended that for best understanding of an active component with queue to first read the sections for a passive component without queue followed by a passive component with queue, before proceeding to the active component with queue section.

6.1 Creating a Passive Component

In this section we will be discuss how to create a component called *example_component*. We will start simple, and then add (and subtract) features throughout subsequent sections to demonstrate the features of Adamant components and how to use them.

Note: These sections often use the phrase building "components". To be more specific, what we are really creating here are *component types*, which can be thought of as analogous to a classes in the object-oriented sense. Later, in Section 7, we will talk about creating instantiations of these *component types* called *component instances*. *Component instances* are analogous to an object in the object-oriented sense. This distinction is important because more than one *component instance* can be created from a single *component type*. From now on, we will talk about building "components," but remember, we are actually creating *component types*.

Components are usually created in their own directory. Let's create one for our example component.

```

> mkdir example_component # make component directory
> cd example_component
> touch .all_path          # add directory to build path

```

To define a component in this directory, a developer must create a YAML model file in the form *name.component.yaml* where *name* is the desired name of the *component type*. Below is the component definition for the *example_component*. This definition is stored in a file called *example_component.component.yaml* within this new directory.

```

1  ---
2  execution: passive
3  # Optional - description of component
4  description: This is the example component.
5  # Required - List of Connectors
6  connectors:
7  #####
8  # Invokee Connectors
9  #####
10 # Required - the datatype to be passed along connector
11 - type: Tick.T
12   # Required - the kind of connector
13   kind: recv_sync
14   # Optional - name of connector
15   name: Tick_T_Recv_Sync
16   # Optional - description of connector
17   description: This connector provides the schedule tick for the component.
18 #####
19 # Invoker Connectors
20 #####
21 - return_type: Sys_Time.T
22   kind: get
23   description: This connector is used to fetch the current system time.
24 - type: Packet.T
25   kind: send
26   # Optional - define the size of an arrayed invoker connector, a value
27   # of 0 signifies that the connector array size is unconstrained, and
28   # will be constrained by the assembly. By default, if this is not
29   # specified, then count is set to 1, and the connector will not be
30   # arrayed.
31   count: 0
32   description: This connector is used to send out a telemetry packet.

```

Comments are included in the model to explain whether or not each field is optional or required. A component must always specify its *execution*. The details on component *execution* are discussed in the [Architectural Description Document](#) and copied below for reference:

- **Passive** - Passive components do not have a thread of their own and are expected to execute on the thread of their connector invokers.
- **Active** - Active components contain their own thread on which they execute.
- **Either** - Either components are designed to function either as passive or active components. Which *execution* type a component will get resolved to is decided at the assembly level.

The component we have defined above is a *passive* component, which means that it does not have its own thread of execution, and will only execute on the thread of any connector *invokers*. In this case, the only way the component can execute is through a call to its *Tick_T_Recv_Sync* connector.

The most important part of any component model is its connector definition. This YAML model declares *example_component* with three connectors. One connector is an *invokee* connector of type

`Tick.T`. This connector is a scheduling connector. When the connector is called the component does its “work”. The component also has two *invoker* connectors. The first is used to fetch the system time via type `Sys_Time.T`. The second is used to send out a packet of type `Packet.T`.

In the model, for each connector we have not only defined the connector *types* (and *return_types*), which are all *packed records* (see Section 5.1), but we also define the connector *kind*. Connector *kinds* are discussed thoroughly in the [Architectural Description Document](#) but a diagram of all possible connector kinds is shown below for reference:

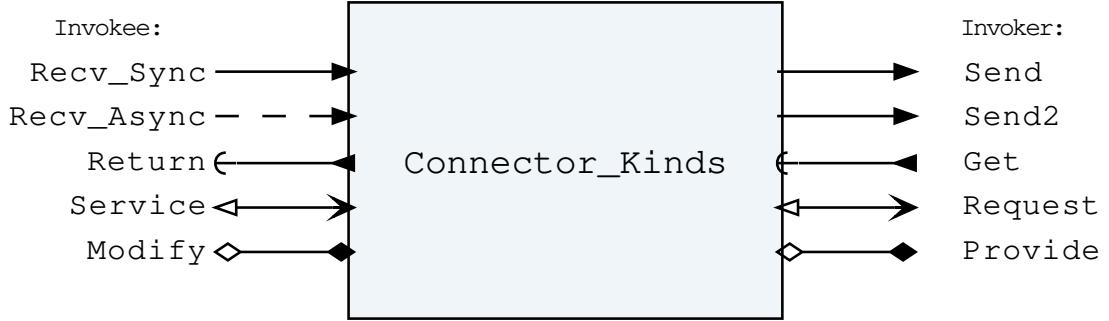


Figure 4: Example component diagram to demonstrate different connector *kinds*

Recall that connector *kinds* are used to specify the direction and synchronicity of data flow along a connector. Connector *kinds* fall into one of four categories, whose names correspond to the direction of the data flow with respect to the *invokee*. A summary of the available connector *kind* pairs are shown below.

Category	Invoker Kind	Invokee Kind	Has <i>type</i> ?	Has <i>return_type</i> ?
In	send	<code>recv_sync</code> or <code>recv_async</code>	yes	no
Out	get	<code>return</code>	no	yes
In Return	request	<code>service</code>	yes	yes
In Out	provide	<code>modify</code>	yes	no

The chosen connector *kind* pair determines whether or not a *type*, *return_type*, or both is required for the connector definition. Note that a model error will be produced if you fail to comply with the table above, so have no fear of making mistakes.

The definition of the `Packet_T_Send` connector specifies the optional field `count`. The `count` field is used to specify an *arrayed* connector. When not specified, the connector `count` is assumed to be 1, which means that the connector is not arrayed. If a positive value greater than 1 is provided, then the component is created with an array of connectors of the specified size. If the `count` is defined as 0, as is the case here, then the connector array size is said to be *unconstrained* and will be set during the component’s instantiation within an assembly, see Section 7. In this case we have created an unconstrained array of `Packed.T send` connectors, which can be called by index.

Note that we have defined an optional *name* for the `Tick_T_Recv_Sync` connector. If no *name* is specified then Adamant will generate a name for the connector in the form *type_kind* where *type* is the connector type with any “.”s replaced by “_”s and *kind* is the connector’s *kind*. In general, it is considered good practice to NOT provide names for connectors, as the autocoded names are informative and follow a convention that provides information about of the connector’s form and function. However, there are certainly exceptions to this rule. Sometimes defining a custom name makes more sense, like in the case where a component has two connectors of the same *type* that

perform distinct behaviors.

6.1.1 Creating a Component Diagram

Components in Adamant are often presented using diagrams, which contain a visual representation of the semantics encoded in the component YAML file. A component diagram can be generated from the *example_component* model by running the following from the component directory:

```
> redo build/svg/example_component.svg
```

which produces an SVG diagram of the component that can be viewed in any web browser. An *eps* version, which is easy to import into PDF documentation can also be constructed:

```
> redo build/eps/example_science.eps
```

and the output is shown below:



As can be seen, the *example_component* is *passive* because the outline around the component is not bold. It has a single *invokee* connector named *Tick_T_Recv_Sync* and two *invoker* connectors called *Sys_Time_T_Get* and *Packet_T_Send*. *Packet_T_Send* is denoted as an arrayed connector using the *[<>]* notation, where *<>* signifies that the connector array size is unconstrained, and will be determined by the assembly.

6.1.2 Anatomy of a Component

Before we begin to implement the *example_component*, we need to understand how a component is constructed using the Ada language. Below is a UML-like class diagram that shows the Ada packages that make up a component.

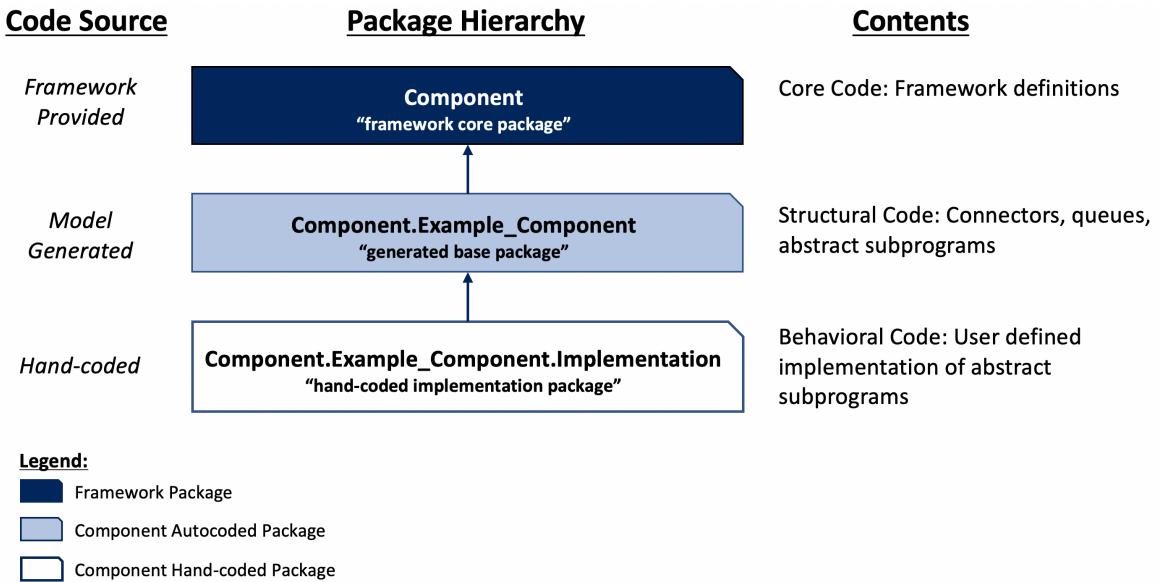


Figure 5: This diagram shows the three different Ada packages that make up a component.

The top package in the diagram is also the upper most “parent” in an object-oriented sense. Each subsequent package is a child package of the first, which in Ada means they have access to the private definitions within the parent package. In addition, each subsequent package includes a derived tagged type called `Instance`, that derives from the parent package `Instance`. In this way, each `Instance` type “extends” the functionality of the parent `Instance` type. There are three inheritance levels that make up a component in Adamant, and they each serve a unique purpose.

The top level parent is the `Component` package, whose source can be found in the Adamant core in `src/core/component`. This package includes definitions and subprograms that are shared by all components in Adamant. Most importantly, this package defines the *class-wide* component type that all components inherit from. This class-wide type is what allows dynamic dispatching within components, and is the key language feature needed to implement connectors.

The next level is unique to every *component type* definition. That is, every component YAML model generates a different version of this base package. For the YAML that we created for the `example_component` in the previous sections, an autocoded base package with the name `Component.Example_Component` will be generated in the files `component-example_component.ads` and `component-example_component.adb`. The steps for generating these files are presented in the following section. The purpose of this package is to codify the structure defined in the model. Specifically, connectors, internal queues, and other entities defined in the model are included in the `Instance` type at this level. Furthermore, this package will contain an abstract subprogram for every *invoker* connector that a component defines. These are unimplemented subprograms that MUST be implemented by an inheriting package or a compilation error will result. This feature ensures that any connectors defined in the model are actually implemented in the code, otherwise the code will not compile.

The final level in the package hierarchy is the hand-coded implementation package `Component.Example_Component.Implementation`. This package includes a type `Instance` that inherits from the `Instance` type defined in the generated base package. At this level “templates” can be autocoded to assist the developer in beginning to implement the component. Directions for generating implementation package templates can be found in Section 6.1.4. In this package, the programmer is responsible for hand-coding the behavioral logic of the component. Specifically, this means implementing the abstract *invoker* subprograms defined in the base package, encoding the actions that the component should take whenever these connectors are invoked from

the outside.

The hierarchy is constructed in this way for a few primary reasons. First, it allows the vast majority of boilerplate code that you would typically find in a framework to live within the autogenerated base packages. This avoids the common copy and paste errors that boilerplate code commonly introduces, and also keeps the implementation code readable. Secondly, the use of abstract subprograms in the generated base package ensures that the implementation code is always consistent with the model that describes it. This ensures that all other products generated from the model, documentation, ground system interfaces, etc. are also consistent with the form and structure of the component. After modeling, the task left to the programmer becomes simple: hand-code the behavioral logic that gets executed when a component's connectors are called.

The following sections walk through the steps needed to generate the component base package, generate the implementation package templates, and compile a component to make sure it is syntactically correct.

6.1.3 Component Base Package

In this section we take a look at the component base package. Usually you will not have to build this package manually. Since this package is a dependency of the hand-coded implementation package, it will get compiled automatically when you build your hand-code. However, it is a good idea to at least take a look at the base package once, to understand what kind of things it contains. We explore it in this section, and explain some of the details.

To generate the base package we can run `redo` to compile it which will generate the autocode and then build it:

```
> redo build/obj/Linux/component-example_component.o
```

This should generate and compile two files `build/src/component-example_component.ads` and `build/src/component-example_component.adb`. Let's take a look the specification.

component-example_component.ads:

```
1 -----  
2 -- Example_Component Component Base Spec  
3 --  
4 -- Generated from example_component.component.yaml on 2025-07-15 21:45.  
5 -----  
6  
7 -- Includes:  
8 with Connector_Types;  
9 use Connector_Types;  
10 with Tick;  
11 with Common_Connectors;  
12 with Sys_Time;  
13 with Packet;  
14  
15 -- This is the example component.  
16 package Component.Example_Component is  
17  
18     -- Base class instance:  
19     type Base_Instance is abstract new Component.Core_Instance with private;  
20     type Base_Class_Access is access all Base_Instance'Class;  
21  
22 -----  
23     -- Initialize and finalize heap variables:  
24 -----  
25     -- This procedure allocates memory for the component base class including the
```

```

26   -- internal queue and/or arrayed connectors.
27   --
28   -- Init Base Parameters:
29   -- Packet_T_Send_Count : Connector_Count_Type - The size of the Packet_T_Send
30   -- invoker connector array.
31   --
32   not overriding procedure Init_Base (Self : in out Base_Instance;
33     => Packet_T_Send_Count : in Connector_Count_Type);
34   not overriding procedure Final_Base (Self : in out Base_Instance);
35   -----
36   -- Invokee connector primitives:
37   -----
38   -- This connector provides the schedule tick for the component.
39   -- Define connector package:
40   package Tick_T_Recv_Sync_Connector renames
41     => Common_Connectors.Tick_T_In_Connector;
42   -- Define index type for invokee connector.
43   subtype Tick_T_Recv_Sync_Index is Connector_Index_Type range
44     => Connector_Index_Type'First .. Connector_Index_Type'First;
45   -- Abstract connector procedure to be overridden by child:
46   not overriding procedure Tick_T_Recv_Sync (Self : in out Base_Instance; Arg :
47     => in Tick.T) is abstract;
48   -- Function which returns the hook for this connector. Used when attaching
49   -- this connector to an invoker connector:
50   not overriding function Tick_T_Recv_Sync_Access (Self : in Base_Instance;
51     => Index : in Connector_Index_Type := Connector_Index_Type'First) return not
52     => null Tick_T_Recv_Sync_Connector.Invokee_Hook
53       with Inline => True;
54   -----
55   -- Invoker connector primitives:
56   -----
57   -- This connector is used to fetch the current system time.
58   -- Define connector package:
59   package Sys_Time_T_Get_Connector renames
60     => Common_Connectors.Sys_Time_T_Return_Connector;
61   -- Function to attach this invoker connector to an invokee connector:
62   not overriding procedure Attach_Sys_Time_T_Get (Self : in out Base_Instance;
63     => To_Component : in not null Component.Class_Access; Hook : in not null
64     => Sys_Time_T_Get_Connector.Invokee_Hook; To_Index : in Connector_Index_Type
65     => := Connector_Index_Type'First)
66       with Inline => True;
67   -- This connector is used to send out a telemetry packet.
68   -- Define connector package:
69   package Packet_T_Send_Connector renames
70     => Common_Connectors.Packet_T_In_Connector;
71   -- Function to attach this invoker connector to an invokee connector:
72   subtype Packet_T_Send_Index is Connector_Index_Type;
73   type Packet_T_Send_Array is array (Packet_T_Send_Index range <>) of
74     => Packet_T_Send_Connector.Instance;
75   type Packet_T_Send_Array_Access is access Packet_T_Send_Array;
76   not overriding procedure Attach_Packet_T_Send (Self : in out Base_Instance;
77     => From_Index : in Packet_T_Send_Index; To_Component : in not null
78     => Component.Class_Access; Hook : in not null
79     => Packet_T_Send_Connector.Invokee_Hook; To_Index : in Connector_Index_Type
80     => := Connector_Index_Type'First)
81       with Inline => True;
82   -- This abstract procedure must be overridden in the child package to specify
83   -- the behavior when a Packet_T_Send message is dropped due to a full queue.

```

```

69      not overriding procedure Packet_T_Send_Dropped (Self : in out Base_Instance;
70        Index : in Packet_T_Send_Index; Arg : in Packet.T) is abstract;
71
72  private
73    -----
74    -- Definition of cycle function for task execution:
75    -----
76    -- Passive component queue implementation for cycle.
77    -- This method is implemented, but if called will throw an assertion.
78    overriding procedure Cycle (Self : in out Base_Instance);
79
80    -----
81    -- Private invokee connector hooks which
82    -- dispatch invokee calls to the correct
83    -- abstract function defined in the
84    -- child package:
85
86    -- This connector provides the schedule tick for the component.
87    function Tick_T_Recv_Sync_Hook (Class_Self : in out
88      Component.Core_Instance'Class; Arg : in Tick.T; Index : in
89      Connector_Index_Type := Connector_Index_Type'First; Full_Queue_Behavior :
90      in Full_Queue_Action := Drop) return Connector_Types.Connector_Status;
91
92    -----
93    -- Private invoker connector functions
94    -- for use in the child package:
95
96    -- This connector is used to fetch the current system time.
97    not overriding function Sys_Time_T_Get (Self : in Base_Instance) return
98      Sys_Time.T
99        with Inline => True;
100   not overriding function Is_Sys_Time_T_Get_Connected (Self : in Base_Instance)
101     return Boolean
102       with Inline => True;
103       -- This connector is used to send out a telemetry packet.
104   not overriding procedure Packet_T_Send_If_Connected (Self : in out
105     Base_Instance; Index : in Packet_T_Send_Index; Arg : in Packet.T;
106     Full_Queue_Behavior : in Full_Queue_Action := Drop);
107   not overriding procedure Packet_T_Send (Self : in out Base_Instance; Index :
108     in Packet_T_Send_Index; Arg : in Packet.T; Full_Queue_Behavior : in
109     Full_Queue_Action := Drop)
110     with Inline => True;
111   not overriding function Is_Packet_T_Send_Connected (Self : in Base_Instance;
112     Index : in Packet_T_Send_Index) return Boolean
113       with Inline => True;
114
115    -----
116    -- The base class instance record:
117
118    type Base_Instance is abstract new Component.Core_Instance with record
119      -- Invoker connector objects:
120      Connector_Sys_Time_T_Get : Sys_Time_T_Get_Connector.Instance;
121      Connector_Packet_T_Send : Packet_T_Send_Array_Access := null;
122    end record;
123
124  end Component.Example_Component;

```

The first thing to recognize is the type `Instance` which extends the type `Component.Instance`, declared in the framework's core `Component` package. The details of the `Instance` type are private, and thus listed in the private section. Also, note that the type is declared as `abstract` since it contains some abstract subprograms, ie. primitives that are unimplemented. This means

that `Instance` cannot be instantiated or used by itself, since it is not a complete type. We must inherit from this type and implement the abstract subprograms before we can instantiate an object that uses this code. We will see how that is done in the next section.

In the public section there is also a very important abstract procedure `Tick_T_Recv_Sync` which is the handler for our `Tick_T_Recv_Sync` invokee connector. Note that it has the exact same name as the connector itself, so is easy to remember. This procedure MUST be implemented in the hand-coded implementation package, otherwise a compilation error will result. We will hand-code the behavior of this invokee connector handler in the following section.

If we look in the private section we can see that the type `Instance` is implemented as a record that contains two items. Both items are connector objects for the component's invoker connectors. A component will always contain connector objects for its invoker connectors in its base package. To make a call to these connectors there are two private, inlined procedures that are autocoded, `Packet_T_Send` and `Sys_Time_T_Get`. These subprograms are important because they are meant to be called from the implementation package in order to call those connectors. The names of the procedures exactly match the connector names defined in the model, so they should be easy to remember when implementing your component.

Note that the declaration of `Packet_T_Send` also includes an `index` argument that is used to specify which connector in the connector array to call. Argument `index` is of type `Packet_T_Send_Index` which you can see is defined as a subtype of `Positive` and thus will have a valid range from 1 to the size of the connector array. The actual size of the connector array is defined during runtime using the `Init_Base` procedure. For more details on component base package initialization see Section 6.7.2.

For each invoker connector there are also a few more autocoded methods that may be useful. Let's use the connector `Packet_T_Send` connector as an example, and explain all the autocoded subprograms related to it:

- **`Packet_T_Send`** - Used to call the invoker connector. If the invoker connector is not connected to another component, then this call will result in a runtime assertion.
- **`Is_Packet_T_Send_Connected`** - Used to check if the connector is connected to another component's connector during runtime.
- **`Packet_T_Send_If_Connected`** - Used to call the invoker connector, only if it is connected.

These are the most important subprograms to keep in mind while hand-coding the implementation class, since they are used to call the invoker connectors. In general it is best to use the `Packet_T_Send_If_Connected` procedure when the connection is viewed as *optional* in the assembly. If the component must absolutely be connected on that connector to function correctly, then use the standard `Packet_T_Send` procedure.

Note that `Packet_T_Send` and `Packet_T_Send_If_Connected` call a *send*-kinded connector, and thus provide a special option `Full_Queue_Behavior`. This option allows the call to specify the behavior when the connector is connected to an `recv_async` connector of another component and the component's queue is full. The options are `Drop` (the default if unspecified), in which case the data is dropped, or `Wait`, in which case the call blocks and waits for the queue to no longer be full before proceeding. You should always use the option `Drop` unless you have a very good reason for using `Wait`. This keeps the execution of your code deterministic and predictable and forces the proper sizing of queues.

The remaining package definitions, not described in detail in the paragraphs above, can for the most part be ignored unless you are interested in understanding the inner workings. The purpose of these definitions is supply the structure necessary to make connectors work within the architecture. You can also see the implementation of these subprograms in the body `build/src/component-example_component.adb`, which is not shown here for brevity.

6.1.4 Component Implementation Package

Since we dived into the details of the base package in the previous section, we are now ready to write our own hand-code which extends it. After creating your component model, the steps below are often the next commands an Adamant developer runs. These commands autogenerate the implementation package templates from the component model and then copy them into the same directory as the component model.

```
> redo build/template/component-example_component-implementation.ads  
> redo build/template/component-example_component-implementation.adb  
> cp build/template/* . # copy templates into component dir
```

You can also use the special `redo templates` command to do the same thing:

```
> redo templates  
> cp build/template/* . # copy the template files into component dir
```

As a general convention, you should always store your hand-coded implementation package in the same directory as the component model. Never work on the files directly in `build/template` since that directory will get removed when running `redo clean`. OK, let's take a look at our shiny new templates - specification file first.

component-example_component-implementation.ads:

```
1 -----  
2 -- Example_Component Component Implementation Spec  
3 -----  
4  
5 -- Includes:  
6 with Tick;  
7  
8 -- This is the example component.  
9 package Component.Example_Component.Implementation is  
10  
11     -- The component class instance record:  
12     type Instance is new Example_Component.Base_Instance with private;  
13  
14     private  
15  
16         -- The component class instance record:  
17         type Instance is new Example_Component.Base_Instance with record  
18             null; -- TODO  
19         end record;  
20  
21 -----  
22 -- Set Up Procedure  
23 -----  
24 -- Null method which can be implemented to provide some component  
25 -- set up code. This method is generally called by the assembly  
26 -- main.adb after all component initialization and tasks have been started.  
27 -- Some activities need to only be run once at startup, but cannot be run  
28 -- safely until everything is up and running, ie. command registration,  
29     -- initial  
30     -- data product updates. This procedure should be implemented to do these  
31     -- things  
32     -- if necessary.  
33     overriding procedure Set_Up (Self : in out Instance) is null;  
34 -----
```

```

34  -- Invokee connector primitives:
35  -----
36  -- This connector provides the schedule tick for the component.
37  overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in
38  → Tick.T);
39  -----
40  -- Invoker connector primitives:
41  -----
42  -- This procedure is called when a Packet_T_Send message is dropped due to a
43  → full queue.
44  overriding procedure Packet_T_Send_Dropped (Self : in out Instance; Index : 
45  → in Packet_T_Send_Index; Arg : in Packet.T) is null;
end Component.Example_Component.Implementation;

```

The first thing to notice here is the type `Instance` which inherits from the base package type `Example_Component.Instance` that was presented in the previous section. If we look at the implementation of the type in the private section we can see that it is an empty record with the comment `TODO`. This is where we can add any necessary component state information including other objects, variables, etc.

Next, in the private section, there are a few subprograms of importance. First the `Set_Up` function will not be discussed here; see Section 6.7.6 for details.

Following, there is the procedure called `Tick_T_Recv_Sync`. This procedure overrides the abstract `Tick_T_Recv_Sync` procedure declared in the base package in the previous section. This procedure acts as the handler function that gets called whenever the `Tick_T_Recv_Sync` connector is called from an outside component. We will need to defined the behavior for this procedure in the package body.

Lastly, there is a `Packet_T_Send_Dropped` procedure whose implementation is set to `null`, which means that when the procedure is called it does nothing. This procedure is automatically called whenever a call to `Packet_T_Send` gets dropped, due to a full queue, by a receiving component. By default, this procedure is implemented as `null` and the sending component performs no action. The standard Adamant convention is that the receiving component MUST implement the correct behavior to perform when something is dropped off a queue. The sender does not share that responsibility. However, this method can be implemented should the programmer deem it necessary. There are more details on dropped asynchronous connector calls in Section 6.3.

Now we are ready to take a look at the generated template for the package body.

`component-example_component-implementation.adb:`

```

1  -----
2  -- Example_Component Component Implementation Body
3  -----
4
5  package body Component.Example_Component.Implementation is
6
7  -----
8  -- Invokee connector primitives:
9  -----
10 -- This connector provides the schedule tick for the component.
11 overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in
12 → Tick.T) is
13   -- TODO declarations
14 begin
15   null; -- TODO statements

```

```

15    end Tick_T_Recv_Sync;
16
17 end Component.Example_Component.Implementation;
```

Simple, right?! The `Tick_T_Recv_Sync` procedure gets called whenever the `Tick_T_Recv_Sync` connector is invoked by an outside component. All we have to do now is hand-code what we want the component to do when this happens.

Note that the component implementation templates should always successfully compile. You can compile the implementation object by running:

```
> redo build/obj/Linux/component-example_component-implementation.o
```

or you can run `redo all` to compile every object that can be compiled for a component, including the implementation.

Now it is time to hand-code some behavior into the component. When the `Tick_T_Recv_Sync` connector gets called we want the component to send out a packet to every component connected to it. The packet will be properly timestamped, and will contain a single byte of data that increments ever time `Tick_T_Recv_Sync` is called. The first step is to add this counter to our component record, so we can keep track of its value during program execution. Our newly implemented specification looks like:

component-example_component-implementation.ads:

```

1 -----
2 -- Example_Component Component Implementation Spec
3 -----
4
5 -- Includes:
6 with Tick;
7 with Basic_Types; use Basic_Types;
8
9 -- This is the example component.
10 package Component.Example_Component.Implementation is
11
12   -- The component class instance record:
13   type Instance is new Example_Component.Base_Instance with private;
14
15   private
16
17   -- The component class instance record:
18   type Instance is new Example_Component.Base_Instance with record
19     -- Define the counter byte:
20     Counter : Byte := 0;
21   end record;
22
23 -----
24   -- Set Up Procedure
25 -----
26   -- Null method which can be implemented to provide some component
27   -- set up code. This method is generally called by the assembly
28   -- main.adb after all component initialization and tasks have been started.
29   -- Some activities need to only be run once at startup, but cannot be run
30   -- safely until everything is up and running, ie. command registration,
31   -- initial
32   -- data product updates. This procedure should be implemented to do these
33   -- things
34   -- if necessary.
```

```

33 overriding procedure Set_Up (Self : in out Instance) is null;
34
35 -----
36 -- Invokee connector primitives:
37 -----
38 -- This connector provides the schedule tick for the component.
39 overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in
40   => Tick.T);
41
42 -----
43 -- Invoker connector primitives:
44 -----
45 -- This procedure is called when a Packet_T_Send message is dropped due to a
46 -- full queue.
47 overriding procedure Packet_T_Send_Dropped (Self : in out Instance; Index :
48   => in Packet_T_Send_Index; Arg : in Packet.T) is null;
49
50 end Component.Example_Component.Implementation;

```

The only modification that we made was to add the variable counter to the component Instance record. We initialize the variable to zero. Now let's take a look at the body:

component-example_component-implementation.adb:

```

1 -----
2 -- Example_Component Component Implementation Body
3 -----
4
5 with Interfaces;
6
7 package body Component.Example_Component.Implementation is
8
9 -----
10 -- Invokee connector primitives:
11 -----
12 -- This connector provides the schedule tick for the component.
13 overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in
14   => Tick.T) is
15   use Interfaces;
16
17   -- Ignore tick, since we don't actually use it for anything:
18   Ignore : Tick.T renames Arg;
19
20   -- Create a packet:
21   Pkt : Packet.T := (
22     Header => (
23       Time => Self.Sys_Time_T_Get,
24       Id => 0,
25       Sequence_Count => 0,
26       Buffer_Length => 1           -- We are using 1 byte of data.
27     ),
28     Buffer => [others => 0]
29   );
30 begin
31   -- Set the packet data:
32   Pkt.Buffer (Pkt.Buffer'First) := Self.Counter;
33
34   -- Send the packet to every connected connector:
35   for Idx in Self.Connector_Packet_T_Send.all'Range loop
36     Self.Packet_T_Send_If_Connected (Idx, Pkt);
37   end loop;

```

```

37      -- Increment counter:
38      Self.Counter := @ + 1;
39
40  end Tick_T_Recv_Sync;
41
42 end Component.Example_Component.Implementation;

```

We have added the behavioral logic that gets called when the `Tick_T_Recv_Sync` connector is invoked. First we declare a variable `pkt` of type `Packet.T`. You can find the definition for the `Packet.T` packed record in `src/types/packet/`. We define all parts of the packet including the Header and the Buffer. Note that we set the timestamp of the packet by directly calling our invoker connector to fetch time `Self.Sys_Time_T_Get`. Recall that this subprogram is implemented in the base package that was discussed in the previous section. The `Id` and `Sequence_Count` are set to zero, and the `Buffer_Length` is set to 1, because we are creating a packet where only the first byte of the Buffer is used. Finally, we initialize the entire Buffer byte array to zeros.

In the procedure body, the first step is to set the first byte of the packet data buffer, `Buffer`, to our current counter value, `Self.Counter`. Next, we iterate through every connector in the `Self.Connector_Packet_T_Send` array that was declared in the base package, and then call each `Packet_T_Send` connector in turn, sending them each a version of the packet we created. If any of these connectors is not connected to an external component, then nothing will happen, since we used the subprogram `Packet_T_Send_If_Connected` to call the `Packet_T_Send` connector. Finally, we increment our counter, `Self.Counter`, so that it is one greater the next time we create a packet.

In the following section we will unit test this component to make sure that the logic was implemented properly.

6.2 Unit Testing a Passive Component

Unit testing a component is similar to unit testing a package, see Section 4.3. Reading that section first will make understanding section much easier.

Unit testing a component is different than unit testing a package in that we want to be able to accurately simulate the execution context of a component during deployment. More specifically, we want to be able to simulate all the components that could be connected to this component in a final assembly. However, to keep things simple, Adamant achieves this simulation through a single "reciprocal" tester component, which is attached to the component during unit test. The tester component is unique in that it contains the exact opposite connectors as the component, providing a corresponding connector for each of the component's connectors. In this way, the tester can attach to the component and holistically simulate the component's environment. Below is a diagram that demonstrates how a tester component connects to the `example_component` during unit testing.

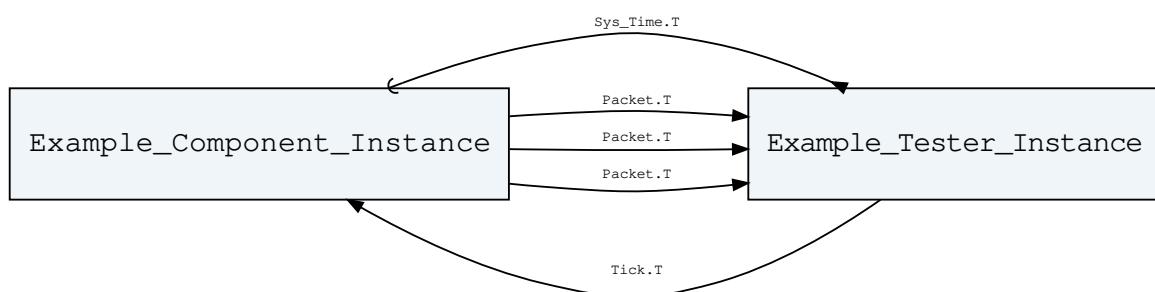


Figure 6: A diagram showing how the `example_component` is connected to its "reciprocal" tester component during unit testing.

As can be seen, there are connections for receiving `Tick.T` and `Sys_Time.T` and an array of 3

`Packet.T` connections from the component to the tester. The tester component itself is completely autocoded by Adamant using the component's YAML model file. The steps for creating the tester are described in following sections.

6.2.1 Anatomy of a Component Unit Test

Before getting into the actual mechanics of unit testing, it is important to understand all of the various Ada packages involved during unit testing. The vast majority of these packages are autogenerated by Adamant, however it is important to understand their function in order to use them properly. The diagram below is the same as Figure 5, except we have added the additional packages necessary for unit testing.

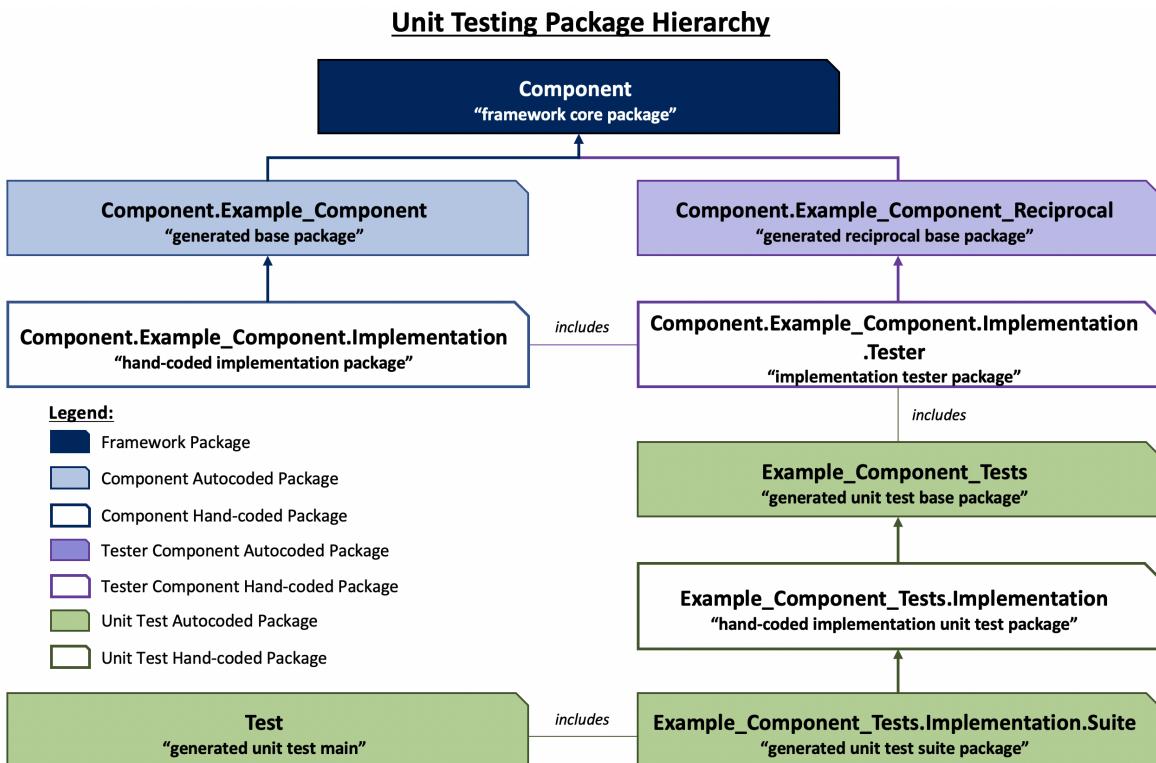


Figure 7: This diagram shows the different Ada packages used to unit test a component.

It is important to note that two YAML models are used to generate the packages shown above, the component model, `example_component.component.yaml`, and a unit test model, `example_component.tests.yaml`. Creating the unit test model is shown in the next section.

The packages in this diagram colored blue make up a component in Adamant, as discussed in Section 6.1.2. The `Component` package is provided by Adamant, the `Component.Example_Component` package is autogenerated from the component model, and the `Component.Example_Component.Implementation` package contains the hand-coded behavior of the component.

An analogous structure is defined for the tester component (shown in violet). The tester component is indeed a component, and so shares the Adamant provided `Component` package as a parent. The component model is used to generate the tester component base package `Component.Example_Component.Reciprocal`. Finally, an autogenerated `Component.Example_Component.Implementation.Tester` package implements the tester

component's behavioral code. This package provides the simulated environment used during unit testing the component. As will be discussed in the next section, the autocoded version of this package can usually be used unmodified to unit test a component effectively. However, the package can also be modified to add extra simulation capability and allow in depth "white-box" testing, discussed in Section 6.2.6.

Important to realize is that the implementation tester package, `Component.Example_Component.Implementation.Tester`, is a child package of the component implementation package, `Component.Example_Component.Implementation`, which gives it access to the private variables and subprograms within the component. Also shown, the tester package *includes* the component within itself, which allows it to directly access and manipulate the component object during test. This setup aids in "white-box" testing, discussed in Section 6.2.6.

Note, both the blue and violet packages are both derived from the component model. The green packages are generated from the unit test model.

The unit test package structure is complicated, but it is designed specifically to interface with the AUnit unit test framework that is used under the hood by Adamant. Three of the four packages are autogenerated directly from the unit test model, and in most cases never need to be looked at or modified by the user. The most important package is the `Example_Component_Tests.Implementation` package, in which the hand-coded unit tests are written by the developer. The autogenerated `Example_Component_Tests` package includes abstract procedures to ensure that all the unit tests defined in the unit test model are indeed implemented in the `Example_Component_Tests.Implementation` package. The autogenerated `Example_Component_Tests.Implementation.Implementation.Suite` package provides the hooks that allow AUnit to run each unit test. Finally, the boilerplate Test package provides the main program that actually runs the unit test.

Note that the unit test package hierarchy *includes* the tester component as a member variable, and thus can directly control the tester component. In this way, the unit test can make calls directly to the tester, enabling it to send the component under test data through connectors. The tester component then receives any outgoing connector calls from the component under test, collecting the information it receives. The unit test can then check the details of this collected information, ensuring that it looks as expected. This procedure is the primary mechanism used for unit testing a component.

While the unit testing package diagram looks complex, of all the packages shown only two need to be modified to successfully build and unit test a component, the component implementation package, `Component.Example_Component.Implementation`, and the unit test implementation package, `Example_Component_Tests.Implementation`. Sometimes it is also advantageous to provide a more sophisticated unit testing environment by augmenting the autogenerated tester component implementation package, `Component.Example_Component.Implementation.Tester`, but this is usually not necessary.

The following sections walk through the steps necessary to create a unit test using the package layout we just explored.

6.2.2 Creating a Unit Test Model

In this section we will create a unit test model to assist in unit testing the *example_component*. First we create a unit test directory called *test/* under our component directory:

```
> mkdir test # Make test/ within example_component/
> cd test
```

Next, we are going to create an Adamant component unit test model file. The procedure here is almost the same as the steps taken to unit test an standard Ada package, presented in Section 4.3.2.

One difference is in how we name the unit test model file. Component unit test model files should always be of the form *specific_name.component_name.tests.yaml* where *component_name* describes the component you are testing, and must be spelled exactly like the component name. *specific_name* is optional and allows you to create a specific name for your unit test suite. This feature is useful when you want to use more than one unit test suite to test your component. We will simply name our model *example_component.tests.yaml*. The contents are shown below:

```

1  ---
2  # Optional - description of unit test suite
3  description: This is a set of unit tests for the Example Component.
4  # Required - list of tests
5  tests:
6      # Required - name of test
7      - name: Test_That_Should_Pass
8          # Optional - description of test
9          description: This test should pass.
10         - name: Test_That_Should_Fail
11             description: This test should not pass.

```

Similar to Section 4.3.2, we define a unit test model with two tests, one that we expect to pass, and the other which we expect to fail, for demonstration purposes.

OK, before we start unit testing, there is one more detail to take care of. Because we are using the AUnit library, we need to make sure our *build target* is set to `Linux_Test`, as the `Linux` target does not link in AUnit by default. We can run the command `export TARGET=Linux_Test`, but then we would have to remember to do this every time we want to run the test. A better approach is to create an *env.py* that does this for us. Adamant uses the same *env.py* for almost all unit tests, so you can easily copy it from another unit test directory. Here are the standard contents of *env.py*, which is created in the *test* directory.

```

1  from environments import test  # noqa: F401

```

If all of that *env.py* discussion is news to you, check out Section 9.10.3 for more details.

The presence of the *example_component.tests.yaml* file in the test directory will allow Adamant to generate many new build rules for unit testing (some output removed for brevity).

```

> redo what
redo  what
redo all
redo clean
redo clean_all
redo templates
redo publish
redo targets
redo prove
redo analyze
redo style
redo pretty
redo test_all
redo analyze_all
redo coverage_all
redo build/bin/Linux_Test/test.elf
redo build/html/example_component_tests.html
redo build/obj/Linux_Test/component-example_component-implementation-tester.o
redo build/obj/Linux_Test/component-example_component_reciprocal.o
redo build/obj/Linux_Test/example_component_tests-implementation-suite.o

```

```

redo build/obj/Linux_Test/example_component_tests-implementation.o
redo build/obj/Linux_Test/example_component_tests.o
redo build/obj/Linux_Test/test.o
redo build/pdf/example_component_tests.pdf
redo build/src/component-example_component_reciprocal.adb
redo build/src/component-example_component_reciprocal.ads
redo build/src/example_component_tests-implementation-suite.adb
redo build/src/example_component_tests-implementation-suite.ads
redo build/src/example_component_tests.adb
redo build/src/example_component_tests.ads
redo build/template/component-example_component-implementation-tester.adb
redo build/template/component-example_component-implementation-tester.ads
redo build/template/example_component_tests-implementation.adb
redo build/template/example_component_tests-implementation.ads
redo build/template/test.adb

```

We can map the files above to their package names in the package diagram shown in Section 6.2.1. Below is the map of package names to the location of their autogenerated source:

- **Component.Example_Component.Reciprocal** -
build/src/component-example_component_reciprocal.ads and
build/src/component-example_component_reciprocal.adb
- **Component.Example_Component.Implementation.Tester** -
build/template/component-example_component-implementation-tester.ads
and
build/template/component-example_component-implementation-tester.adb
- **Example_Component_Tests** -
build/src/example_component_tests.ads and
build/src/example_component_tests.adb
- **Example_Component_Tests.Implementation** -
build/template/example_component_tests-implementation.ads and
build/template/example_component_tests-implementation.adb
- **Example_Component_Tests.Implementation.Suite** -
build/src/example_component_tests-implementation-suite.ads and
build/src/example_component_tests-implementation-suite.adb
- **Test** - *build/template/test.adb*

As usual, the source code generated in *build/src/* is not meant to be modified, and the source code generated in *build/template/* is meant to be copied out and modified as necessary. We perform the usual steps of building the template files and copying them into the *test* directory.

```

> redo build/template/component-example_component-implementation-tester.ads
> redo build/template/component-example_component-implementation-tester.adb
> redo build/template/example_component_tests-implementation.ads
> redo build/template/example_component_tests-implementation.adb
> redo build/template/test.adb
> cp build/template/* . # copy the template files into test/

```

After this step, we can actually compile and run the unit tests. Since the unit tests are not actually implemented yet, we expect them to fail.

```
> redo test
```

which should produce the output:

```
OK Test_That_Should_Pass

FAIL Test_That_Should_Fail
    Assertion:
      3
    !=
      3
  failed.
  at example_component_tests-implementation.adb:88

Total Tests Run: 2
Successful Tests: 1
Failed Assertions: 1
Unexpected Errors: 0
```

As expected, both of our tests fail because they are unimplemented. The next sections describe how we can effectively implement unit tests for the *example_component*.

6.2.3 The “Reciprocal” Tester Component

Before we actually begin writing the unit test code, let’s take a look at the tester component. This component acts as a “reciprocal” in that it contains the exact opposite connections as the component. This allows the tester to completely simulate the environment around the component. Besides the interface, the component also contains some autocoded test *histories* and helper functions that greatly assist the unit test process. We will explore these facilities in this section and show how to use them in the next section.

As was shown in the previous section, the tester component has an analogous package layout to a standard component. For the *example_component*, the following packages make up the tester:

- **Component.Example_Component.Reciprocal** -
build/src/component-example_component_reciprocal.ads and
build/src/component-example_component_reciprocal.adb
- **Component.Example_Component.Implementation.Tester** -
build/template/component-example_component-implementation-tester.ads and
build/template/component-example_component-implementation-tester.adb

The first is the tester base package, and the second is the tester implementation package. All of these packages can easily be generated via *redo* to see their contents. The *Component.Example_Component.Reciprocal* package is very similar (but opposite) to the component’s own base package *Component.Example_Component*, which was described in detail in Section 6.1.3. The one difference is that everything in the *Component.Example_Component.Reciprocal* package is public. There is no private section. This package is meant for unit testing only, so the privatization features only stand to make unit testing more difficult, so they are not used in the same way for the tester as they are for the component *Component.Example_Component* package. The *Component.Example_Component.Reciprocal* will not be discussed further here, but the reader can build and look at it to see how it is constructed.

As eluded to earlier, the tester implementation package, *Component.Example_Component.Implementation.Tester*, provides the behavior of the tester component. The autocoded template provided Adamant is often sufficient enough to allow thorough unit testing of the component,

however functionality can be added to this package by hand to provide a more sophisticated unit test environment. One such example is to augment the tester implementation package for “white”-box testing, which is discussed in Section 6.2.6.

We generated the templates for the Component.Example_Component.Implementation.Tester package and copied them to the *test/* directory in the previous section. Let’s take a look at the specification for that package now to get an idea of what is included.

component-example_component-implementation-tester.ads

```

1  -----
2  -- Example_Component Component Tester Spec
3  -----
4
5  -- Includes:
6  with Component.Example_Component_Reciprocal;
7  with Printable_History;
8  with Sys_Time.Representation;
9  with Packet.Representation;
10
11 -- This is the example component.
12 package Component.Example_Component.Implementation.Tester is
13
14  use Component.Example_Component_Reciprocal;
15  -- Invoker connector history packages:
16  package Sys_Time_T_Return_History_Package is new Printable_History
17    (Sys_Time.T, Sys_Time.Representation.Image);
18  package Packet_T_Recv_Sync_History_Package is new Printable_History
19    (Packet.T, Packet.Representation.Image);
20
21  -- Component class instance:
22  type Instance is new Component.Example_Component_Reciprocal.Base_Instance
23    with record
24      -- The component instance under test:
25      Component_Instance : aliased
26        Component.Example_Component.Implementation.Instance;
27      -- Connector histories:
28      Sys_Time_T_Return_History : Sys_Time_T_Return_History_Package.Instance;
29      Packet_T_Recv_Sync_History : Packet_T_Recv_Sync_History_Package.Instance;
30    end record;
31  type Instance_Access is access all Instance;
32
33  -----
34  -- Initialize component heap variables:
35  -----
36  procedure Init_Base (Self : in out Instance; Packet_T_Send_Count : in
37    Connector_Count_Type);
38  procedure Final_Base (Self : in out Instance);
39
40  -----
41  -- Test initialization functions:
42  -----
43  procedure Connect (Self : in out Instance);
44
45  -----
46  -- Invokee connector primitives:
47  -----
48  -- This connector is used to fetch the current system time.
49  overriding function Sys_Time_T_Return (Self : in out Instance) return
50    Sys_Time.T;
51  -- This connector is used to send out a telemetry packet.

```

```

46    overriding procedure Packet_T_Recv_Sync (Self : in out Instance; Arg : in
47        Packet.T);
48
end Component.Example_Component.Implementation.Tester;

```

The first thing to take notice here is that the whole package is public. There is no privatized information. This means that we can freely call and access any variables in this package from within our unit test code.

The tester component `Instance` object inherits directly from the tester base package `Component.Example_Component_Reciprocal.Instance`. This means that the tester implementation package must implement any abstract subprograms defined in the base package, as was discussed for the *example_component* in Section 6.1.4. Included in the `Instance` record are a few important definitions:

- **component_Instance** - This is the actual component under test. In this case it is an instantiation of the *example_component* component, `Component.Example_Component.Implementation.Instance`. Note that the tester implementation package `Component.Example_Component.Implementation.Tester` is a child package of the component package `Component.Example_Component.Implementation` so the tester can access all of the private subprograms and record entities within the component directly.
- **system_Time** - This is used to simulate the system time during unit test. The value specified here will be returned to the *example_component* when the `Sys_Time_T_Get` connector is invoked.
- **Sys_Time_T_Return_History** and **Packet_T_Recv_Sync_History** - These are *history* objects for the component's `Sys_Time_T_Get` and `Packet_T_Recv_Sync` connectors. The tester implementation package will always define histories to collect anything that be sent out by a component. This includes creating histories for all of the component's *invoker* connectors, two in this case.

The *history* objects allow the tester to store the data sent along a connector each time it is received. In this way, the unit test code can query the histories to see how many times a connector was called and check the specific data items that were sent. This will be demonstrated in the following section. The source code for Adamant unit test histories can be found in `src/unit_test/history/`.

Below the `Instance` definition there are some subprograms. The `Init_Base` and `Final_Base` procedures are used to initialize the component and tester component base packages. In particular, anything that needs the heap is allocated and freed in these subprograms such as arrayed connectors and internal queues. The `Connect` procedure connects the tester component's connectors to the component's connectors. All three of these functions should be called in the unit test `Set_Up_Test` fixture, so that they run prior to every unit test.

The final procedures, `Sys_Time_T_Return` and `Packet_T_Recv_Sync`, are the connector handler procedures that get called when the *example_component* invokes its `Sys_Time_T_Get` and `Packet_T_Send` connectors, respectively. The implementation of these subprograms is to add any data that they receive to the tester component's *histories*. The implementation file is shown below:

component-example_component-implementation-tester.adb

```

1 -----
2 -- Example_Component Component Tester Body
3 -----
4
5 package body Component.Example_Component.Implementation.Tester is
6

```

```

7 -----  

8 -- Initialize heap variables:  

9 -----  

10 procedure Init_Base (Self : in out Instance; Packet_T_Send_Count : in  

11   & Connector_Count_Type) is  

12 begin  

13   -- Initialize component heap:  

14   Self.Component_Instance.Init_Base (Packet_T_Send_Count =>  

15     & Packet_T_Send_Count);  

16  

17   -- Initialize tester heap:  

18   -- Connector histories:  

19   Self.Sys_Time_T_Return_History.Init (Depth => 100);  

20   Self.Packet_T_Recv_Sync_History.Init (Depth => 100);  

21 end Init_Base;  

22  

23 procedure Final_Base (Self : in out Instance) is  

24 begin  

25   -- Destroy tester heap:  

26   -- Connector histories:  

27   Self.Sys_Time_T_Return_History.Destroy;  

28   Self.Packet_T_Recv_Sync_History.Destroy;  

29  

30   -- Destroy component heap:  

31   Self.Component_Instance.Final_Base;  

32 end Final_Base;  

33  

34 -----  

35 -- Test initialization functions:  

36 -----  

37 procedure Connect (Self : in out Instance) is  

38 begin  

39   Self.Component_Instance.Attach_Sys_Time_T_Get (To_Component =>  

40     & Self'Unchecked_Access, Hook => Self.Sys_Time_T_Return_Access);  

41   Self.Component_Instance.Attach_Packet_T_Send (From_Index => 1,  

42     & To_Component => Self'Unchecked_Access, Hook =>  

43     & Self.Packet_T_Recv_Sync_Access);  

44   Self.Component_Instance.Attach_Packet_T_Send (From_Index => 2,  

45     & To_Component => Self'Unchecked_Access, Hook =>  

46     & Self.Packet_T_Recv_Sync_Access);  

47   Self.Component_Instance.Attach_Packet_T_Send (From_Index => 3,  

48     & To_Component => Self'Unchecked_Access, Hook =>  

49     & Self.Packet_T_Recv_Sync_Access);  

50   Self.Attach_Tick_T_Send (To_Component =>  

51     & Self.Component_Instance'Unchecked_Access, Hook =>  

52     & Self.Component_Instance.Tick_T_Recv_Sync_Access);  

53 end Connect;  

54  

55 -----  

56 -- Invokee connector primitives:  

57 -----  

58 -- This connector is used to fetch the current system time.  

59 overriding function Sys_Time_T_Return (Self : in out Instance) return  

60   & Sys_Time.T is  

61   -- Return the system time:  

62   To_Return : constant Sys_Time.T := Self.System_Time;  

63 begin  

64   -- Push the argument onto the test history for looking at later:  

65   Self.Sys_Time_T_Return_History.Push (To_Return);  

66   return To_Return;  

67 end Sys_Time_T_Return;

```

```

56      -- This connector is used to send out a telemetry packet.
57      overriding procedure Packet_T_Recv_Sync (Self : in out Instance; Arg : in
58          → Packet.T) is
59      begin
60          -- Push the argument onto the test history for looking at later:
61          Self.Packet_T_Recv_Sync_History.Push (Arg);
62      end Packet_T_Recv_Sync;
63
64  end Component.Example_Component.Implementation.Tester;

```

Note, it is common while testing a component to realize that you need to change the component model in order to add a connector that you forgot, or add an *ided entity* like a *command*, as will be discussed in Section 6.8. In either case, the developer must modify the component YAML model and then rebuild the tester component templates as described in the previous section. If any modifications have been made to the tester component, then the template must be manually merged with the modified version. `diff`, `git diff`, or a more sophisticated merge tool may aid you in this process. If no modifications were made, then directly copying the newly generated template over the current version in the `test/` directory will suffice.

In the following section we use the tester component package within our unit test code to test the *example_component*.

6.2.4 Implementing the Unit Tests

In this section we will fill in the logic of the unit test template code that we generated in Section 6.2.2. We will also utilize the tester component, described in the previous section, to send data to and collect data from the *example_component* during test.

To get started, recall the unit test packages that can be generated from the our unit test model *example_component.tests.yaml*:

- **Example_Component_Tests -**
`build/src/example_component_tests.ads` and
`build/src/example_component_tests.adb`
- **Example_Component_Tests.Implementation -**
`build/template/example_component_tests-implementation.ads` and
`build/template/example_component_tests-implementation.adb`
- **Example_Component_Tests.Implementation.Suite -**
`build/src/example_component_tests-implementation-suite.ads` and
`build/src/example_component_tests-implementation-suite.adb`
- **Test -**`build/template/test.adb`

The `Example_Component_Tests.Implementation.Suite` and `Test` packages provide scaffolding code for interfacing the unit tests with the AUnit testing framework. These packages never need to be modified or even looked at by the developer writing the unit tests. Feel free to generate and explore them if you wish, but they also will not be shown here.

The `Example_Component_Tests` package is the unit test base package. This package is Adamant generated source that contains the abstract subprogram definitions for the unit test procedures that we will write. The purpose of this package is to ensure that a developer implements every unit test by inheriting from this package and writing the unit test code for every test defined in the unit test model. Let's take a quick look at the specification for this package.

example_component_tests.ads

```

1 -----  

2 -- Example_Component Tests Base Spec  

3 --  

4 -- Generated from example_component.tests.yaml on 2025-07-15 21:45.  

5 -----  

6  

7 -- Standard includes:  

8 with AUnit;  

9 with AUnit.Test_Fixtures;  

10 with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;  

11 with Ada.Calendar; use Ada.Calendar;  

12 with File_Logger;  

13 -- Component Tester Include:  

14 with Component.Example_Component.Implementation.Tester;  

15  

16 -- This is a set of unit tests for the Example Component.  

17 package Example_Component_Tests is  

18     -- Test data and state:  

19     type Base_Instance is abstract new AUnit.Test_Fixtures.Test_Fixture with  

20         private;  

21  

22     -- Abstract unit tests:  

23     -- This test should pass.  

24     procedure Test_That_Should_Pass (Self : in out Base_Instance) is abstract;  

25     -- This test should not pass.  

26     procedure Test_That_Should_Fail (Self : in out Base_Instance) is abstract;  

27  

28     -- Fixture functions  

29     procedure Set_Up_Test (Self : in out Base_Instance) is abstract;  

30     procedure Tear_Down_Test (Self : in out Base_Instance) is abstract;  

31  

32     -- Number of tests variable  

33     Num_Tests : constant Positive := 2;  

34  

35     -- Test name list type for the names  

36     type Test_Name_List_Type is array (Natural range 0 .. Num_Tests - 1) of  

37         Unbounded_String;  

38  

39     Test_Name_List : constant Test_Name_List_Type :=  

40         [  

41             0 => To_Unbounded_String ("Test_That_Should_Pass"),  

42             1 => To_Unbounded_String ("Test_That_Should_Fail")  

43         ];  

44  

45     private  

46         -- Logging procedures:  

47         procedure InitLogging (Self : in out Base_Instance; File_Name : in String);  

48         procedure EndLogging (Self : in out Base_Instance; File_Name : in String);  

49         procedure Log (Self : in out Base_Instance; String_To_Log : in String);  

50  

51         -- Fixture procedures:  

52         overriding procedure Set_Up (Self : in out Base_Instance);  

53         overriding procedure Tear_Down (Self : in out Base_Instance);  

54  

55         -- Test data and state:  

56         type Base_Instance is abstract new AUnit.Test_Fixtures.Test_Fixture with  

57             record  

58                 -- Counter to track the test we are currently running so that we can log  

59                 -- correctly.  

60                 Test_Name_Index : Natural := 0;  

61                 -- File for logging

```

```

58     Logger : aliased File_Logger.Instance;
59     -- Time for when the test starts to track the duration of the test
60     Start_Test_Time : Time;
61     -- The tester component:
62     Tester :
63         -- Component.Example_Component.Implementation.Tester.Instance_Access;
64     end record;
end Example_Component_Tests;

```

There are two important things to notice here. First, there are abstract subprograms for the `Test_That_Should_Pass` and `Test_That_Should_Fail` tests that we defined in our `example_component.tests.yaml` model. We will need to implement these in the `Example_Component_Tests.Implementation` package otherwise a compilation error will result. Secondly, inside the unit test `Base_Instance` record there is a definition for `tester` which is of type `Component.Example_Component.Implementation.Tester.Instance_Access`. This is an access type that points to the tester component implementation that we discussed in the previous section. We can see in the body, shown below, that the tester is allocated on the heap prior to each test and destroyed after using the `Set_Up_Test` and `Tear_Down_Test` test fixtures.

`example_component_tests.adb`

```

1  -----
2  -- Example_Component Tests Base Body
3  --
4  -- Generated from example_component.tests.yaml on 2025-07-15 21:45.
5  -----
6
7  -- Includes:
8  with Ada.Command_Line;
9  with Ada.Directories; use Ada.Directories;
10 with Ada.Calendar.Formatting;
11 with String_Util;
12 with Safe_Deallocator;
13
14 package body Example_Component_Tests is
15
16     -----
17     -- Test Logging:
18     -----
19
20     procedure Log (Self : in out Base_Instance; String_To_Log : in String) is
21     begin
22         -- Make sure we have a log file to log to.
23         Self.Logger.Log (String_Util.Trim_Both (String_To_Log));
24     end Log;
25
26     -- Initialize the logging for the log file ensuring the correct directory is
27     -- available
28     procedure Init_Logging (Self : in out Base_Instance; File_Name : in String)
29     -- is
30         use Ada.Calendar.Formatting;
31         -- The path returned by the Command_Name is the path to the test.elf, so
32         -- back out to the build directory
33         Log_Dir : constant String := Containing_Directory (Containing_Directory
34             -- (Containing_Directory (Ada.Command_Line.Command_Name))) & "/log/" &
35             File_Name & ".log";
36     begin
37         -- Save the time to calculate the duration of the test later on
38         Self.Start_Test_Time := Clock;
39         -- Create the log file for the test

```

```

35     Self.Logger.Open (Log_Dir);
36     Self.Log ("    Beginning log for " & File_Name & " at " &
37         => String_Util.Trim_Both (Image (Clock)));
38 end InitLogging;

39
procedure EndLogging (Self : in out Base_Instance; File_Name : in String) is
40     use Ada.Calendar.Formatting;
41     -- Now calculate the duration
42     Test_Duration : constant Duration := (Clock - Self.Start_Test_Time);
43 begin
44     -- Close the log that was used during this test.
45     Self.Log ("    Ending log for " & File_Name & " at " &
46         => String_Util.Trim_Both (Image (Clock)) & " and took " &
47         => String_Util.Trim_Both (Duration'Image (Test_Duration)) & " seconds to
48         => run.");
49     Self.Logger.Close;
50 end EndLogging;

51
52
53 overriding procedure SetUp (Self : in out Base_Instance) is
54     Test_String : constant String := To_String (Test_Name_List
55         => (Self.Test_Name_Index));
56 begin
57     -- Use the helper function to get the name of the test to setup
58     Self.InitLogging (Test_String);
59     -- Log that we are starting to setup
60     Self.Log ("    Starting SetUp for test " & Test_String);
61     -- Dynamically allocate the component tester:
62     Self.Tester := new
63         => Component.Example_Component.Implementation.Tester.Instance;
64     -- Link the log access type to the logger in the reciprocal
65     Self.Tester.Set_Logger (Self.Logger'Unchecked_Access);
66     -- Call up to the implementation setup
67     Base_Instance'Class (Self).SetUp_Test;
68     Self.Log ("    Finishing SetUp for test " & Test_String);
69 end SetUp;

70
71 overriding procedure TearDown (Self : in out Base_Instance) is
72     procedure FreeTester is new Safe_Deallocator.Deallocate_If_Testing (
73         Object => Component.Example_Component.Implementation.Tester.Instance,
74         Name =>
75             => Component.Example_Component.Implementation.Tester.Instance_Access
76     );
77     Closing_Test : constant String := To_String (Test_Name_List
78         => (Self.Test_Name_Index));
79 begin
80     -- Log the tear down
81     Self.Log ("    Starting TearDown for test " & Closing_Test);
82     -- Call up to the implementation for any tear down
83     Base_Instance'Class (Self).TearDown_Test;
84     -- End the logging for the current test
85     Self.Log ("    Finishing TearDown for test " & Closing_Test);
86     Self.EndLogging (Closing_Test);
87     -- Increment counter for the next test name in the list and pass the log
88     -- to close back up to the tear down (or component unit test)
89     Self.Test_Name_Index := @ + 1;
90     -- Delete tester:
91     FreeTester (Self.Tester);

```

```

87     end Tear_Down;
88 end Example_Component_Tests;
```

Recreating the tester component on the heap for each test ensures that the state of the tester component, and the actual component under test contained within, is reset prior to each test. This isolates each unit test from one another, making them easier to write and maintain.

The package we are most concerned with is the unit test implementation package, `Example_Component_Tests.Implementation`, whose files we generated and then copied from the `build/template` directory in Section 6.2.2. It is in these files that we will write our unit test code. Let's start by looking at the autogenerated specification.

example_component_tests-implementation.ads

```

1 -----
2 -- Example_Component Tests Spec
3 -----
4
5 -- This is a set of unit tests for the Example Component.
6 package Example_Component_Tests.Implementation is
7
8     -- Test data and state:
9     type Instance is new Example_Component_Tests.Base_Instance with private;
10    type Class_Access is access all Instance'Class;
11
12    private
13        -- Fixture procedures:
14        overriding procedure Set_Up_Test (Self : in out Instance);
15        overriding procedure Tear_Down_Test (Self : in out Instance);
16
17        -- This test should pass.
18        overriding procedure Test_That_Should_Pass (Self : in out Instance);
19        -- This test should not pass.
20        overriding procedure Test_That_Should_Fail (Self : in out Instance);
21
22        -- Test data and state:
23        type Instance is new Example_Component_Tests.Base_Instance with record
24            null;
25        end record;
26    end Example_Component_Tests.Implementation;
```

First, note the type `Instance` which inherits from `Example_Component_Tests.Base_Instance` which was declared in the unit test base package. This means we can access the tester object that was defined in the base package. Second, we can see definitions for the test fixtures `Set_Up_Test` and `Tear_Down_Test` and our two unit tests `Test_That_Should_Pass` and `Test_That_Should_Fail`. Usually, you will never have to modify the specification, unless you want to add extra state variables to the unit test record type. This is rarely done, but might be useful to track some state between unit tests. Now let's take a look at the body:

example_component_tests-implementation.adb

```

1 -----
2 -- Example_Component Tests Body
3 -----
4
5 with AUnit.Assertions; use AUnit.Assertions;
6
7 package body Example_Component_Tests.Implementation is
8
9 -----
```

```

10  -- Fixtures:
11  -----
12
13  overriding procedure Set_Up_Test (Self : in out Instance) is
14  begin
15      -- Allocate heap memory to component:
16      Self.Tester.Init_Base (Packet_T_Send_Count => 3);
17
18      -- Make necessary connections between tester and component:
19      Self.Tester.Connect;
20
21      -- Call the component set up method that the assembly would normally call.
22      Self.Tester.Component_Instance.Set_Up;
23
24      -- TODO Insert custom set up code here.
25      null;
26  end Set_Up_Test;
27
28  overriding procedure Tear_Down_Test (Self : in out Instance) is
29  begin
30      -- TODO Insert custom cleanup code here.
31      null;
32      -- Free component heap:
33      Self.Tester.Final_Base;
34  end Tear_Down_Test;
35
36  -----
37  -- Tests:
38  -----
39
40  -- This test should pass.
41  overriding procedure Test_That_Should_Pass (Self : in out Instance) is
42      -- TODO declarations
43  begin
44      -- TODO replace the following with actual test code.
45      Assert (False, "Test 'Test_That_Should_Pass' is unimplemented.");
46  end Test_That_Should_Pass;
47
48  -- This test should not pass.
49  overriding procedure Test_That_Should_Fail (Self : in out Instance) is
50      -- TODO declarations
51  begin
52      -- TODO replace the following with actual test code.
53      Assert (False, "Test 'Test_That_Should_Fail' is unimplemented.");
54  end Test_That_Should_Fail;
55
56 end Example_Component_Tests.Implementation;

```

First, look at the fixtures. The `Set_Up_Test` subprogram, which is called by the parent class `Set_Up` procedure, proceeds to call the tester subprograms `Init_Base` and `Connect` which set up the tester component. These subprograms were described in detail in the previous section. The call to `Self.Tester.Component_Instance.Set_Up` calls the component's `Set_Up` method, which is described in Section 6.7.6. The `Tear_Down_Test` procedure simply calls the tester `Final_Base` procedure.

Second are the two skeleton implementations of our unit tests that produce failure messages, as was seen in Section 6.2.2. Now let's get to work filling in those TODOs with actual unit test code. Below is the unit test implementation package body, filled in with actual hand-written unit test code.

example_component_tests-implementation.adb

```

1  -----
2  -- Example_Component Tests Body
3  -----
4
5  with Basic_Assertions; use Basic_Assertions;
6  with Packet.Assertion; use Packet.Assertion;
7
8  package body Example_Component_Tests.Implementation is
9
10   -----
11  -- Fixtures:
12  -----
13
14  overriding procedure Set_Up_Test (Self : in out Instance) is
15  begin
16    -- Allocate heap memory to component:
17    Self.Tester.Init_Base (Packet_T_Send_Count => 3);
18
19    -- Make necessary connections between tester and component:
20    Self.Tester.Connect;
21
22    -- Call the component set up method that the assembly would normally call.
23    Self.Tester.Component_Instance.Set_Up;
24  end Set_Up_Test;
25
26  overriding procedure Tear_Down_Test (Self : in out Instance) is
27  begin
28    -- Free component heap:
29    Self.Tester.Final_Base;
30  end Tear_Down_Test;
31
32  -----
33  -- Tests:
34  -----
35
36  overriding procedure Test_That_Should_Pass (Self : in out Instance) is
37    -- Define the packet we are going to compare against.
38    A_Packet : Packet.T := (
39      Header => (
40        Time => (0, 0),
41        Id => 0,
42        Sequence_Count => 0,
43        Buffer_Length => 1
44      ),
45      Buffer => [others => 0]
46    );
47  begin
48    -- Send the example component a tick:
49    Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
50
51    -- Check the tester component histories. The example component should
52    -- have requested Time one time through the Sys_Time.T connector.
53    Natural Assert.Eq (Self.Tester.Sys_Time_T_Return_History.Get_Count, 1);
54
55    -- The component should have send out 3 packets, one two each of its
56    -- 3 Packet.T connections. All three of these connectors are connected
57    -- to the tester Packet_T_Recv_Sync connector, so we should have gotten
58    -- 3 packets in that history.
59    Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 3);
60
61    -- OK now check the contents of the packets.

```

```

62     Packet_Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (1),
63         => A_Packet);
64     Packet_Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (2),
65         => A_Packet);
66     Packet_Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (3),
67         => A_Packet);
68
69     -- Send another tick:
70     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
71
72     -- Check the tester component histories again.
73     Natural Assert.Eq (Self.Tester.Sys_Time_T_Return_History.Get_Count, 2);
74     Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 6);
75
76     -- OK now check the contents of the packets. Expect the data to have
77     -- incremented this time.
78     A_Packet.Buffer (0) := 1;
79     Packet_Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (4),
80         => A_Packet);
81     Packet_Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (5),
82         => A_Packet);
83     Packet_Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (6),
84         => A_Packet);
85
86 end Test_That_Should_Pass;
87
88 overriding procedure Test_That_Should_Fail (Self : in out Instance) is
89 begin
90     -- Send the example component a tick:
91     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
92
93     -- This is going to fail since we actually received 3 packets and we are
94     -- testing NOT EQUAL (neq) !
95     Natural Assert.Neq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 3);
96
97 end Test_That_Should_Fail;
98
99 end Example_Component_Tests.Implementation;

```

For these tests we use the Adamant provided `Basic_Assertions` package which implements the Adamant `Smart Assert` package for many common Ada types, including `Natural`. The assertions provided by the `Basic_Assertions` package are easier to use than `pragma Assert` and provide much more informative error messages than AUnit's `AUnit.Assertions` package with very little effort. The `Basic_Assertions` and `Smart Assert` packages can both be found in `src/unit_test/smart_assert/`.

In the first test we send the component a `Tick.T` on its `Tick_T_Recv_Sync` connector. We achieve this by invoking the tester component's `Tick_T_Send` connector (which is connected to the component in `Set_Up`). Once the `Tick.T` is sent to the *example_component* we expect it to emit timestamped packets to each of its connected `Packet.T` connections. First we check to make sure one call to `Sys_Time_T_Get` was made. We check this by asking the tester component's `Sys_Time_T_Return_History` how many requests it has received (`get_Count`). In this case we want to make sure 1 request was received for time.

Next we make sure that 3 packets were emitted from the component. Note that we expect 3 packets because we instantiated the *example_component* with 3 `Packet_T_Send` connectors in the `Set_Up_Test` fixture via the call to `Self.Tester.Init_Base`, which will call the component's `Init_Base` procedure. By default, each of these 3 arrayed connectors are connected to a single connector on the tester component, in this case `Packet_Recv_Sync`. We make sure that 3 were emitted by checking the `Packet_Recv_Sync_History.get_Count`.

Now that we know that 3 packets were emitted, we check each individual packet that was sent

out against a packet that we declared locally. To check the whole packet we simply use the `Packet.Assertion` package, which is included with all *packed types*, see Section 5.1.6. For the first `Tick.T` that we send, we expect the whole packet to be initialized to zero, except the packet's length which should be 1.

Next, we repeat the test again by sending another `Tick.T` on the tester component's `Tick_T_Send` connector. The only difference this time is that we expect the emitted packets to have an incremented data value. So, we modify the comparison packet's first data byte to be equal to 1 and then run the same checks as before.

The purpose of the second test is to demonstrate what a failed test looks like. In this case, we simply send a `Tick.T` on the tester component's `Tick_T_Send` connector and then check to make sure that some amount other than 3 packets was emitted. Since we know from the previous test that 3 packets will be sent out, this test should fail. The output of running these unit tests can be seen by running:

```
> redo test
```

which prints the output:

```
OK Test_That_Should_Pass

FAIL Test_That_Should_Fail
    Assertion:
        3
    !=
        3
failed.
    at example_component_tests-implementation.adb:88

Total Tests Run:    2
Successful Tests:  1
Failed Assertions: 1
Unexpected Errors: 0
```

As can be seen from the output, AUnit prints a nice summary of the two tests, telling us that the first test passed and the second test failed. Using the `Basic_ASSERTIONS` produces a very nice error message for our failed test telling us that the check of `3 /= 3` (`3` not equal to `3`) failed on a specific line of code.

If we modify the second test so that it passes, we get the lovely output:

```
OK Test_That_Should_Pass
OK Test_That_Should_Fail

Total Tests Run:    2
Successful Tests:  2
Failed Assertions: 0
Unexpected Errors: 0
```

Note, it is common while developing tests to write the source code for few unit tests only to realize later that you need some more unit tests defined to fully test your component. In this case, the standard procedure is to update the model file, ie. `example_component.tests.yaml`, with your new test definitions, rebuild the templates as seen in Section 6.2.2, and then manually merge the templates

with the unit test code that you have been writing. `diff`, `git diff`, or a more sophisticated merge tool may aid you in this process. Know that if you make a mistake while merging, Adamant will catch most issues, like forgetting to implement a test, through modeling errors and compilation errors.

This section demonstrated how to “black-box” test the *example_component* by invoking its connectors and checking the output packets. The next section will introduce the concept of “white-box” testing, which allows us to probe the internals of the component during unit test.

6.2.5 Component Unit Test Logger

Just like with the package unit testing, Section 4.3.3, there is also a logging capability as part of the component unit test system. The logger will automatically log each call of any connector and the data passed along the connector. The user has the ability to turn on and off each individual connector log statement, which will be discussed more later. Logs are always written to the *build/log* directory. A new log file is created for each user-defined unit test.

Using the same *example_component* and unit tests from before, user defined log statements have now been added to the test.

example_component_tests-implementation.adb

```
1 -----  
2 -- Example_Component Tests Body  
3 -----  
4  
5 with Basic_Assertions; use Basic_Assertions;  
6 with Packet.Assertion; use Packet.Assertion;  
7  
8 package body Example_Component_Tests.Implementation is  
9  
10 -----  
11 -- Fixtures:  
12 -----  
13  
14 overriding procedure Set_Up_Test (Self : in out Instance) is  
15 begin  
16     -- Allocate heap memory to component:  
17     Self.Tester.Init_Base (Packet_T_Send_Count => 3);  
18  
19     -- Make necessary connections between tester and component:  
20     Self.Tester.Connect;  
21  
22     -- Call the component set up method that the assembly would normally call.  
23     Self.Tester.Component_Instance.Set_Up;  
24 end Set_Up_Test;  
25  
26 overriding procedure Tear_Down_Test (Self : in out Instance) is  
27 begin  
28     -- Free component heap:  
29     Self.Tester.Final_Base;  
30 end Tear_Down_Test;  
31  
32 -----  
33 -- Tests:  
34 -----  
35  
36 overriding procedure Test_That_Should_Pass (Self : in out Instance) is  
37     -- Define the packet we are going to compare against.  
38     A_Packet : Packet.T := (  
39         Header => (
```

```

40             Time => (0, 0),
41             Id => 0,
42             Sequence_Count => 0,
43             Buffer_Length => 1
44         ),
45         Buffer => [others => 0]
46     );
47 begin
48     -- Send the example component a tick:
49     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
50
51     -- Check the tester component histories. The example component should
52     -- have requested Time one time through the Sys_Time.T connector.
53     Natural Assert.Eq (Self.Tester.Sys_Time_T_Return_History.Get_Count, 1);
54
55     -- The component should have send out 3 packets, one two each of its
56     -- 3 Packet.T connections. All three of these connectors are connected
57     -- to the tester Packet_T_Recv_Sync connector, so we should have gotten
58     -- 3 packets in that history.
59     Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 3);
60
61     -- OK now check the contents of the packets.
62     Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (1),
63         => A_Packet);
64     Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (2),
65         => A_Packet);
66     Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (3),
67         => A_Packet);
68
69     -- Send another tick:
70     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
71
72     -- Check the tester component histories again.
73     Natural Assert.Eq (Self.Tester.Sys_Time_T_Return_History.Get_Count, 2);
74     Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 6);
75
76     -- OK now check the contents of the packets. Expect the data to have
77     -- incremented this time.
78     A_Packet.Buffer (0) := 1;
79     Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (4),
80         => A_Packet);
81     Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (5),
82         => A_Packet);
83     Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (6),
84         => A_Packet);
85 end Test_That_Should_Pass;

86 overriding procedure Test_That_Should_Fail (Self : in out Instance) is
87 begin
88     -- Send the example component a tick:
89     Self.Log ("Sending a Tick with Log 1");
90     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
91
92     -- Make sure the internal counter inside the component is now 1:
93     Byte Assert.Eq (Self.Tester.Get_Component_Counter, 1);
94
95     -- Send the example component a tick:
96     Self.Tester.Log ("Sending a Tick with Log 2");
97     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
98
99     -- Make sure the internal counter inside the component is now 2:

```

```

95     Byte Assert.Eq (Self.Tester.Get_Component_Counter, 2);
96
97     -- Send the example component a tick:
98     Self.Tester.Log ("    Sending a Tick with spaces and Log 2");
99     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
100
101    -- Make sure the internal counter inside the component is now 3:
102    Byte Assert.Eq (Self.Tester.Get_Component_Counter, 3);
103  end Test_That_Should_Fail;
104
105 end Example_Component_Tests.Implementation;

```

Log statements are in the form <timestamp> <simulated_system_time> <direction> <message> where <timestamp> is wall clock Unix seconds and <simulated_system_time> is the simulated in-test time as denoted by `Self.Tester.System_Time` in Unix seconds. The <direction> denotes the call direction of the connector invocations where <-> denotes the start of a call from the tester component to the component under test, -> denotes the return of a call from the tester to the component under test, and -> denotes calls from the component under test to the tester component.

With a component unit test, the user has the choice of two log procedures. Using `Self.Log` will log like the package, where there is no <simulated_system_time>, and if `Self.Tester.Log` is used, the log will contain the computer timestamp and the simulated system time in the message. Both are used to give an example and is shown in the output log file.

```

build/log/Test_That_Should_Fail.log
1752615974.361185000 Beginning log for Test_That_Should_Fail at 2025-07-15 21:46:14
1752615974.361189000 Starting Set_Up for test Test_That_Should_Fail
1752615974.361199000 0.000000000 -> Sys_Time_T_Return returned at index 1 and address 0
1752615974.361200000 0.000000000 -> Packet_T_Recv_Sync returned at index 1 and address 0
1752615974.361200000 0.000000000 -> Packet_T_Recv_Sync returned at index 1 and address 0
1752615974.361201000 0.000000000 -> Packet_T_Recv_Sync returned at index 1 and address 0
1752615974.361202000 0.000000000 <- Tick_T_Send attached to index 1 and address 00000000
1752615974.361202000 Finishing Set_Up for test Test_That_Should_Fail
1752615974.361203000 Sending a Tick with Log 1
1752615974.361204000 0.000000000 <- Tick_T_Recv_Sync : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361205000 0.000000000 -> Sys_Time_T_Get : (Seconds => 0, Subseconds => 0)
1752615974.361206000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361207000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361208000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361208000 0.000000000 -- Tick_T_Recv_Sync finished.
1752615974.361210000 0.000000000 Sending a Tick with Log 2
1752615974.361210000 0.000000000 <- Tick_T_Recv_Sync : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361211000 0.000000000 -> Sys_Time_T_Get : (Seconds => 0, Subseconds => 0)
1752615974.361212000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361213000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361214000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361215000 0.000000000 -- Tick_T_Recv_Sync finished.
1752615974.361215000 0.000000000      Sending a Tick with spaces and Log 2
1752615974.361216000 0.000000000 <- Tick_T_Recv_Sync : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361217000 0.000000000 -> Sys_Time_T_Get : (Seconds => 0, Subseconds => 0)
1752615974.361218000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361219000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361220000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361220000 0.000000000 -- Tick_T_Recv_Sync finished.
1752615974.361221000 Starting Tear_Down for test Test_That_Should_Fail
1752615974.361221000 Finishing Tear_Down for test Test_That_Should_Fail
1752615974.361222000 Ending log for Test_That_Should_Fail at 2025-07-15 21:46:14 and terminating process

```

And the passing test has none of our injected log messages, but there is still default logging occurring from all of the interfaces being used during the test.

build/log/Test_That_Should_Fail.log

```
1752615974.361042000 Beginning log for Test_That_Should_Pass at 2025-07-15 21:46:14
1752615974.361050000 Starting Set_Up for test Test_That_Should_Pass
1752615974.361071000 0.000000000 -> Sys_Time_T_Return returned at index 1 and address 0
1752615974.361072000 0.000000000 -> Packet_T_Recv_Sync returned at index 1 and address 0
1752615974.361073000 0.000000000 -> Packet_T_Recv_Sync returned at index 1 and address 0
1752615974.361074000 0.000000000 -> Packet_T_Recv_Sync returned at index 1 and address 0
1752615974.361074000 0.000000000 <- Tick_T_Send attached to index 1 and address 00000000
1752615974.361075000 Finishing Set_Up for test Test_That_Should_Pass
1752615974.361077000 0.000000000 <- Tick_T_Recv_Sync : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361078000 0.000000000 -> Sys_Time_T_Get : (Seconds => 0, Subseconds => 0)
1752615974.361080000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361082000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361083000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361083000 0.000000000 -- Tick_T_Recv_Sync finished.
1752615974.361114000 0.000000000 <- Tick_T_Recv_Sync : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361115000 0.000000000 -> Sys_Time_T_Get : (Seconds => 0, Subseconds => 0)
1752615974.361116000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361117000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361118000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds => 0))
1752615974.361118000 0.000000000 -- Tick_T_Recv_Sync finished.
1752615974.3611141000 Starting Tear_Down for test Test_That_Should_Pass
1752615974.3611142000 Finishing Tear_Down for test Test_That_Should_Pass
1752615974.3611143000 Ending log for Test_That_Should_Pass at 2025-07-15 21:46:14 and total time 0.000000000
```

Finally, the user has the ability to turn on and off each connector call log using member boolean variables inside the tester. The naming is setup such that the user only needs to know the name of the connector they want to turn on or off. The log enable booleans are in the form *Log_<Connector_Name>* and ided entities have the form *Log_<Ided_Entity_Name>*. The following example turns off all connector data except for all packet data in the test fixture *Set_Up_Test*.

```
1 -----  
2 -- Example_Component Tests Body  
3 -----  
4  
5 with Basic_Assertions; use Basic_Assertions;  
6 with Packet.Assertion; use Packet.Assertion;  
7  
8 package body Example_Component_Tests.Implementation is  
9  
10 -----  
11 -- Fixtures:  
12 -----  
13  
14 overriding procedure Set_Up_Test (Self : in out Instance) is  
15 begin  
16     -- Setup flags to log only packets  
17     Self.Tester.Log_Sys_Time_T_Get := False;  
18     Self.Tester.Log_Tick_T_Recv_Sync := False;  
19     -- Allocate heap memory to component:  
20     Self.Tester.Init_Base (Packet_T_Send_Count => 3);  
21  
22     -- Make necessary connections between tester and component:  
23     Self.Tester.Connect;  
24
```

```

25      -- Call the component set up method that the assembly would normally call.
26      Self.Tester.Component_Instance.Set_Up;
27  end Set_Up_Test;
28
29  overriding procedure Tear_Down_Test (Self : in out Instance) is
30  begin
31      -- Free component heap:
32      Self.Tester.Final_Base;
33  end Tear_Down_Test;
34
35  -----
36  -- Tests:
37  -----
38
39  overriding procedure Test_That_Should_Pass (Self : in out Instance) is
40      -- Define the packet we are going to compare against.
41      A_Packet : Packet.T := (
42          Header => (
43              Time => (0, 0),
44              Id => 0,
45              Sequence_Count => 0,
46              Buffer_Length => 1
47          ),
48          Buffer => [others => 0]
49      );
50  begin
51      -- Send the example component a tick:
52      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
53
54      -- Check the tester component histories. The example component should
55      -- have requested Time one time through the Sys_Time.T connector.
56      Natural Assert.Eq (Self.Tester.Sys_Time_T_Return_History.Get_Count, 1);
57
58      -- The component should have send out 3 packets, one two each of its
59      -- 3 Packet.T connections. All three of these connectors are connected
60      -- to the tester Packet_T_Recv_Sync connector, so we should have gotten
61      -- 3 packets in that history.
62      Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 3);
63
64      -- OK now check the contents of the packets.
65      Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (1),
66          => A_Packet);
66      Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (2),
67          => A_Packet);
67      Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (3),
68          => A_Packet);
68
69      -- Send another tick:
70      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
71
72      -- Check the tester component histories again.
73      Natural Assert.Eq (Self.Tester.Sys_Time_T_Return_History.Get_Count, 2);
74      Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 6);
75
76      -- OK now check the contents of the packets. Expect the data to have
77      -- incremented this time.
78      A_Packet.Buffer (0) := 1;
79      Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (4),
80          => A_Packet);
80      Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (5),
81          => A_Packet);

```

```

81     Packet_Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (6),
82                         => A_Packet);
83 end Test_That_Should_Pass;
84
85 overriding procedure Test_That_Should_Fail (Self : in out Instance) is
86 begin
87     -- Setup flags to log only packets
88     Self.Tester.Log_Sys_Time_T_Get := False;
89     Self.Tester.Log_Tick_T_Recv_Sync := False;
90     -- Send the example component a tick:
91     Self.Log ("Sending a Tick with Log 1");
92     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
93
94     -- Make sure the internal counter inside the component is now 1:
95     Byte Assert.Eq (Self.Tester.Get_Component_Counter, 1);
96
97     -- Send the example component a tick:
98     Self.Tester.Log ("Sending a Tick with Log 2");
99     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
100
101    -- Make sure the internal counter inside the component is now 2:
102    Byte Assert.Eq (Self.Tester.Get_Component_Counter, 2);
103
104    -- Send the example component a tick:
105    Self.Tester.Log ("    Sending a Tick with spaces and Log 2");
106    Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
107
108    -- Make sure the internal counter inside the component is now 3:
109    Byte Assert.Eq (Self.Tester.Get_Component_Counter, 3);
110 end Test_That_Should_Fail;
111
112 end Example_Component_Tests.Implementation;

```

And the output is now missing any time ticks or calls where the system time is retrieved.

build/log/Test_That_Should_Fail.log

```

1752615980.829236000 Beginning log for Test_That_Should_Fail at 2025-07-15 21:46:20
1752615980.829239000 Starting Set_Up for test Test_That_Should_Fail
1752615980.829249000 0.000000000 -> Packet_T_Recv_Sync returned at index 1 and address
1752615980.829250000 0.000000000 -> Packet_T_Recv_Sync returned at index 1 and address
1752615980.829250000 0.000000000 -> Packet_T_Recv_Sync returned at index 1 and address
1752615980.829251000 Finishing Set_Up for test Test_That_Should_Fail
1752615980.829251000 Sending a Tick with Log 1
1752615980.829253000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds
1752615980.829254000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds
1752615980.829255000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds
1752615980.829257000 0.000000000 Sending a Tick with Log 2
1752615980.829258000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds
1752615980.829259000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds
1752615980.829260000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds
1752615980.829261000 0.000000000      Sending a Tick with spaces and Log 2
1752615980.829262000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds
1752615980.829262000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds
1752615980.829263000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds
1752615980.829264000 Starting Tear_Down for test Test_That_Should_Fail
1752615980.829265000 Finishing Tear_Down for test Test_That_Should_Fail
1752615980.829266000 Ending log for Test_That_Should_Fail at 2025-07-15 21:46:20 and to

```

```
build/log/Test_That_Should_Fail.log
1752615980.829092000 Beginning log for Test_That_Should_Pass at 2025-07-15 21:46:20
1752615980.829100000 Starting Set_Up for test Test_That_Should_Pass
1752615980.829124000 0.000000000 -> Packet_T_Recv_Sync returned at index 1 and address
1752615980.829125000 0.000000000 -> Packet_T_Recv_Sync returned at index 1 and address
1752615980.829126000 0.000000000 -> Packet_T_Recv_Sync returned at index 1 and address
1752615980.829126000 Finishing Set_Up for test Test_That_Should_Pass
1752615980.829130000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds
1752615980.829132000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds
1752615980.829133000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds
1752615980.829167000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds
1752615980.829168000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds
1752615980.829169000 0.000000000 -> Packet_T_Send : (Time => (Seconds => 0, Subseconds
1752615980.829192000 Starting Tear_Down for test Test_That_Should_Pass
1752615980.829194000 Finishing Tear_Down for test Test_That_Should_Pass
1752615980.829194000 Ending log for Test_That_Should_Pass at 2025-07-15 21:46:20 and to
```

6.2.6 “White-box” Unit Testing

In this section we will “white-box” test the *example_component*, by checking the internal state of the component during the test, not just the connector outputs. Specifically, we want to make sure the internal variable counter defined with the component increments with every call to the Tick_T_Recv_Sync. To do this, we need a way to “see” into the private variables inside of the *example_component*. The mechanism to do this in Ada is via a child package, which has access to the private definitions within its parent package. Luckily, we have already created such a child package for the *example_component*, the tester implementation package. Specifically, this is the Component.Example_Component.Implementation.Tester package which is a child of the component implementation package, Component.Example_Component.Implementation.

We cannot access the Counter variable directly from our unit test code, since it is not a child package of component implementation package. Instead we need to modify the Component.Example_Component.Implementation.Tester to include a public function that provides the unit test code access. To do this we will modify the Component.Example_Component.Implementation.Tester package to include a new function which returns the components internal counter value called Get_Component_Counter. We will have to modify both the specification and the body to do this. The hand-modified tester component body is shown below with the new function:

component-example_component-implementation-tester.adb

```
1 -----
2 -- Example_Component Component Tester Body
3 -----
4
5 package body Component.Example_Component.Implementation.Tester is
6
7 -----
8 -- Initialize heap variables:
9 -----
10 procedure Init_Base (Self : in out Instance; Packet_T_Send_Count : in
11   & Connector_Count_Type) is
12 begin
13   -- Initialize component heap:
14   Self.Component_Instance.Init_Base (Packet_T_Send_Count =>
15   & Packet_T_Send_Count);
16
17   -- Initialize tester heap:
```

```

16      -- Connector histories:
17      Self.Sys_Time_T_Return_History.Init (Depth => 10);
18      Self.Packet_T_Recv_Sync_History.Init (Depth => 10);
19  end Init_Base;
20
21  procedure Final_Base (Self : in out Instance) is
22  begin
23      -- Destroy tester heap:
24      -- Connector histories:
25      Self.Sys_Time_T_Return_History.Destroy;
26      Self.Packet_T_Recv_Sync_History.Destroy;
27
28      -- Destroy component heap:
29      Self.Component_Instance.Final_Base;
30  end Final_Base;
31
32  -----
33  -- Test initialization functions:
34  -----
35  procedure Connect (Self : in out Instance) is
36  begin
37      Self.Component_Instance.Attach_Sys_Time_T_Get (Self'Unchecked_Access,
38          → Self.Sys_Time_T_Return_Access);
39      Self.Component_Instance.Attach_Packet_T_Send (1, Self'Unchecked_Access,
40          → Self.Packet_T_Recv_Sync_Access);
41      Self.Component_Instance.Attach_Packet_T_Send (2, Self'Unchecked_Access,
42          → Self.Packet_T_Recv_Sync_Access);
43      Self.Component_Instance.Attach_Packet_T_Send (3, Self'Unchecked_Access,
44          → Self.Packet_T_Recv_Sync_Access);
45      Self.Attach_Tick_T_Send (Self.Component_Instance'Unchecked_Access,
46          → Self.Component_Instance.Tick_T_Recv_Sync_Access);
47  end Connect;
48
49  -----
50  -- Invokee connector primitives:
51  -----
52  -- This connector is used to fetch the current system time.
53  overriding function Sys_Time_T_Return (Self : in out Instance) return
54      → Sys_Time.T is
55      To_Return : Sys_Time.T;
56  begin
57      -- Return the system time:
58      To_Return := Self.System_Time;
59      -- Push the argument onto the test history for looking at later:
60      Self.Sys_Time_T_Return_History.Push (To_Return);
61      return To_Return;
62  end Sys_Time_T_Return;
63
64  -- This connector is used to send out a telemetry packet.
65  overriding procedure Packet_T_Recv_Sync (Self : in out Instance; Arg : in
66      → Packet.T) is
67  begin
68      -- Push the argument onto the test history for looking at later:
69      Self.Packet_T_Recv_Sync_History.Push (Arg);
70  end Packet_T_Recv_Sync;
71
72  -----
73  -- Custom functions for white-box testing
74  -----
75  not overriding function Get_Component_Counter (Self : in Instance) return
76      → Byte is

```

```

69 begin
70     return Self.Component_Instance.Counter;
71 end Get_Component_Counter;
72
73 end Component.Example_Component.Implementation.Tester;

```

The new function simply returns `Self.Component_Instance.Counter` since it has access to the component's internal state. Now we can update our unit tests to use this new feature. Below is the new unit test implementation body. We left the first test alone, but modified the second test to invoke the `Tick_T_Send` connector, and make sure the component's internal `Counter` variable increments with each call.

`example_component_tests-implementation.adb`

```

1 -----
2 -- Example_Component Tests Body
3 -----
4
5 with Basic_Assertions; use Basic_Assertions;
6 with Packet.Assertion; use Packet.Assertion;
7
8 package body Example_Component_Tests.Implementation is
9
10    -----
11    -- Fixtures:
12    -----
13
14    overriding procedure Set_Up_Test (Self : in out Instance) is
15 begin
16        -- Allocate heap memory to component:
17        Self.Tester.Init_Base (Packet_T_Send_Count => 3);
18
19        -- Make necessary connections between tester and component:
20        Self.Tester.Connect;
21
22        -- Call the component set up method that the assembly would normally call.
23        Self.Tester.Component_Instance.Set_Up;
24    end Set_Up_Test;
25
26    overriding procedure Tear_Down_Test (Self : in out Instance) is
27 begin
28        -- Free component heap:
29        Self.Tester.Final_Base;
30    end Tear_Down_Test;
31
32    -----
33    -- Tests:
34    -----
35
36    overriding procedure Test_That_Should_Pass (Self : in out Instance) is
37        -- Define the packet we are going to compare against.
38        A_Packet : Packet.T := (
39            Header => (
40                Time => (0, 0),
41                Id => 0,
42                Sequence_Count => 0,
43                Buffer_Length => 1
44            ),
45            Buffer => [others => 0]
46        );
47    begin

```

```

48      -- Send the example component a tick:
49      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
50
51      -- Check the tester component histories. The example component should
52      -- have requested Time one time through the Sys_Time.T connector.
53      Natural Assert.Eq (Self.Tester.Sys_Time_T_Return_History.Get_Count, 1);
54
55      -- The component should have send out 3 packets, one two each of its
56      -- 3 Packet.T connections. All three of these connectors are connected
57      -- to the tester Packet_T_Recv_Sync connector, so we should have gotten
58      -- 3 packets in that history.
59      Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 3);
60
61      -- OK now check the contents of the packets.
62      Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (1),
63          ↪ A_Packet);
63      Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (2),
64          ↪ A_Packet);
64      Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (3),
65          ↪ A_Packet);
66
66      -- Send another tick:
67      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
68
69      -- Check the tester component histories again.
70      Natural Assert.Eq (Self.Tester.Sys_Time_T_Return_History.Get_Count, 2);
71      Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 6);
72
73      -- OK now check the contents of the packets. Expect the data to have
74      -- incremented this time.
75      A_Packet.Buffer (0) := 1;
76      Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (4),
77          ↪ A_Packet);
77      Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (5),
78          ↪ A_Packet);
78      Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (6),
79          ↪ A_Packet);
79 end Test_That_Should_Pass;

80
81 overriding procedure Test_That_Should_Fail (Self : in out Instance) is
82 begin
83     -- Send the example component a tick:
84     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
85
86     -- Make sure the internal counter inside the component is now 1:
87     Byte Assert.Eq (Self.Tester.Get_Component_Counter, 1);
88
89     -- Send the example component a tick:
90     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
91
92     -- Make sure the internal counter inside the component is now 2:
93     Byte Assert.Eq (Self.Tester.Get_Component_Counter, 2);
94
95     -- Send the example component a tick:
96     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
97
98     -- Make sure the internal counter inside the component is now 3:
99     Byte Assert.Eq (Self.Tester.Get_Component_Counter, 3);
100 end Test_That_Should_Fail;
101
102 end Example_Component_Tests.Implementation;

```

Now we can run the tests with the command:

```
> redo test
```

which prints the output:

```
OK Test_That_Should_Pass
OK Test_That_Should_Fail
```

```
Total Tests Run: 2
Successful Tests: 2
Failed Assertions: 0
Unexpected Errors: 0
```

Excellent! Using this simple pattern you can interrogate any internal state of a component during unit testing. This allows for fine grained validation of the component that otherwise would not be possible.

6.2.7 Unit Test Coverage Report

Creating unit test coverage reports for component unit tests is the same as creating coverage reports for ordinary modeled unit tests, see Section 4.3.4. The procedure will be repeated below with the *example_component* unit test for clarity.

It is often desirable to know how much of the implementation code that a unit test actually tests. One metric for quantifying this is through a “coverage” report, which shows which lines of code the unit tests traversed, and which lines of code were not executed. For Adamant, a 100% line coverage metric is required for component unit testing, and justification must be provided if this goal cannot be achieved. Note that Adamant uses the [gcovr](#) tool to generate reports.

Once we have a passing set of unit tests, creating a coverage report is not difficult. From the *test* directory created in the previous sections we can run:

```
> redo coverage # create unit test coverage report
```

which produces the output:

```
(INFO) Reading coverage data...
(INFO) Writing coverage report...
```

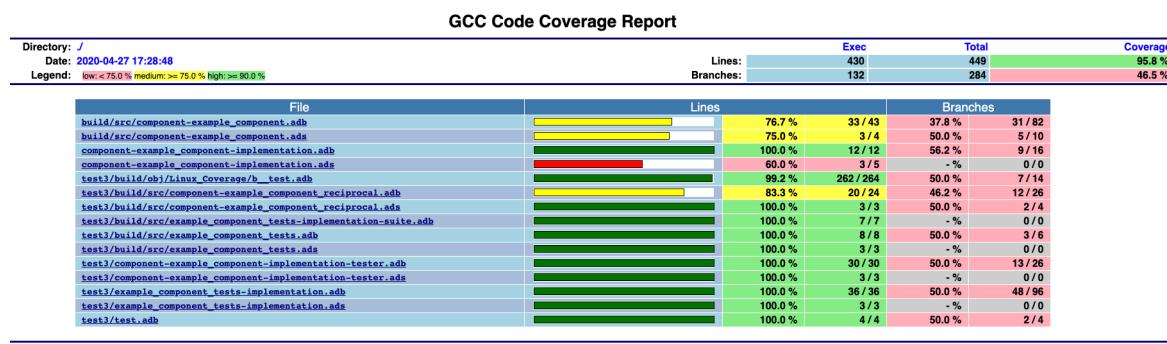
GCC Code Coverage Report				
File	Lines	Exec	Cover	Missing
build/src/component-example_component.adb	46	40	87%	44, 47, 52, 89, 91, 117
build/src/component-example_component.ads	4	4	100%	
component-example_component-implementation.adb	13	13	100%	
component-example_component-implementation.ads	5	3	60%	33, 45

test3/build/obj/Linux_Coverage/b__test.adb	316	314	99%	352,408
test3/build/src/component-example_component_reciprocal.adb	59	50	84%	43-44, 71-72, 88, 128, 130
test3/build/src/component-example_component_reciprocal.ads	3	3	100%	
test3/build/src/example_component_tests-implementation-suite.adb	7	7	100%	
test3/build/src/example_component_tests.adb	34	34	100%	
test3/build/src/example_component_tests.ads	6	6	100%	
test3/component-example_component-implementation-tester.adb	30	30	100%	
test3/component-example_component-implementation-tester.ads	3	3	100%	
test3/example_component_tests-implementation.adb	35	35	100%	
test3/example_component_tests-implementation.ads	3	3	100%	
test3/test.adb	4	4	100%	
<hr/>				
TOTAL	568	549	96%	
<hr/>				

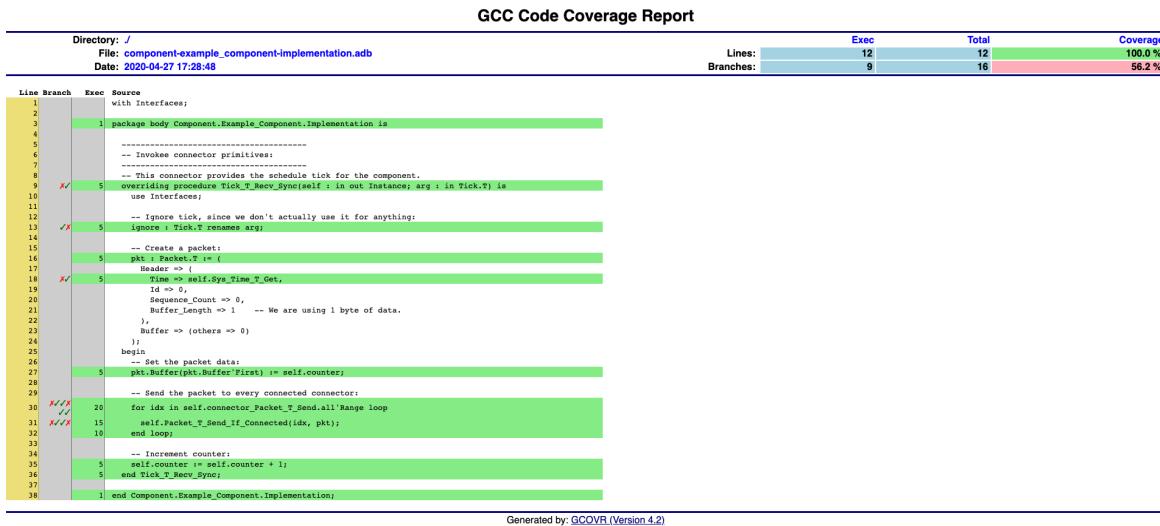
which shows that we have 100% coverage on *component-example_component-implementation.adb*, which is the package we are interested in fully testing. Note, this print out can also be found in the autogenerated text file *build/coverage/coverage.txt*.

In addition to a text file, the `redo coverage` command also creates a user friendly HTML output that shows each files and color-coded lines of code executed. This can be viewed by opening *build/coverage/test.html* in your favorite web browser. Below are a few examples of what the HTML output looks like:

Summary Page:



Specific File Page:



6.2.8 Unit Test Documentation

Creating unit test documentation for component unit tests is the same as creating documentation for ordinary modeled unit tests, see Section 4.3.5. The procedure will be repeated below with the *example_component* unit test for clarity.

Adamant provides automatic generation of documentation for any unit test model, including unit tests for components. Currently two versions of documentation can be created: HTML, which is useful for presentation in meetings or for quick reference, and PDF (via L^AT_EX), which is useful for formal documentation.

To build HTML documentation run:

```
> redo build/html/example_component_tests.html
```

which when opened with your favorite web browser looks something like this:

Example_Component_Tests Unit Test Suite for the Example_Component Component

Description: This is a set of unit tests for the Example Component.

Test Name	Description
Test_That_Should_Pass	This test should pass.
Test_That_Should_Fail	This test should not pass.

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/example_component/test3/example_component.tests.yaml on 2020-04-27 17:28.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

To build the PDF documentation run:

```
> redo build/pdf/example_component_tests.pdf
```

Which produces a PDF output that looks like the following.

This is a set of unit tests for the Example Component.

- **Test_That_Should_Pass** - This test should pass.
 - **Test_That_Should_Fail** - This test should not pass.
-

6.3 Creating a Queued Component

This section shows an example of building and unit testing a component with an asynchronous connector, which requires an internal queue be instantiated within the component. This section will highlight the differences in working with a component of this type compared to working with a simple passive component, presented in Section 6.1. It is expected that you understand the concepts presented in that section before reading this section, as the details presented there will not be repeated here.

6.3.1 The Component Queue

In Adamant, a component is defined by declaring its interface in a component model, which consists of a list of connectors. If any of these connectors are of kind `recv_async` then the component is constructed with an internal queue to store messages for processing at a later time.

In some situations, asynchronous communication might be advantageous. Consider a component whose task is to log data to disk. In order to allow other components to send data to this logger component without waiting for the entire disk write operation to complete (which might be very slow), an asynchronous connection might be used in order to queue the data in memory. In this way, messages sent from components to the logger component will be immediately queued up, allowing those components to continue their normal operation without waiting for the disk write to complete. When the logger executes at some later time (most likely at a lower priority), it will be able to store these messages to disk.

A queued component is any component that has one or more asynchronous invoker connectors. If an Adamant component contains at least one asynchronous invoker connector than a single internal queue is instantiated within the component. This queue is generic and can hold items of many different types. If a component contains multiple asynchronous connectors, all of these items are still enqueued on the same single internal queue. Using a single queue makes the execution logic of a component simple to understand and implement. When a component “pops” an item off the queue, a label is included that tell the component which asynchronous connector the item is associated with. Using the label, the component autocode can then call the correct invokee handler procedure to process the connector call.

The standard Adamant component queue package can be found `src/data_structures/labeled_queue/`. The queue itself is allocated as an array of bytes on the heap. The length of the queue (in bytes) is determined at component initialization and is specified in the assembly model. Each item put on the queue takes up 5 bytes of additional overhead to store the length of the item enqueued and a label which specifies the type of the item, ie. which asynchronous connector the data belongs to. The component queue is implemented as an Ada *protected object*. This means that access to the queue by multiple tasks in the system is safe and free from corruption. Because of this the queue can also be used for task synchronization, as will be shown in Section 6.4.

Note that Adamant also offers a priority queue as an alternative to the standard queue for when dequeuing items in first-in-first-out (FIFO) order is not sufficient. Creating a component with a priority queue is discussed in Section 6.3.4.

In the following section we create a component with an internal queue and demonstrate how it is used.

6.3.2 Implementing a Queued Component

In this section we will still be working with a passive component, that is a component without a thread of execution. This means that the component will still only act on the thread of its connector invokers. In this case, a synchronous connector must be used to service any connector calls that have been queued. This design is ideal for components that need to execute at a very specific time, say on a periodic rate group, but can receive work to do asynchronously.

Let's create a new component called *queued_component* in a new directory.

```
> mkdir queued_component    # make component directory
> cd queued_component
> touch .all_path          # add directory to build path
```

To define the component in this directory, we create a YAML model file *queued_component.component.yaml* with the following contents.

queued_component.component.yaml

```
1  ---
2  execution: passive
3  description: This is the queued component.
4  connectors:
5      #####
6      # Invokee Connectors
7      #####
8      - type: Tick.T
9          kind: recv_sync
10         description: This connector provides the schedule tick for the component.
11     #####
12     # Asynchronous Invokee Connectors
13     #####
14     - type: Packed_Byte.T
15         kind: recv_async
16         description: This connector receives a single byte asynchronously that is
17             ↳ used to populate the outgoing packet.
18     - type: Packed_U16.T
19         kind: recv_async
20         description: This connector receives a 16-bit number asynchronously that is
21             ↳ used to populate the outgoing packet.
22     #####
23     # Invoker Connectors
24     #####
25     - return_type: Sys_Time.T
26         kind: get
27         description: This connector is used to fetch the current system time.
28     - type: Packet.T
29         kind: send
30         description: This connector is used to send out a telemetry packet.
```

This model file generates the following component diagram. See Section 6.1.1 for directions on how to create a component diagram.



Table 7: A queued component which has two asynchronous connectors.

We want this component to be similar to the *example_component* we constructed in the previous sections. It will still produce a timestamped packet every time a `Tick.T` is received. However, this time, the contents of the packet will be set asynchronously by one of two asynchronous connectors. The first asynchronous connector will contain an 8-bit value that will get enqueued, the second a 16-bit value. In either case, when the data is received, the packets produced on the `Tick.T` will hold a copy of the last received data, and the packet length will be set to 1 or 2 bytes accordingly.

First let's take a look at the component base package so that we can see the component's internal queue. The base package can autocoded and compiled with the following command.

```
> redo build/obj/Linux/component-queued_component.o
```

This should generate and compile two files `build/src/component-queued_component.ads` and `build/src/component-queued_component.adb`. Let's take a look the specification.

component-queued_component.ads:

```

1 -----  

2 -- Queued_Component Component Base Spec  

3 --  

4 -- Generated from queued_component.component.yaml on 2025-07-15 21:48.  

5 -----  

6  

7 -- Includes:  

8 with Connector_Types;  

9 use Connector_Types;  

10 with Labeled_Queue;  

11 with Basic_Types;  

12 with Tick;  

13 with Common_Connectors;  

14 with Packed_Byt;  

15 with In_Connector;  

16 with Packed_U16;  

17 with Sys_Time;  

18 with Packet;  

19  

20 -- This is the queued component.  

21 package Component.Queued_Component is  

22  

23     -- Base class instance:  

24     type Base_Instance is abstract new Component.Core_Instance with private;  

25     type Base_Class_Access is access all Base_Instance'Class;  

26  

27     -- The maximum size of an element (in bytes) that can be put onto the queue,  

28     -- including any bytes used for overhead  

29     -- when storing the element. This value can be used to aid in sizing the  

29     -- queue for this component inside the assembly.  

30     Max_Queue_Element_Size : constant Natural;  

30     -- Primitive to return the value above, sometimes this is more convenient to  

30     -- use.

```

```

31  not overriding function Get_Max_Queue_Element_Size (Self : in Base_Instance)
32    => return Natural
33    with Inline => True;
34
35    -- Get the current percent usage of the component's internal queue, if there
36    -- is one, else assert. This function is used by the queue monitor component
37    -- as a backdoor convenience to bypass the connector system, reducing
38    -- complexity
39    -- of reporting queue usage for every component.
40  overriding function Get_Queue_Current_Percent_Used (Self : in out
41    => Base_Instance) return Basic_Types.Byte
42    with Inline => True;
43
44    -- Get the maximum percent usage of the component's internal queue, if there
45    -- is one, else assert. This function is used by the queue monitor component
46    -- as a backdoor convenience to bypass the connector system, reducing
47    -- complexity
48    -- of reporting queue usage for every component. ie. "high water mark"
49  overriding function Get_Queue_Maximum_Percent_Used (Self : in out
50    => Base_Instance) return Basic_Types.Byte
51    with Inline => True;
52
53  -----
54  -- Initialize and finalize heap variables:
55  -----
56  -- This procedure allocates memory for the component base class including the
57  -- internal queue and/or arrayed connectors.
58  --
59  -- Init Base Parameters:
60  -- Queue_Size : Natural - The number of bytes that can be stored in the
61  -- component's internal queue.
62  --
63  not overriding procedure Init_Base (Self : in out Base_Instance; Queue_Size :
64    => in Natural);
65  not overriding procedure Final_Base (Self : in out Base_Instance);
66
67  -----
68  -- Invokee connector primitives:
69  -----
70  -- This connector provides the schedule tick for the component.
71  -- Define connector package:
72  package Tick_T_Recv_Sync_Connector renames
73    Common_Connectors.Tick_T_In_Connector;
74  -- Define index type for invokee connector.
75  subtype Tick_T_Recv_Sync_Index is Connector_Index_Type range
76    Connector_Index_Type'First .. Connector_Index_Type'First;
77  -- Abstract connector procedure to be overridden by child:
78  not overriding procedure Tick_T_Recv_Sync (Self : in out Base_Instance; Arg :
79    => in Tick.T) is abstract;
80  -- Function which returns the hook for this connector. Used when attaching
81  -- this connector to an invoker connector:
82  not overriding function Tick_T_Recv_Sync_Access (Self : in Base_Instance;
83    => Index : in Connector_Index_Type := Connector_Index_Type'First) return not
84    => null Tick_T_Recv_Sync_Connector.Invokee_Hook
85    with Inline => True;
86
87  -- This connector receives a single byte asynchronously that is used to
88  -- populate
89  -- the outgoing packet.
90  -- Define connector package:
91  package Packed_Byt_T_Recv_Async_Connector is new In_Connector
92    => (Packed_Byt.T);

```

```

79   -- Define index type for invokee connector.
80   subtype Packed_Byte_T_Recv_Async_Index is Connector_Index_Type range
81     → Connector_Index_Type'First .. Connector_Index_Type'First;
82   -- Abstract connector procedure to be overridden by child:
83   not overriding procedure Packed_Byte_T_Recv_Async (Self : in out
84     → Base_Instance; Arg : in Packed_Byte.T) is abstract;
85   -- This abstract procedure must be overridden in the child package to specify
86   -- the behavior when a Packed_Byte_T_Recv_Async message is dropped due to a
87   -- full queue.
88   not overriding procedure Packed_Byte_T_Recv_Async_Dropped (Self : in out
89     → Base_Instance; Arg : in Packed_Byte.T) is abstract;
90   -- Function which returns the hook for this connector. Used when attaching
91   -- this connector to an invoker connector:
92   not overriding function Packed_Byte_T_Recv_Async_Access (Self : in
93     → Base_Instance; Index : in Connector_Index_Type :=
94     → Connector_Index_Type'First) return not null
95     → Packed_Byte_T_Recv_Async_Connector.Invokee_Hook
96       with Inline => True;
97
98   -- This connector receives a 16-bit number asynchronously that is used to
99   -- populate
100  -- the outgoing packet.
101  -- Define connector package:
102  package Packed_U16_T_Recv_Async_Connector is new In_Connector (Packed_U16.T);
103  -- Define index type for invokee connector.
104  subtype Packed_U16_T_Recv_Async_Index is Connector_Index_Type range
105    → Connector_Index_Type'First .. Connector_Index_Type'First;
106  -- Abstract connector procedure to be overridden by child:
107  not overriding procedure Packed_U16_T_Recv_Async (Self : in out
108    → Base_Instance; Arg : in Packed_U16.T) is abstract;
109  -- This abstract procedure must be overridden in the child package to specify
110  -- the behavior when a Packed_U16_T_Recv_Async message is dropped due to a
111  -- full queue.
112  not overriding procedure Packed_U16_T_Recv_Async_Dropped (Self : in out
113    → Base_Instance; Arg : in Packed_U16.T) is abstract;
114  -- Function which returns the hook for this connector. Used when attaching
115  -- this connector to an invoker connector:
116  not overriding function Packed_U16_T_Recv_Async_Access (Self : in
117    → Base_Instance; Index : in Connector_Index_Type :=
118    → Connector_Index_Type'First) return not null
119      → Packed_U16_T_Recv_Async_Connector.Invokee_Hook
120        with Inline => True;
121
122  -----
123  -- Invoker connector primitives:
124  -----
125  -- This connector is used to fetch the current system time.
126  -- Define connector package:
127  package Sys_Time_T_Get_Connector renames
128    → Common_Connectors.Sys_Time_T_Return_Connector;
129  -- Function to attach this invoker connector to an invokee connector:
130  not overriding procedure Attach_Sys_Time_T_Get (Self : in out Base_Instance;
131    → To_Component : in not null Component_Class_Access; Hook : in not null
132    → Sys_Time_T_Get_Connector.Invokee_Hook; To_Index : in Connector_Index_Type
133    → := Connector_Index_Type'First)
134      with Inline => True;
135
136  -- This connector is used to send out a telemetry packet.
137  -- Define connector package:
138  package Packet_T_Send_Connector renames
139    → Common_Connectors.Packet_T_In_Connector;

```

```

-- Function to attach this invoker connector to an invokee connector:
not overriding procedure Attach_Packet_T_Send (Self : in out Base_Instance;
    ↵ To_Component : in not null Component_Class_Access; Hook : in not null
    ↵ Packet_T_Send_Connector.Invokee_Hook; To_Index : in Connector_Index_Type
    ↵ := Connector_Index_Type'First)
    with Inline => True;
-- This abstract procedure must be overridden in the child package to specify
-- the behavior when a Packet_T_Send message is dropped due to a full queue.
not overriding procedure Packet_T_Send_Dropped (Self : in out Base_Instance;
    ↵ Arg : in Packet.T) is abstract;

private
-----
-- Types for handling asynchronous messages
-- on the queue:
-----
-- Define the connector identifier enum:
type Connector_Identifier_Enum is (
    Quit,
    Packed_Byte_T_Recv_Async,
    Packed_U16_T_Recv_Async
);
for Connector_Identifier_Enum use (
    Quit => 0,
    Packed_Byte_T_Recv_Async => 1,
    Packed_U16_T_Recv_Async => 2
);
-- Define packed connector identifier record.
type Connector_Identifier_Type is record
    Id : Connector_Identifier_Enum;
end record
with Size => 8,
     Object_Size => 8,
     Value_Size => 8,
     Alignment => 1,
     Volatile => False;

-- Define the size of the record to be compact to save space on
-- the queue (24 bits).
for Connector_Identifier_Type use record
    Id at 0 range 0 .. 7;
end record;

-- Create the queue package for this component:
pragma Warnings (Off, "overlay changes scalar storage order");
package Queue_Package is new Labeled_Queue (Label_Type =>
    ↵ Connector_Identifier_Type);
pragma Warnings (On, "overlay changes scalar storage order");

-- Calculate the maximum message size that will be
-- stored inside of the component queue:
Max_Message_Size : constant Natural :=
    Integer'Max (Packed_Byte.Size_In_Bytes,
    Integer'Max (Packed_U16.Size_In_Bytes,
    0));
-- Declare byte array type of the maximum message size:
subtype Queue_Byte_Array is Basic_Types.Byte_Array (0 .. Max_Message_Size -
    ↵ 1);

```

```

170      -- Resolve public maximum element size as the maximum data message size + the
171      -- queue overhead size (in bytes):
172      Max_Queue_Element_Size : constant Natural := Max_Message_Size +
173          Queue_Package.Element_Storage_Overhead;
174
175      -- Private invokee connector dispatch functions:
176      not overriding procedure Dispatch_Quit (Ignore : in out Base_Instance;
177          Ignore_Bytes : in Basic_Types.Byte_Array);
178      not overriding procedure Dispatch_Packed_Byte_T_Recv_Async (Self : in out
179          Base_Instance; Bytes : in Basic_Types.Byte_Array);
180      not overriding procedure Dispatch_Packed_U16_T_Recv_Async (Self : in out
181          Base_Instance; Bytes : in Basic_Types.Byte_Array);
182
183      -- Procedure lookup table for dispatching to correct connector handler:
184      type Dispatch_Procedure is not null access procedure (Self : in out
185          Base_Instance; Bytes : in Basic_Types.Byte_Array);
186      type Dispatch_Table_T is array (Connector_Identifier_Enum) of
187          Dispatch_Procedure;
188      Dispatch_Table : constant Dispatch_Table_T := [
189          Quit => Dispatch_Quit'Access,
190          Packed_Byte_T_Recv_Async => Dispatch_Packed_Byte_T_Recv_Async'Access,
191          Packed_U16_T_Recv_Async => Dispatch_Packed_U16_T_Recv_Async'Access
192      ];
193
194      -----
195      -- Private methods for a component queue
196
197      -- Function which dispatches up to "n" messages from the queue, or drains the
198      -- queue,
199      -- whichever happens first. It returns the number of messages dispatched:
200      not overriding function Dispatch_N (Self : in out Base_Instance; N : in
201          Natural := 1) return Natural;
202      -- Function which tries to drain the queue. It returns the number of messages
203      -- dispatched:
204      not overriding function Dispatch_All (Self : in out Base_Instance) return
205          Natural;
206      -- Function which waits on queue for message and then dispatches it.
207      procedure Dispatch_Block (Self : in out Base_Instance);
208      -- Function dispatches message from queue if available, otherwise returns
209      -- False.
210      function Dispatch_Nonblock (Self : in out Base_Instance) return Boolean;
211
212      -----
213      -- Private enqueue methods for different types on the queue
214
215      -- Enqueue a specific type for each connector onto the queue.
216      not overriding function Enqueue_Packed_Byte_T_Recv_Async (Self : in out
217          Base_Instance; Arg : in Packed_Byte.T; Full_Queue_Behavior : in
218          Full_Queue_Action := Drop) return Connector_Types.Connector_Status;
219      not overriding function Enqueue_Packed_U16_T_Recv_Async (Self : in out
220          Base_Instance; Arg : in Packed_U16.T; Full_Queue_Behavior : in
221          Full_Queue_Action := Drop) return Connector_Types.Connector_Status;
222
223      -----
224      -- Definition of cycle function for task execution:
225
226      -- Passive component queue implementation for cycle.
227      -- This method is implemented, but if called will throw an assertion.
228      overriding procedure Cycle (Self : in out Base_Instance);
229
230      -----

```

```

215  -- Private invokee connector hooks which
216  -- dispatch invokee calls to the correct
217  -- abstract function defined in the
218  -- child package:
219  -----
220  -- This connector provides the schedule tick for the component.
221  function Tick_T_Recv_Sync_Hook (Class_Self : in out
222    -- Component.Core_Instance'Class; Arg : in Tick.T; Index : in
223    -- Connector_Index_Type := Connector_Index_Type'First; Full_Queue_Behavior :
224    -- in Full_Queue_Action := Drop) return Connector_Types.Connector_Status;
225  -- This connector receives a single byte asynchronously that is used to
226  -- populate
227  -- the outgoing packet.
228  function Packed_Byt_T_Recv_Async_Hook (Class_Self : in out
229    -- Component.Core_Instance'Class; Arg : in Packed_Byt.T; Index : in
230    -- Connector_Index_Type := Connector_Index_Type'First; Full_Queue_Behavior :
231    -- in Full_Queue_Action := Drop) return Connector_Types.Connector_Status;
232  -- This connector receives a 16-bit number asynchronously that is used to
233  -- populate
234  -- the outgoing packet.
235  function Packed_U16_T_Recv_Async_Hook (Class_Self : in out
236    -- Component.Core_Instance'Class; Arg : in Packed_U16.T; Index : in
237    -- Connector_Index_Type := Connector_Index_Type'First; Full_Queue_Behavior :
238    -- in Full_Queue_Action := Drop) return Connector_Types.Connector_Status;
239  -----
240  -- Private invoker connector functions
241  -- for use in the child package:
242  -----
243  -- This connector is used to fetch the current system time.
244  not overriding function Sys_Time_T_Get (Self : in Base_Instance) return
245    -- Sys_Time.T
246    with Inline => True;
247  not overriding function Is_Sys_Time_T_Get_Connected (Self : in Base_Instance)
248    -- return Boolean
249    with Inline => True;
250  -- This connector is used to send out a telemetry packet.
251  not overriding procedure Packet_T_Send_If_Connected (Self : in out
252    -- Base_Instance; Arg : in Packet.T; Full_Queue_Behavior : in
253    -- Full_Queue_Action := Drop);
254  not overriding procedure Packet_T_Send (Self : in out Base_Instance; Arg : in
255    -- Packet.T; Full_Queue_Behavior : in Full_Queue_Action := Drop)
256    with Inline => True;
257  not overriding function Is_Packet_T_Send_Connected (Self : in Base_Instance)
258    -- return Boolean
259    with Inline => True;
260  -----
261  -- The base class instance record:
262  -----
263  type Base_Instance is abstract new Component.Core_Instance with record
264    -- Internal asynchronous connector queue:
265    Queue : Queue_Package.Instance;
266    -- Invoker connector objects:
267    Connector_Sys_Time_T_Get : Sys_Time_T_Get_Connector.Instance;
268    Connector_Packet_T_Send : Packet_T_Send_Connector.Instance;
269  end record;
270
271 end Component.Queued_Component;

```

This look very similar to the *example_component* base package that is presented in Section 6.1.3.

The differences are discussed below.

First, take note of some public subprograms that were generated: `Get_Max_Queue_Element_Size`, `Get_Queue_Current_Percent_Used`, and `Get_Queue_Maximum_Percent_Used`. These public functions are meant to be used by the assembly and the Queue Monitor component that is included as part of Adamant. They should not be called within the component and so will not be discussed further here.

Also defined in the public section is `Init_Base` and `Final_Base` procedures. These subprograms are responsible for allocating the queue on the heap, and deallocating it. Note that the parameter to set the queue size specifies the size in bytes. The `Init_Base` procedure is called by the assembly during initialization of the component. See Section 6.7.2 for more details on this subprogram.

Next, let's take a look at the private section. There is a new enumeration type declared called `Connector_Identifier_Type`, which contains an enumeration literal for each of the asynchronous connectors in our component and a special `Quit` enumeration. This type is the queue label, and will be used to determine the type of object enqueued on the component's queue.

The next code of note are the two private subprograms `dispatch_n` and `dispatch_all`. These procedures "dispatch" N items off the queue, or drain the queue respectively. When an item is dispatched, it is removed from the queue, the label is inspected, and the correct connector handler procedure is called to process the queued item. These procedures are meant to be called within a queued, passive component to specify when the component should process any asynchronous items that it received. If these procedures are never called, then the queue will never be serviced, and the component will never act on any asynchronous calls, so they MUST be used within the component. These subprograms are not necessary in an queued, active component, as is discussed in Section 6.4.

Finally, if we look at the private component record definition we can see a new variable `queue` which is the actual queue object instantiated within the component. The type of the queue is `Queue_Package.Instance` which is a result of a generic instantiation of the Adamant `Labeled_Queue` type, seen in the private section. Note that the queue object itself should rarely need to be accessed from within the component hand-code, as the only interaction with the object is usually through the `dispatch_n` and `dispatch_all` procedures discussed in the previous paragraph.

OK, now it is time to hand-code the implementation for this component. As with the `example_component` we will first build the templates and copy them into our component directory.

```
> redo build/template/component-queued_component-implementation.ads
> redo build/template/component-queued_component-implementation.adb
> cp build/template/* . # copy templates into component dir
```

Next, we modify the templates to include the component's behavior. Below is the specification of the *queued component* implementation package.

component-queued_component-implementation.ads:

```
1 -----
2 -- Queued_Component Component Implementation Spec
3 -----
4
5 -- Includes:
6 with Tick;
7 with Packed_Byte;
8 with Packed_U16;
9 with Packet_Types; use Packet_Types;
```

```

11 -- This is the queued component.
12 package Component.Queued_Component.Implementation is
13
14   -- The component class instance record:
15   type Instance is new Queued_Component.Base_Instance with private;
16
17   private
18
19     -- The component class instance record:
20     type Instance is new Queued_Component.Base_Instance with record
21       -- The length of the packet data to send out:
22       Data_Length : Packet_Buffer_Length_Type := 0;
23       -- The content of the packet data to send out:
24       Data : Basic_Types.Byte_Array (0 .. 1) := [0, 0];
25     end record;
26
27   -----
28   -- Set Up Procedure
29   -----
30   -- Null method which can be implemented to provide some component
31   -- set up code. This method is generally called by the assembly
32   -- main.adb after all component initialization and tasks have been started.
33   -- Some activities need to only be run once at startup, but cannot be run
34   -- safely until everything is up and running, ie. command registration,
35   -- initial
36   -- data product updates. This procedure should be implemented to do these
37   -- things
38   -- if necessary.
39   overriding procedure Set_Up (Self : in out Instance) is null;
40
41   -----
42   -- Invokee connector primitives:
43   -----
44   -- This connector provides the schedule tick for the component.
45   overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in
46   -- Tick.T);
47   -- This connector receives a single byte asynchronously that is used to
48   -- populate the outgoing packet.
49   overriding procedure Packed_Byte_T_Recv_Async (Self : in out Instance; Arg : in
50   -- Packed_Byte.T);
51   -- This procedure is called when a Packed_Byte_T_Recv_Async message is
52   -- dropped due to a full queue.
53   overriding procedure Packed_Byte_T_Recv_Async_Dropped (Self : in out
54   -- Instance; Arg : in Packed_Byte.T);
55   -- This connector receives a 16-bit number asynchronously that is used to
56   -- populate the outgoing packet.
57   overriding procedure Packed_U16_T_Recv_Async (Self : in out Instance; Arg :
58   -- in Packed_U16.T);
59   -- This procedure is called when a Packed_U16_T_Recv_Async message is dropped
60   -- due to a full queue.
61   overriding procedure Packed_U16_T_Recv_Async_Dropped (Self : in out Instance;
62   -- Arg : in Packed_U16.T);
63
64   -----
65   -- Invoker connector primitives:
66   -----
67   -- This procedure is called when a Packet_T_Send message is dropped due to a
68   -- full queue.
69   overriding procedure Packet_T_Send_Dropped (Self : in out Instance; Arg : in
70   -- Packet.T) is null;

```

```
59 end Component.Queued_Component.Implementation;
```

The only modification made to this file from the template was to add two variable to the component record: data_Length which holds the length of data that will be used in the emitted packet, and data which is the actual data that will be used in the emitted packet. These variables will be set within the two asynchronous connector handlers, as seen in the body.

component-queued_component-implementation.adb:

```
1 -----  
2 -- Queued_Component Component Implementation Body  
3 -----  
4  
5 with Ada.Text_IO; use Ada.Text_IO;  
6  
7 package body Component.Queued_Component.Implementation is  
8  
9 -----  
10 -- Invokee connector primitives:  
11 -----  
12 -- This connector provides the schedule tick for the component.  
13 overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in  
→ Tick.T) is  
14     -- Ignore tick, since we don't actually use it for anything:  
15     Ignore : Tick.T renames Arg;  
16     -- Ignore the return from dispatch_all.  
17     Ignore2 : Natural;  
18 begin  
19     -- First, we need to service any asynchronous messages that we have  
→ received.  
20     Ignore2 := Self.Dispatch_All;  
21     -- ^ if anything is on the queue at this point, then the  
→ Packed_Byt_T_Recv_Async  
22     -- or Packed_U16_T_Recv_Async handlers will be called based on the data  
→ popped  
23     -- off the queue. The data returned is the number of items popped, which  
→ we  
-- ignore in this case, since this value is unused.  
24  
25     -- Let's create a packet.  
26     declare  
27         -- Fill in the packet length with the stored packet length.  
28         Pkt : Packet.T := (  
29             Header => (  
30                 Time => Self.Sys_Time_T_Get,  
31                 Id => 0,  
32                 Sequence_Count => 0,  
33                 Buffer_Length => Self.Data_Length  
34             ),  
35             Buffer => [others => 0]  
36         );  
37         begin  
38             -- Set the packet data:  
39             Pkt.Buffer (0 .. 1) := Self.Data;  
40  
41             -- Send the packet.  
42             Self.Packet_T_Send_If_Connected (Pkt);  
43         end;  
44     end Tick_T_Recv_Sync;  
45  
46     -- This connector receives a single byte asynchronously that is used to  
→ populate the outgoing packet.
```

```

48 overriding procedure Packed_Byte_T_Recv_Async (Self : in out Instance; Arg :
49   in Packed_Byte.T) is
50 begin
51   -- Save off the new packet data and length:
52   Self.Data (0 .. 0) := Packed_Byte.Serialization.To_Byte_Array (Arg);
53   Self.Data_Length := Packed_Byte.Serialization.Serialized_Length; -- 1
54 end Packed_Byte_T_Recv_Async;
55
56 -- This connector receives a 16-bit number asynchronously that is used to
57 -- populate the outgoing packet.
58 overriding procedure Packed_U16_T_Recv_Async (Self : in out Instance; Arg :
59   in Packed_U16.T) is
60 begin
61   -- Save off the new packet data and length:
62   Self.Data (0 .. 1) := Packed_U16.Serialization.To_Byte_Array (Arg);
63   Self.Data_Length := Packed_U16.Serialization.Serialized_Length; -- 2
64 end Packed_U16_T_Recv_Async;
65
66 -- This procedure is called when a Packed_Byte_T_Recv_Async message is
67 -- dropped due to a full queue.
68 overriding procedure Packed_Byte_T_Recv_Async_Dropped (Self : in out
69   Instance; Arg : in Packed_Byte.T) is
70   Ignore : Instance renames Self;
71 begin
72   Put_Line ("Oh no! The queue overflowed!");
73 end Packed_Byte_T_Recv_Async_Dropped;
74
75 -- This procedure is called when a Packed_U16_T_Recv_Async message is dropped
76 -- due to a full queue.
77 overriding procedure Packed_U16_T_Recv_Async_Dropped (Self : in out Instance;
78   Arg : in Packed_U16.T) is
79   Ignore : Instance renames Self;
80 begin
81   Put_Line ("Oh no! The queue overflowed!");
82 end Packed_U16_T_Recv_Async_Dropped;
83
84 end Component.Queued_Component.Implementation;

```

The first subprogram we should look at here is the `Tick_T_Recv_Sync` procedure that gets invoked when the `Tick_T_Recv_Sync` connector is called. When a `Tick.T` is received, the component first calls the `Self.Dispatch_All` function that is implemented in the component base package. Recall that this function will drain the component's internal queue, calling the correct connector handlers for each item dequeued. Specifically, this procedure will take items off the queue and call either the `Packed_Byte_T_Recv_Async` procedure or the `Packed_U16_T_Recv_Async` procedure depending on the type of the popped item. Next, a packet is created and sent out. The length of the packet is determined by the `Self.Data_Length` variable and the contents by the `data` variable.

Next, there are two subprograms `Packed_Byte_T_Recv_Async` and `Packed_U16_T_Recv_Async`. These procedures are the asynchronous connector handlers that will be invoked when `Self.Dispatch_All` or `Self.Dispatch_N` is called. They both perform similar actions, saving off the received data and data length into the component's internal variables for later packetization.

Finally, there are two subprograms `Packed_Byte_T_Recv_Async_Dropped` and `Packed_U16_T_Recv_Async_Dropped`. These procedures will be called by the autocoded base package whenever the `Packed_Byte_T_Recv_Async` or `Packed_U16_T_Recv_Async` connectors is invoked, but the internal queue is full. In this case, the data is dropped, the connector handler is not called, and, instead, the `*_Dropped` procedure is called. This functionality allows a component to perform some action when the internal queue overflows. Normally you will want to code some intelligent behavior here, but for simplicity the code above just produces a fail print using

`Ada.Text_IO.`

Note that this design is completely thread safe, assuming only a single task calls the `Tick_T_Recv_Sync` connector, which is the intended use. The internal queue is used as a synchronization mechanism. Data can be enqueued by multiple tasks simultaneously. Data is dequeued by the task that calls `Tick_T_Recv_Sync` connector, processed, and then emitted as a packet on the same task.

We can make sure that this code compiles by running:

```
> redo build/obj/Linux/component-queued_component-implementation.o
```

or by simply running `redo all`.

In the next section we will unit test this component.

6.3.3 Unit Testing a Queued Component

In this section we will create a unit test model to assist in unit testing the `queued_component`. This section will not discuss the details of unit testing a component. It will only highlight the differences in unit testing a queued component as compared to a passive component. You should be familiar with the concepts presented in Section 6.2 before reading this section.

To start, we create a unit test directory called `test/` under our component directory:

```
> mkdir test  # Make test/ within queued_component/
> cd test
```

Next, we are going to create a component unit test model file `queued_component.tests.yaml`. The contents are shown below:

```
1  ---
2  description: This is a set of unit tests for the Queued Component.
3  tests:
4    - name: Test_Nominal
5      description: This test tests the nominal behavior of the component.
6    - name: Test_Queue_Overflow
7      description: This test makes sure an assertion is thrown by the component if
       → the component's queue overflows.
```

We define two unit tests, one that tests the nominal execution of the component, sending it `Tick.Ts` and expecting packets to be produced, the other testing the behavior when the component's internal queue becomes full.

We also need to add an `env.py` file to this directory as we did in Section 6.2 to make sure we are using the `Linux_Test` target.

Next, we can generate the unit test templates and copy them into the test directory.

```
> redo build/template/component-queued_component-implementation-tester.ads
> redo build/template/component-queued_component-implementation-tester.adb
> redo build/template/queued_component_tests-implementation.ads
> redo build/template/queued_component_tests-implementation.adb
> redo build/template/test.adb
> cp build/template/* .  # copy the template files into test/
```

Now the unit tests can be implemented by editing `queued_component_tests-implementation.adb`.

The implemented unit tests are shown below.

queued_component_tests-implementation.adb:

```
1 -----  
2 -- Queued_Component Tests Body  
3 -----  
4  
5 with Basic_Assertions; use Basic_Assertions;  
6 with Packet.Assertion; use Packet.Assertion;  
7  
8 package body Queued_Component_Tests.Implementation is  
9  
10 -----  
11 -- Fixtures:  
12 -----  
13  
14 overriding procedure Set_Up_Test (Self : in out Instance) is  
15 begin  
16     -- Allocate heap memory to component:  
17     Self.Tester.Init_Base (Queue_Size =>  
18         => Self.Tester.Component_Instance.Get_Max_Queue_Element_Size * 3);  
19  
20     -- Make necessary connections between tester and component:  
21     Self.Tester.Connect;  
22  
23     -- Call the component set up method that the assembly would normally call.  
24     Self.Tester.Component_Instance.Set_Up;  
25 end Set_Up_Test;  
26  
27 overriding procedure Tear_Down_Test (Self : in out Instance) is  
28 begin  
29     -- Free component heap:  
30     Self.Tester.Final_Base;  
31 end Tear_Down_Test;  
32  
33 -----  
34 -- Tests:  
35 -----  
36  
37 overriding procedure Test_Nominal (Self : in out Instance) is  
38     -- Define the packet we are going to compare against.  
39     A_Packet : Packet.T := (  
40         Header => (  
41             Time => (0, 0),  
42             Id => 0,  
43             Sequence_Count => 0,  
44             Buffer_Length => 1  
45         ),  
46         Buffer => [others => 0]  
47     );  
48 begin  
49     -- Put some data in the component's queue:  
50     Self.Tester.Packed_Byte_T_Send ((Value => 5));  
51     Self.Tester.Packed_U16_T_Send ((Value => 4));  
52  
53     -- Expect no packets to be send out:  
54     Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 0);  
55  
56     -- Send a tick:  
57     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
```

```

58      -- Expect one packet with a U16 in it:
59      Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 1);
60      A_Packet.Header.Buffer_Length := 2;
61      A_Packet.Buffer (0) := 0;
62      A_Packet.Buffer (1) := 4;
63      Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (1),
64                          A_Packet);
65
66      -- Put some data in the component's queue:
67      Self.Tester.Packed_U16_T_Send ((Value => 15));
68      Self.Tester.Packed_Byt_T_Send ((Value => 12));
69
70      -- Send a tick:
71      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
72
73      -- Expect a second packet with a byte in it:
74      Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 2);
75      A_Packet.Header.Buffer_Length := 1;
76      A_Packet.Buffer (0) := 12;
77      A_Packet.Buffer (1) := 0;
78      Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (2),
79                          A_Packet);
80  end Test_Nominal;
81
82 overriding procedure Test_Queue_Overflow (Self : in out Instance) is
83 begin
84     -- Fill up the queue:
85     Self.Tester.Packed_U16_T_Send ((Value => 4));
86     Self.Tester.Packed_U16_T_Send ((Value => 4));
87     Self.Tester.Packed_U16_T_Send ((Value => 4));
88     -- This call should produce a unit test failure.
89     Self.Tester.Packed_U16_T_Send ((Value => 4));
90 end Test_Queue_Overflow;
91
92 end Queued_Component_Tests.Implementation;

```

The first unit test sends some data to the component through the asynchronous connectors. It then sends a tick and makes sure a packet is produced with the correct output data.

The second unit test simply fills the component queue. Note that queue size is set in the Set_Up_Test fixture to 3 times the maximum queue element size. In this case, the maximum queue element would correspond to enqueueing a Packed_U16 type. This is achieved by calling Self.Tester.Packed_U16_T_Send 4 times. The fourth call should overflow the queue.

The unit test can be compiled and run using the command:

```
> redo test
```

which produces the output

```
Oh no! The queue overflowed!
```

```
OK Test_Nominal
```

```
ERROR Test_Queue_Overflow
ADA ASSERTIONS ASSERTION_ERROR
Exception Message: The component's queue filled up when Packed_U16_T_Send was called
Traceback:
```

```
Total Tests Run: 2
Successful Tests: 1
Failed Assertions: 0
Unexpected Errors: 1
```

As can be seen, the component's `*_Dropped` handler was called because the "Oh no! The queue overflowed!" message is printed. We can also see that the second unit test failed with an assertion. This assertion is being triggered inside of the tester component, which throws an assertion if it ever detects that the internal component's queue overflowed. This is the default behavior to preventing the accidental overflowing of a queue during unit test.

There are times, however, when we actually *want* to overflow a queue during unit test to test that specific behavior. In this case, we can modify the unit test as follows.

`queued_component_tests-implementation.adb:`

```
1-----  
2  -- Queued_Component Tests Body  
3-----  
4  
5  with Basic_Assertions; use Basic_Assertions;  
6  with Packet.Assertion; use Packet.Assertion;  
7  
8  package body Queued_Component_Tests.Implementation is  
9  
10 -----  
11  -- Fixtures:  
12 -----  
13  
14  overriding procedure Set_Up_Test (Self : in out Instance) is  
15  begin  
16      -- Allocate heap memory to component:  
17      Self.Tester.Init_Base (Queue_Size =>  
18          → Self.Tester.Component_Instance.Get_Max_Queue_Element_Size * 3);  
19  
20      -- Make necessary connections between tester and component:  
21      Self.Tester.Connect;  
22  
23      -- Call the component set up method that the assembly would normally call.  
24      Self.Tester.Component_Instance.Set_Up;  
25  end Set_Up_Test;  
26  
27  overriding procedure Tear_Down_Test (Self : in out Instance) is  
28  begin  
29      -- Free component heap:  
30      Self.Tester.Final_Base;  
31  end Tear_Down_Test;  
32  
33 -----  
34  -- Tests:  
35 -----  
36  
37  overriding procedure Test_Nominal (Self : in out Instance) is  
38      -- Define the packet we are going to compare against.  
39      A_Packet : Packet.T := (  
40          Header => (  
41              Time => (0, 0),  
42              Id => 0,  
43              Sequence_Count => 0,
```

```

43         Buffer_Length => 1
44     ),
45     Buffer => [others => 0]
46 );
47 begin
48     -- Put some data in the component's queue:
49     Self.Tester.Packed_Byte_T_Send ((Value => 5));
50     Self.Tester.Packed_U16_T_Send ((Value => 4));
51
52     -- Expect no packets to be send out:
53     Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 0);
54
55     -- Send a tick:
56     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
57
58     -- Expect one packet with a U16 in it:
59     Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 1);
60     A_Packet.Header.Buffer_Length := 2;
61     A_Packet.Buffer (0) := 0;
62     A_Packet.Buffer (1) := 4;
63     Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (1),
64                       → A_Packet);
65
66     -- Put some data in the component's queue:
67     Self.Tester.Packed_U16_T_Send ((Value => 15));
68     Self.Tester.Packed_Byte_T_Send ((Value => 12));
69
70     -- Send a tick:
71     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
72
73     -- Expect a second packet with a byte in it:
74     Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 2);
75     A_Packet.Header.Buffer_Length := 1;
76     A_Packet.Buffer (0) := 12;
77     A_Packet.Buffer (1) := 0;
78     Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (2),
79                       → A_Packet);
80 end Test_Nominal;
81
82 overriding procedure Test_Queue_Overflow (Self : in out Instance) is
83 begin
84     -- Fill up the queue:
85     Self.Tester.Packed_U16_T_Send ((Value => 4));
86     Self.Tester.Packed_U16_T_Send ((Value => 4));
87     Self.Tester.Packed_U16_T_Send ((Value => 4));
88
89     -- Tell the tester component that we are expecting the next connector
90     -- call to be dropped. This allows us to continue the test without
91     -- an assertion being thrown by the tester.
92     Self.Tester.Expect_Packed_U16_T_Send_Dropped := True;
93
94     -- This call should overflow the queue, but not cause the test to
95     -- fail.
96     Self.Tester.Packed_U16_T_Send ((Value => 4));
97 end Test_Queue_Overflow;
98
99 end Queued_Component_Tests.Implementation;

```

In this case, we have added the line `Self.Tester.Expect_Packed_U16_T_Send_Dropped := True;` which tells the tester component that the next connector call will overflow the queue. When this is set, the component will not throw an assertion. See the new output below.

```
Oh no! The queue overflowed!
```

```
OK Test_Nominal
OK Test_Queue_Overflow

Total Tests Run: 2
Successful Tests: 2
Failed Assertions: 0
Unexpected Errors: 0
```

Note that any subsequent call that may overflow the queue will trigger the assertion again. The `Self.Tester.Expect_Packed_U16_T_Send_Dropped` variable must be set before every call to a connector that will be dropped. In this way, the tester component will always alert the developer if something was dropped off a queue during unit test that was not intended.

6.3.4 Implementing a Priority Queued Component

The previous sections presented how to construct and unit test a component that contains a standard internal queue. The Adamant standard queue is a very efficient (in both space and time) first-in-first-out FIFO queue. However, sometimes it is desirable to have a queue that implements priority-based dequeuing logic instead of FIFO. Priority queues allow the component to process messages deemed more important prior to servicing others, regardless of the order in which they were enqueued.

A few implementation differences between the Adamant standard queue and the priority queue should be understood. The standard queue is implemented as a single buffer of memory where queued items are stored efficiently, packed together, with no wasted bytes. As such, initializing the standard queue requires that the user specify the total number of bytes the queue should contain. The priority queue does not make as efficient use of memory. Each element on the priority queue takes up the number of bytes necessary to hold the maximum sized element that the queue will need to hold. So if two different types need to be stored on the queue, one takes up 5 bytes and another takes up 500 bytes, each element on the queue will be 500 bytes in length. Enqueuing the 5 byte item will leave 495 bytes unused in that queue entry. Instead of initializing the queue as a total number of bytes, a priority queue is initialized with a *depth* which specifies how many entries of maximum size the queue will hold before beginning to drop items.

A priority queue is also less efficient in terms of performance than a standard Adamant queue. The standard queue has a complexity of $O(1)$ queue and dequeue execution times. The priority queue is slower, taking $O(\log n)$ queue and dequeue where n is the number of items currently stored on the queue.

With this in mind, priority queues should only be used where servicing items based on priority is essential, and that benefit outweighs the performance hits in both time and space. If you are still convinced you need to implement a component with a priority queue, continue on.

Adamant provides priority queueing for a component if a priority is specified for a `recv_async` connector. For example, we can instantiate the `queued_component` presented in the previous sections with an internal priority queue by defining priorities for each of the two `recv_async` connectors. Let's call it `priority_queued_component`.

`priority_queued_component.component.yaml`

```
1  ---
2  execution: passive
3  description: This is the priority queued component.
4  connectors:
5    #####
```

```

6   # Invokee Connectors
7   #####
8   - type: Tick.T
9     kind: recv_sync
10    description: This connector provides the schedule tick for the component.
11   #####
12   # Asynchronous Invokee Connectors
13   #####
14   - type: Packed_Byte.T
15     kind: recv_async
16    description: This connector receives a single byte asynchronously that is
17      ↳ used to populate the outgoing packet. Data received on this connector is
18      ↳ considered lower priority than data received on the
19      ↳ Packed_U16_T_Recv_Async connector.
20    priority: 1
21   - type: Packed_U16.T
22     kind: recv_async
23    description: This connector receives a 16-bit number asynchronously that is
24      ↳ used to populate the outgoing packet. Data received on this connector is
25      ↳ considered higher priority than data received on the
26      ↳ Packed_Byte_T_Recv_Async connector.
27    priority: 2
28   #####
29   # Invoker Connectors
30   #####
31   - return_type: Sys_Time.T
32     kind: get
33    description: This connector is used to fetch the current system time.
34   - type: Packet.T
35     kind: send
36    description: This connector is used to send out a telemetry packet.

```

You can see that a priority of 1 has been specified for the `Packed_Byte_T_Recv_Async` connector and a priority of 2 has been specified for the `Packed_U16_T_Recv_Async` connector. Connector's of higher numeric valued priorities will be serviced before connectors of lower numeric valued priorities. So in this case if both an `Packed_U16_T_Recv_Async` and a `Packed_Byte_T_Recv_Async` item are put on the queue, the `Packed_U16_T_Recv_Async` item will be dequeued first, no matter the order in which the items were enqueued.

The connector priority can be any integer value between 0 and 255, inclusive. Connectors with an identical priority value will have their enqueued items serviced in FIFO order. Note that if not specified, the priority for a `recv_async` connector is assumed to be 0. If all connectors have the same priority, the component is instantiated with a standard Adamant queue instead of a priority queue. Specifying priority for a connector not of kind `recv_async` is considered illegal and will produce a modeling error.

The model file above generates the following component diagram. See Section 6.1.1 for directions on how to create a component diagram.

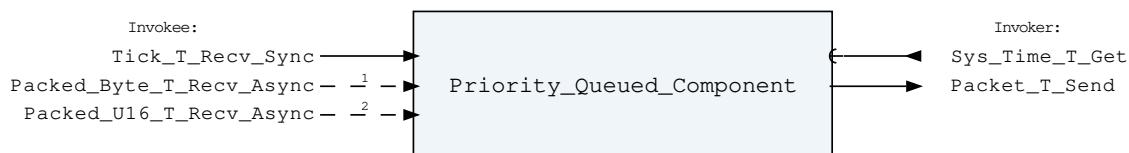


Table 8: A priority queued component which has two asynchronous connectors of different priority.

You can see how the priorities of each `recv_async` connector is now visible in the diagram.

Implementing and unit testing a component with a priority queue is no different than implementing a component with a standard queue. You just have to remember that the `recv_async` connector handlers will be called in the order of the items enqueued, and not in FIFO order.

6.4 Creating an Active Component

In this section we will be working with an *active* component, that is, a component with a thread of execution, or a *task* allocated to it. This means that the component will run whenever the task becomes ready to run and no other higher priority tasks are running. The Ada scheduler manages the running of active tasks based on their priority. More details on the component task are provided in the next section.

6.4.1 The Component Task

A component is considered "active" in Adamant when it is provided a task of execution. This is done by specifying the component *execution* as "active" in the component model, as will be seen in the following section. When a component is defined active, the assembly must instantiate a task for that component during initialization. The definition of this task is the same for any active component, and is defined in the Adamant component core package in `src/core/component/`. To better understand what a component task looks like the specification of the component package is shown below.

component.ads:

```
1  with Task_Types;
2  with Ada.Synchronous_Task_Control;
3  with System;
4  with Basic_Types;
5
6  -- This is a base component. There is not much here, but
7  -- this provides a base type for which other types can
8  -- use to operate on more specific components.
9  package Component is
10
11    -- Component Type:
12    type Core_Instance is abstract tagged limited null record;
13    type Class_Access is access all Core_Instance'Class;
14
15    -- List type:
16    type Component_List is array (Natural range <>) of Class_Access;
17    type Component_List_Access is access all Component_List;
18
19    -- Component abstract method, executed cyclically in task:
20    procedure Cycle (Ignore : in out Core_Instance) is abstract;
21
22    -- Task type for active components:
23    task type Active_Task (
24      Task_Data : Task_Types.Task_Info_Access;
25      Class_Self : not null Class_Access;
26      Signal : not null access Ada.Synchronous_Task_Control.Suspension_Object;
27      Pri : System.Priority;
28      Stack_Size : Natural;
29      Secondary_Stack_Size : Natural
30    ) -- Set the priority and stack size for the task:
31      with Priority => Pri,
32           Storage_Size => Stack_Size,
33           Secondary_Stack_Size => Secondary_Stack_Size;
34
35    -- Null method which can be overridden to provide some component
```

```

36  -- set up code. This method is generally called by the assembly
37  -- main.adb after all component initialization and tasks have been started.
38  -- Some activities need to only be run once at startup, but cannot be run
39  -- safely until everything is up and running, ie. command registration,
40  -- initial
41  -- data product updates. This procedure should be overridden to do these
42  -- things
43  -- if necessary.
44  procedure Set_Up (Self : in out Core_Instance) is null;
45
46  --
47
48  -- Get the current percent usage of the component's internal queue, if there
49  -- is one, else assert. This function is used by the queue monitor component
50  -- as a backdoor convenience to bypass the connector system, reducing
51  -- complexity
52  -- of reporting queue usage for every component.
53  function Get_Queue_Current_Percent_Used (Self : in out Core_Instance) return
54  -- Basic_Types.Byte;
55
56  -- Get the maximum percent usage of the component's internal queue, if there
57  -- is one, else assert. This function is used by the queue monitor component
58  -- as a backdoor convenience to bypass the connector system, reducing
59  -- complexity
60  -- of reporting queue usage for every component. ie. "high water mark"
61  function Get_Queue_Maximum_Percent_Used (Self : in out Core_Instance) return
62  -- Basic_Types.Byte;
63
64 end Component;

```

There are a lot of low level details here that we don't need to get into, however, there are a few things to note. First look at the abstract procedure `Cycle`. This abstract subprogram must be overridden by an inheriting type in order to specify the behavior that the component will perform on each "cycle" of the task. This subprogram is called inside the task definition for `Active_Task`.

We can see that `Active_Task` takes many parameters in its discriminant including vital task information such as the priority and stack size. To fully understand what happens in a component task we need to look at the body.

component.adb:

```

1  with Task_Util;
2
3  package body Component is
4
5    task body Active_Task is
6      begin
7        -- Initialize the stack's task_data and stack.
8        Task_Util.Initialize_Task (Task_Data, Pri, Stack_Size,
9        -- Secondary_Stack_Size);
10
11        -- Wait to start until signaled:
12        Ada.Synchronous_Task_Control.Suspend_Until_True (Signal.all);
13        while not Ada.Synchronous_Task_Control.Current_State (Signal.all) loop
14          -- Call the task procedure:
15          Class_Self.all.Cycle;
16          -- Update secondary stack usage:
17          Task_Util.Update_Secondary_Stack_Usage (Task_Data);

```

```

17      end loop;
18  end Active_Task;
19
20  function Get_Queue_Current_Percent_Used (Self : in out Core_Instance) return
21    Basic_Types.Byte is
22    pragma Annotate (GNATSA, Intentional, "subp always fails",
23                      "Intentional - this subp should never be called on a component without
24                      a queue.");
25    Ignore : Core_Instance renames Self;
26  begin
27    pragma Assert (False, "This component does not contain a queue because
28                      this subprogram was not overridden.");
29    return Basic_Types.Byte'Last;
30  end Get_Queue_Current_Percent_Used;
31
32  function Get_Queue_Maximum_Percent_Used (Self : in out Core_Instance) return
33    Basic_Types.Byte is
34    pragma Annotate (GNATSA, Intentional, "subp always fails",
35                      "Intentional - this subp should never be called on a component without
36                      a queue.");
37    Ignore : Core_Instance renames Self;
38  begin
39    pragma Assert (False, "This component does not contain a queue because
40                      this subprogram was not overridden.");
41    return Basic_Types.Byte'Last;
42  end Get_Queue_Maximum_Percent_Used;
43
44 end Component;

```

If we look at task body `Active_Task` we can see that it enters an infinite loop where it calls `Cycle` repeatedly. The `Cycle` procedure does all the work in this case, and must be defined by an inheriting type.

As you can imagine, a programmer can override the `Cycle` program and code their own custom logic to perform on a component's task. This is demonstrated in Section 6.5. However, in Adamant systems, this is rarely, if ever, done. The most common pattern is to use an active component in tandem with a component's internal queue to define the execution of the component. By default, if a component is both active and has at least one asynchronous connector, the `Cycle` procedure gets defined in the component's base package with the following behavior.

1. The component task "blocks" (waits) on queue until it is no longer empty.
2. When an item is placed on the queue the component becomes ready to run. When there are no higher priority tasks running, the scheduler will run this task.
3. The component pops an item off the queue and then processes it by calling the appropriate asynchronous connector handler procedure.
4. Go back to step 1.

In this way, the execution of the component is metered by the work it needs to do. The work it does is determined by the items that are put on its queue, which are processed serially. In this case, the queue acts as a synchronization mechanism, providing thread safety for the active component's execution by ordering the component processing in a FIFO manner. The processing of each item is completed before the next item is popped off the queue.

In the following section a component is constructed using this pattern.

6.4.2 Implementing an Active Component

Let's create a new component called *active_component* in a new directory.

```
> mkdir active_component      # make component directory
> cd active_component
> touch .all_path           # add directory to build path
```

To define the component in this directory, we create a YAML model file *active_component.component.yaml* with the following contents.

active_component.component.yaml

```

1  ---
2  execution: active
3  description: This is the active component.
4  connectors:
5  #####
6  # Asynchronous Invokee Connectors
7  #####
8  - type: Packed_Byte.T
9    kind: recv_async
10   description: This connector receives a single byte asynchronously that is
11     ↪ used to populate the outgoing packet.
12   - type: Packed_U16.T
13     kind: recv_async
14     description: This connector receives a 16-bit number asynchronously that is
15       ↪ used to populate the outgoing packet.
16 #####
17 # Invoker Connectors
18 #####
19 - return_type: Sys_Time.T
20   kind: get
21   description: This connector is used to fetch the current system time.
22 - type: Packet.T
23   kind: send
24   description: This connector is used to send out a telemetry packet.
```

This model file generates the following component diagram. See Section 6.1.1 for directions on how to create a component diagram.



Table 9: An active component which has two asynchronous connectors.

We want this component to be similar to the *queued_component* we constructed in the previous sections. It will still produce a timestamped packet based on input data from two asynchronous connectors. However, this time, there is no *Tick.T* connector that drives the component's execution. Instead, the component is made active, and so will produce a packet any time data is put on its internal queue.

First let's take a look at the component base package so that we can see the component's definition of the *Cycle* procedure. The base package can autocoded and compiled with the following command.

```
> redo build/obj/Linux/component-active_component.o
```

This should generate and compile two files *build/src/component-active_component.ads* and *build/src/component-active_component.adb*. The specification and body are not shown here because they are quite large. However the reader is encouraged to inspect them to see that the `Cycle` procedure is indeed implemented with the logic described in the previous section.

OK, now it is time to hand-code the implementation for this component. As with the `example_component` we will first build the templates and copy them into our component directory.

```
> redo build/template/component-active_component-implementation.ads
> redo build/template/component-active_component-implementation.adb
> cp build/template/* . # copy templates into component dir
```

Next, we modify the templates to include the component's behavior. To implement this component, no modifications were made to the specification. In particular, this component defines no variables in the component record in the implementation specification. The connector handler behavior is encoded in the implementation body, which is shown below.

component-active_component-implementation.adb:

```
1 -----
2 -- Active_Component Component Implementation Body
3 -----
4
5 with Ada.Text_IO; use Ada.Text_IO;
6
7 package body Component.Active_Component.Implementation is
8
9   -----
10  -- Invokee connector primitives:
11  -----
12  -- This connector receives a single byte asynchronously that is used to
13  -- populate the outgoing packet.
14  overriding procedure Packed_Byte_T_Recv_Async (Self : in out Instance; Arg :
15  -- in Packed_Byte.T) is
16    -- Fill in the packet length with the packet length.
17    Pkt : Packet.T := (
18      Header => (
19        Time => Self.Sys_Time_T_Get,
20        Id => 0,
21        Sequence_Count => 0,
22        Buffer_Length => Packed_Byte.Serialization.Serialized_Length
23      ),
24      Buffer => [others => 0]
25    );
26 begin
27   -- Set the packet data:
28   Pkt.Buffer (0 .. 0) := Packed_Byte.Serialization.To_Byte_Array (Arg);
29
30   -- Send the packet.
31   Self.Packet_T_Send_If_Connected (Pkt);
32 end Packed_Byte_T_Recv_Async;
33
34  -- This connector receives a 16-bit number asynchronously that is used to
35  -- populate the outgoing packet.
36  overriding procedure Packed_U16_T_Recv_Async (Self : in out Instance; Arg :
37  -- in Packed_U16.T) is
```

```

34      -- Fill in the packet length with the packet length.
35      Pkt : Packet.T := (
36          Header => (
37              Time => Self.Sys_Time_T_Get,
38              Id => 0, Sequence_Count => 0,
39              Buffer_Length => Packed_U16.Serialization.Serialized_Length
40          ),
41          Buffer => [others => 0]
42      );
43  begin
44      -- Set the packet data:
45      Pkt.Buffer (0 .. 1) := Packed_U16.Serialization.To_Byte_Array (Arg);
46
47      -- Send the packet.
48      Self.Packet_T_Send_If_Connected (Pkt);
49  end Packed_U16_T_Recv_Async;
50
51  -- This procedure is called when a Packed_Bye_T_Recv_Async message is
52  -- dropped due to a full queue.
53  overriding procedure Packed_Bye_T_Recv_Async_Dropped (Self : in out
54  -- Instance; Arg : in Packed_Bye.T) is
55      Ignore : Instance renames Self;
56  begin
57      Put_Line ("Oh no! The queue overflowed!");
58  end Packed_Bye_T_Recv_Async_Dropped;
59
60  -- This procedure is called when a Packed_U16_T_Recv_Async message is dropped
61  -- due to a full queue.
62  overriding procedure Packed_U16_T_Recv_Async_Dropped (Self : in out Instance;
63  -- Arg : in Packed_U16.T) is
64      Ignore : Instance renames Self;
65  begin
66      Put_Line ("Oh no! The queue overflowed!");
67  end Packed_U16_T_Recv_Async_Dropped;
68
69  end Component.Active_Component.Implementation;

```

We can see that the implementations of `Packed_Bye_T_Recv_Async` and `Packed_U16_T_Recv_Async` are very similar. Any time an item is popped off the component's queue one of these handlers is called. A packet is then created containing the data item and sent out of the component. The `Packed_Bye_T_Recv_Async_Dropped` and `Packed_U16_T_Recv_Async_Dropped` subprograms are implemented in the same fashion as in Section 6.3.2 and will get called if the component queue ever becomes too full to hold another item.

We can make sure that this code compiles by running:

```
> redo build/obj/Linux/component-active_component-implementation.o
```

or by simply running `redo all`.

6.4.3 Unit Testing an Active Component

In this section we will create a unit test model to assist in unit testing the `active_component`. This section will not discuss the details of unit testing a component. It will only highlight the differences in unit testing an active component as compared to a queued component. You should be familiar with the concepts presented in Section 6.2 before reading this section.

To start, we create a unit test directory called `test/` under our component directory:

```
> mkdir test  # Make test/ within active_component/
> cd test
```

Next, we are going to create a component unit test model file *active_component.tests.yaml*. The contents are shown below:

```
1  ---
2  description: This is a set of unit tests for the Active Component.
3  tests:
4    - name: Test_Nominal
5      description: This test tests the nominal behavior of the component.
6    - name: Test_Queue_Overflow
7      description: This test makes sure an assertion is thrown by the component if
       ↳ the component's queue overflows.
```

We define two unit tests, one that tests the nominal execution of the component, sending it data items on its internal queue and expecting packets to be produced, the other testing the behavior when the component's internal queue becomes full.

We also need to add an *env.py* file to this directory as we did in Section 6.2 to make sure we are using the `Linux_Test` target.

Next, we can generate the unit test templates and copy them into the test directory.

```
> redo build/template/component-active_component-implementation-tester.ads
> redo build/template/component-active_component-implementation-tester.adb
> redo build/template/active_component_tests-implementation.ads
> redo build/template/active_component_tests-implementation.adb
> redo build/template/test.adb
> cp build/template/* .  # copy the template files into test/
```

Now the unit tests can be implemented by editing *active_component_tests-implementation.adb*. The implemented unit tests are shown below.

active_component_tests-implementation.adb:

```
1  -----
2  -- Active_Component Tests Body
3  -----
4
5  with Basic_Assertions; use Basic_Assertions;
6  with Packet.Assertion; use Packet.Assertion;
7
8  package body Active_Component_Tests.Implementation is
9
10   -----
11   -- Fixtures:
12   -----
13
14   overriding procedure Set_Up_Test (Self : in out Instance) is
15   begin
16     -- Allocate heap memory to component:
17     Self.Tester.Init_Base (Queue_Size =>
18       ↳ Self.Tester.Component_Instance.Get_Max_Queue_Element_Size * 3);
19
20     -- Make necessary connections between tester and component:
21     Self.Tester.Connect;
22
23     -- Call the component set up method that the assembly would normally call.
```

```

23     Self.Tester.Component_Instance.Set_Up;
24 end Set_Up_Test;
25
26 overriding procedure Tear_Down_Test (Self : in out Instance) is
27 begin
28     -- Free component heap:
29     Self.Tester.Final_Base;
30 end Tear_Down_Test;
31
32 -----
33 -- Tests:
34 -----
35
36 overriding procedure Test_Nominal (Self : in out Instance) is
37     -- Define the packet we are going to compare against.
38     A_Packet : Packet.T := (
39         Header => (
40             Time => (0, 0),
41             Id => 0,
42             Sequence_Count => 0,
43             Buffer_Length => 1
44         ),
45         Buffer => [others => 0]
46     );
47 begin
48     -- Put some data in the component's queue:
49     Self.Tester.Packed_Byte_T_Send ((Value => 5));
50     Self.Tester.Packed_U16_T_Send ((Value => 4));
51
52     -- Expect no packets to be sent out. Packets will not be sent out until
53     -- we tell the active component to execute.
54     Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 0);
55
56     -- Tell the active component to execute by dispatching its queue. There
57     -- should be two elements popped off the queue.
58     Natural Assert.Eq (Self.Tester.Dispatch_All, 2);
59
60     -- Expect one packet with a byte in it, and the other with a U16.
61     Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 2);
62     A_Packet.Header.Buffer_Length := 1;
63     A_Packet.Buffer (0) := 5;
64     A_Packet.Buffer (1) := 0;
65     Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (1),
66         => A_Packet);
67     A_Packet.Header.Buffer_Length := 2;
68     A_Packet.Buffer (0) := 0;
69     A_Packet.Buffer (1) := 4;
70     Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (2),
71         => A_Packet);
72
73     -- Put some data in the component's queue:
74     Self.Tester.Packed_U16_T_Send ((Value => 15));
75     Self.Tester.Packed_Byte_T_Send ((Value => 12));
76
77     -- Tell the active component to execute by dispatching its queue. There
78     -- should be two elements popped off the queue.
79     Natural Assert.Eq (Self.Tester.Dispatch_All, 2);
80
81     -- Expect two more packets.
82     Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 4);
83     A_Packet.Header.Buffer_Length := 2;

```

```

82     A_Packet.Buffer (0) := 0;
83     A_Packet.Buffer (1) := 15;
84     Packet_Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (3),
85         => A_Packet);
86     A_Packet.Header.Buffer_Length := 1;
87     A_Packet.Buffer (0) := 12;
88     A_Packet.Buffer (1) := 0;
89     Packet_Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (4),
90         => A_Packet);
91 end Test_Nominal;
92
93 overriding procedure Test_Queue_Overflow (Self : in out Instance) is
94 begin
95     -- Fill up the queue:
96     Self.Tester.Packed_U16_T_Send ((Value => 4));
97     Self.Tester.Packed_U16_T_Send ((Value => 4));
98     Self.Tester.Packed_U16_T_Send ((Value => 4));
99
100    -- Tell the tester component that we are expecting the next connector
101    -- call to be dropped. This allows us to continue the test without
102    -- an assertion being thrown by the tester.
103    Self.Tester.Expect_Packed_U16_T_Send_Dropped := True;
104
105    -- This call should overflow the queue, but not cause the test to
106    -- fail.
107    Self.Tester.Packed_U16_T_Send ((Value => 4));
108 end Test_Queue_Overflow;
109
110 end Active_Component_Tests.Implementation;

```

The first unit test sends some data to the component through the asynchronous connectors. Note, that the act of sending this data to the component does not cause the component to execute, like it might when the component is running in an assembly. Recall that component unit testing in Adamant is single threaded to make the act of unit testing simpler. This means that an active component is not actually given a thread of execution during unit test. This is advantageous because the unit test developer can exactly specify when the component's internal task executes manually. In this way, very fine grain testing can be achieved.

To simulate the component's task executing, the tester component is autocoded with the subprograms `dispatch_all` and `dispatch_n`. Similar to the same-named methods autocoded in the component base package seen in Section 6.3.2, these subprograms will dequeue items off the component's internal queue and process them in the same manner that the `Cycle` procedure would when a task is allocated. The advantage is that the programmer can call these dispatch functions at the exact moment they want the component to run instead of dealing with the complexities of managing the execution of a multithreaded unit test environment.

Shown in the first test, data is passed to the component on its asynchronous connectors. After that a check is made to ensure that no packets were produced by the component yet. Next, the `dispatch_all` procedure is called, causing the component to "execute", draining its queue and performing all associated processing. After this, we expect two packets to have been produced with the correct internal data. This procedure is then repeated to make sure the component behaves properly.

The second unit test simply fills the component queue. Note that queue size is set in the `Set_Up_Test` fixture to 3 times the maximum queue element size. In this case, the maximum queue element would correspond to enqueueing a `Packed_U16` type. This is achieved by calling `Self.Tester.Packed_U16_T_Send` 4 times. The fourth call overflows the queue. This test is identical to the second unit test defined in Section 6.3.3.

The unit test can be compiled and run using the command:

```
> redo test
```

which produces the output

```
Oh no! The queue overflowed!
```

```
OK Test_Nominal  
OK Test_Queue_Overflow  
  
Total Tests Run: 2  
Successful Tests: 2  
Failed Assertions: 0  
Unexpected Errors: 0
```

6.5 Creating a Background Component

As discussed in the previous section, active components in Adamant almost always have an asynchronous queue which aids in metering the execution of that component. There are a few instances where not having an asynchronous connector makes sense, mostly in the case of the lowest priority background component (ie. memory scrubbing), or a component whose execution is metered internally by sleeping periodically. This section will briefly show a component of this nature and explain how to unit test it.

6.5.1 Implementing a Background Component

Let's create a new component called *background_component* in a new directory.

```
> mkdir background_component # make component directory  
> cd background_component  
> touch .all_path # add directory to build path
```

To define the component in this directory, we create a YAML model file *background_component.component.yaml* with the following contents.

background_component.component.yaml

```
1 ---  
2   execution: active  
3   description: This is the background component.  
4   connectors:  
5     #####  
6     # Synchronous Invokee Connectors  
7     #####  
8     - type: Packed_U16.T  
9       kind: recv_sync  
10      description: This connector receives a 16-bit number synchronously that is  
11        ↳ used to populate the outgoing packet.  
12      #####  
13      # Invoker Connectors  
14      #####  
15      - return_type: Sys_Time.T  
16        kind: get  
17        description: This connector is used to fetch the current system time.  
18        - type: Packet.T
```

```

18 kind: send
19 description: This connector is used to send out a telemetry packet.

```

This model file generates the following component diagram. See Section 6.1.1 for directions on how to create a component diagram.



Table 10: An active component which has one synchronous connector.

We want this component to be similar to the *active_component* we constructed in the previous sections, however it contains no asynchronous connectors. Instead the packet data is received synchronously. A packet is sent out at a period rate, managed internally by the component's task.

OK, now it is time to hand-code the implementation for this component. As with the *example_component* we will first build the templates and copy them into our component directory.

```

> redo build/template/component-background_component-implementation.ads
> redo build/template/component-background_component-implementation.adb
> cp build/template/* .   # copy templates into component dir

```

Before we modify the templates, let's take a look at them first. Below is the implementation file.

component-background_component-implementation.adb:

```

1 -----  

2 -- Background_Component Component Implementation Body  

3 -----  

4  

5 package body Component.Background_Component.Implementation is  

6  

7     -----  

8     -- Invokee connector primitives:  

9     -----  

10    -- This connector receives a 16-bit number synchronously that is used to  

11    -- populate  

12    -- the outgoing packet.  

13    overriding procedure Packed_U16_T_Recv_Sync (Self : in out Instance; Arg : in  

14    -- Packed_U16.T) is  

15        -- TODO declarations  

16    begin  

17        null; -- TODO statements  

18    end Packed_U16_T_Recv_Sync;  

19  

20    -----  

21    -- Definition of cycle function for task execution:  

22    -----  

23    -- This is an active component with no queue, so the  

24    -- cycle function for the component's task must be  

25    -- implemented here in the implementation class as  

26    -- a user defined custom function.  

27    overriding procedure Cycle (Self : in out Instance) is  

28        Ignore : Instance renames Self;

```

```

27 begin
28     -- TODO: implement custom task procedure. This function
29     -- will be called indefinitely inside the task loop.
30     pragma Assert (False);
31 end Cycle;
32
33 end Component.Background_Component.Implementation;

```

As can be seen, there is an overriding `Cycle` procedure. Because there is no asynchronous queue, Adamant does not provide a default implementation of this procedure in the base package autocode. Instead, the programmer is asked to define the task behavior by implementing the `Cycle` function in the implementation package. By default, the implementation simply fails with an assertion, making sure the developer is quickly alerted if this procedure is not properly implemented.

Next, let's modify the templates to include the component's behavior. First, let's look at the specification.

component-background_component-implementation.ads:

```

1 -----
2 -- Background_Component Component Implementation Spec
3 -----
4
5 -- Includes:
6 with Packed_U16;
7 with Ada.Real_Time; use Ada.Real_Time;
8
9 -- This is the background component.
10 package Component.Background_Component.Implementation is
11
12     -- The component class instance record:
13     type Instance is new Background_Component.Base_Instance with private;
14
15 private
16     -- Declare a two byte array type for convenience.
17     subtype Two_Byte_Array is Basic_Types.Byte_Array (0 .. 1);
18
19     -- A protected type used for ensuring mutual exclusion around the
20     -- bytes of data put into the packet. Since this data can be set by
21     -- multiple external tasks simultaneously, and could be being copied
22     -- into a packet at the same time, we need to protect access to the
23     -- data too ensure there is no race condition and that the data never
24     -- gets corrupted.
25     protected type Protected_Data is
26         -- Set the data.
27         procedure Set_Data (New_Data : in Two_Byte_Array);
28         -- Fetch the data.
29         function Get_Data return Two_Byte_Array;
30     private
31         Data : Basic_Types.Byte_Array (0 .. 1) := [0, 0];
32     end Protected_Data;
33
34     -- The component class instance record:
35     type Instance is new Background_Component.Base_Instance with record
36         P_Data : Protected_Data;
37         Wake_Up_Time : Ada.Real_Time.Time := Ada.Real_Time.Clock;
38     end record;
39
40 -----
41     -- Set Up Procedure

```

```

42 -----
43 -- Null method which can be implemented to provide some component
44 -- set up code. This method is generally called by the assembly
45 -- main.adb after all component initialization and tasks have been started.
46 -- Some activities need to only be run once at startup, but cannot be run
47 -- safely until everything is up and running, ie. command registration,
48 -- initial
49 -- data product updates. This procedure should be implemented to do these
50 -- things
51 -- if necessary.
52 overriding procedure Set_Up (Self : in out Instance) is null;
53 -----
54 -- Invokee connector primitives:
55 -----
56 -- This connector receives a 16-bit number synchronously that is used to
57 -- populate the outgoing packet.
58 overriding procedure Packed_U16_T_Recv_Sync (Self : in out Instance; Arg : in
59 -- Packed_U16.T);
60 -----
61 -- Invoker connector primitives:
62 -----
63 -- This procedure is called when a Packet_T_Send message is dropped due to a
64 -- full queue.
65 overriding procedure Packet_T_Send_Dropped (Self : in out Instance; Arg : in
66 -- Packet.T) is null;
67 -----
68 -- Definition of cycle function for task execution:
69 -----
70 -- This is an active component with no queue, so the
71 -- cycle function for the component's task must be
72 -- implemented here in the implementation class as
73 -- a user defined custom function.
74 overriding procedure Cycle (Self : in out Instance);
75
76 end Component.Background_Component.Implementation;

```

The first thing to notice is the definition of the protected type `Protected_Data`. This is an Ada entity that provides mutual exclusion to the data members that it protects. In this case, we are building a protected object around a two byte data field that will be used to populate the component's outgoing packet. The protected type provides two methods to access the protected data, a setter and a getter. This data must be protected since many components could call the `background_component`'s `Packed_U16_T_Recv_Sync` connector simultaneously. In addition, the `background_component` could be reading the data into the packet at the same time it is being set by an external component. To prevent the race condition and data corruption that could occur in these scenarios, the data is protected with a protected type, which ensures mutual exclusion.

In the component record, we can see that there is an instance of `Protected_Data` called `p_Data`. There is also a variable called `wake_Up_Time` which stores the next absolute time that the component should wake up and execute. This variable is initialized to the current time.

Now let's take a look at the implementation:

component-background_component-implementation.adb:

```

1 -----
2 -- Background_Component Component Implementation Body
3 -----

```

```

4
5 package body Component.Background_Component.Implementation is
6
7   -----
8   -- Protected type definition:
9   -----
10
10  protected body Protected_Data is
11    procedure Set_Data (New_Data : in Two_Byte_Array) is
12      begin
13        Data := New_Data;
14      end Set_Data;
15
16    function Get_Data return Two_Byte_Array is
17      begin
18        return Data;
19      end Get_Data;
20  end Protected_Data;
21
22   -----
23   -- Invokee connector primitives:
24   -----
25   -- This connector receives a 16-bit number synchronously that is used to
26   -- populate the outgoing packet.
27 overriding procedure Packed_U16_T_Recv_Sync (Self : in out Instance; Arg : in
28   -- Packed_U16.T) is
29 begin
30   -- Set the data safely using the protected object:
31   Self.P_Data.Set_Data (Packed_U16.Serialization.To_Byte_Array (Arg));
32 end Packed_U16_T_Recv_Sync;
33
34   -----
35   -- Definition of cycle function for task execution:
36   -----
37   -- This is an active component with no queue, so the
38   -- cycle function for the component's task must be
39   -- implemented here in the implementation class as
40   -- a user defined custom function.
41 overriding procedure Cycle (Self : in out Instance) is
42 begin
43   -- Delay until the wake up time period:
44   delay until Self.Wake_Up_Time;
45
46   -- Let's create a packet.
47 declare
48   -- Fill in the packet length with the stored packet length.
49   Pkt : Packet.T := (
50     Header => (
51       Time => Self.Sys_Time_T_Get,
52       Id => 0,
53       Sequence_Count => 0,
54       Buffer_Length => 2
55     ),
56     Buffer => [others => 0]
57   );
58 begin
59   -- Set the packet data:
60   Pkt.Buffer (0 .. 1) := Self.P_Data.Get_Data;
61
62   -- Send the packet.
63   Self.Packet_T_Send_If_Connected (Pkt);
64 end;

```

```

63      -- Calculate the next wake up time as 500ms after this wake up time.
64      Self.Wake_Up_Time := @ + Ada.Real_Time.Microseconds (500_000);
65
66  end Cycle;
67
68 end Component.Background_Component.Implementation;

```

First, the `Protected_Data` protected type body is shown. Again, these are simple setter and getter subprograms, but because they are protected, provide the necessary thread safety. We can also see the implementation `Packed_U16_T_Recv_Async` which simply copies the provided data into the protected object.

Next, is the definition of `Cycle`. Recall that this function will be called indefinitely in a loop by the component's task, see Section 6.4.1. First this function delays (sleeps) until the wake up time arrives. This allows other tasks to execute while this task is sleeping. When the wake up time arrives, this task will become ready to run and the scheduler will run it, so long as there are no other higher priority tasks currently running. When given time to execute, the component will then create a packet, copying the protected data into the packet's data field, and then send it out. Finally, the component computes the next wake up time. This execution cycle will repeat forever.

At any time during the components execution, an external call to `Packed_U16_T_Recv_Async` can change the contents of the data that is produced. There is no fear of data corruption since the shared data has been made protected.

We can make sure that this code compiles by running:

```
> redo build/obj/Linux/component-background_component-implementation.o
```

or by simply running `redo all`.

6.5.2 Unit Testing a Background Component

In this section we will create a unit test model to assist in unit testing the `background_component`. This section will not discuss the details of unit testing a component. It will only highlight the differences in unit testing a background component as compared to an active or queued component. You should be familiar with the concepts presented in Section 6.2 before reading this section.

To start, we create a unit test directory called `test/` under our component directory:

```
> mkdir test  # Make test/ within background_component/
> cd test
```

Next, we are going to create a component unit test model file `background_component.tests.yaml`. The contents are shown below:

```

1  ---
2  description: This is a set of unit tests for the Background Component.
3  tests:
4    - name: Test_Nominal
5      description: This test tests the nominal behavior of the component.

```

We define a single unit test that runs the component through a nominal scenario.

We also need to add an `env.py` file to this directory as we did in Section 6.2 to make sure we are using the `Linux_Test` target.

Next, we can generate the unit test templates and copy them into the test directory.

```
> redo build/template/component-background_component-implementation-tester.ads
> redo build/template/component-background_component-implementation-tester.adb
> redo build/template/background_component_tests-implementation.ads
> redo build/template/background_component_tests-implementation.adb
> redo build/template/test.adb
> cp build/template/* . # copy the template files into test/
```

Now the unit tests can be implemented by editing *background_component_tests-implementation.adb*. The implemented unit tests are shown below.

background_component_tests-implementation.adb:

```
1  -----
2  -- Background_Component Tests Body
3  -----
4
5  with Basic_Assertions; use Basic_Assertions;
6  with Packet.Assertion; use Packet.Assertion;
7
8  package body Background_Component_Tests.Implementation is
9
10   -----
11  -- Fixtures:
12  -----
13
14  overriding procedure Set_Up_Test (Self : in out Instance) is
15  begin
16    -- Allocate heap memory to component:
17    Self.Tester.Init_Base;
18
19    -- Make necessary connections between tester and component:
20    Self.Tester.Connect;
21
22    -- Call the component set up method that the assembly would normally call.
23    Self.Tester.Component_Instance.Set_Up;
24  end Set_Up_Test;
25
26  overriding procedure Tear_Down_Test (Self : in out Instance) is
27  begin
28    -- Free component heap:
29    Self.Tester.Final_Base;
30  end Tear_Down_Test;
31
32  -----
33  -- Tests:
34  -----
35
36  overriding procedure Test_Nominal (Self : in out Instance) is
37    -- Define the packet we are going to compare against.
38    A_Packet : Packet.T := (
39      Header => (
40        Time => (0, 0),
41        Id => 0,
42        Sequence_Count => 0,
43        Buffer_Length => 2
44      ),
45      Buffer => [others => 0]
46    );

```

```

47 begin
48     -- Put some data in the component's queue:
49     Self.Tester.Packed_U16_T_Send ((Value => 4));
50     Self.Tester.Packed_U16_T_Send ((Value => 6));
51
52     -- Expect no packets to be send out:
53     Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 0);
54
55     -- Tell the component to execute its "cycle" function, simulating the
56     -- execution of the component's task.
57     Self.Tester.Cycle_Component;
58
59     -- Expect one packet with a U16 in it:
60     Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 1);
61     A_Packet.Buffer (0) := 0;
62     A_Packet.Buffer (1) := 6;
63     Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (1),
64                         A_Packet);
65
66     -- Put some data in the component's queue:
67     Self.Tester.Packed_U16_T_Send ((Value => 15));
68
69     -- Tell the component to execute its "cycle" function, simulating the
70     -- execution of the component's task (twice).
71     Self.Tester.Cycle_Component;
72     Self.Tester.Cycle_Component;
73
74     -- Expect two more packets:
75     Natural Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 3);
76     A_Packet.Buffer (0) := 0;
77     A_Packet.Buffer (1) := 15;
78     Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (2),
79                         A_Packet);
80     Packet Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get (3),
81                         A_Packet);
82 end Test_Nominal;
83
84 end Background_Component_Tests.Implementation;

```

The unit test sends some data to the component through its synchronous receive connector. Note, that the act of sending this data to the component does not cause the component to execute. The component can only execute when its *Cycle* procedure is called. Recall that component unit testing in Adamant is single threaded to make the act of unit testing simpler. This means that an active component is not actually given a thread of execution during unit test. This is advantageous because the unit test developer can exactly specify when the component's internal task executes manually. In this way, very fine grain testing can be achieved.

To simulate the component's task executing, the tester component is autocoded with the subprogram *Cycle_Component* which calls the component's *Cycle* procedure directly, simulating the running of the task. The advantage is that the programmer can call this *Cycle_Component* procedure at the exact moment they want the component to run instead of dealing with the complexities of managing the execution of a multithreaded unit test environment.

Shown in the test, data is passed to the component on its synchronous connector. After that a check is made to ensure that no packets were produced by the component yet. Next, the *Cycle_Component* procedure is called, causing the component to "execute" which causes a single packet to be produced. The unit test ensures that the packet's data is correct and then repeats the procedure again, ensuring that a packet is produced for each call to *Cycle_Component*.

The unit test can be compiled and run using the command:

```
> redo test
```

which produces the output

```
OK Test_Nominal
```

```
Total Tests Run: 1
Successful Tests: 1
Failed Assertions: 0
Unexpected Errors: 0
```

6.6 Protected Objects within Components

Ada uses *protected objects* to provide mutual exclusion and task synchronization for multi-tasked systems. In Adamant, protected objects should be used within a component to protect shared data that could be accessed simultaneously by more than a single task. Protected objects should ALWAYS be used in this case, even if the shared data is a single byte or a single word that you assume is atomically updated. The only way to ensure mutual exclusion and produce platform independent code is to use protected objects. The implementation of protected objects is via code insertion by the compiler, and thus is very performant. One should not hesitate to employ the use of a protected object when it is required.

The following component patterns require the use of a protected object:

- A component has a connector of kind `recv_sync` that can be called by multiple outside components. Any internal data that is updated in this connector handler must be made protected.
- A component has a connector of kind `recv_sync` that writes to data that could be read by the component on another thread of execution. This shared data must be made protected.
- A component has a connector of kind `return` or `modify` that returns data that could be written to on another thread of execution. This shared data must be made protected.

The use of protected objects within Adamant components is not as common as might be assumed by the list above. The Adamant internal queue is itself a protected object. The use of `recv_async` connectors can often bypass the need to create a user-defined protected object. For more information on the component queue see Section 6.3.1.

It should also be noted that user-defined protected objects are never used for task synchronization within Adamant. Adamant component queues provide the sole mechanism for task synchronization. This keeps the execution and interaction of components easy to understand by simply looking at component diagrams.

An example of defining and using a protected object within a component is presented in Section 6.5.1.

6.7 Component Initialization

Components in Adamant get initialized in phases. The initialization of components within each phase is order independent, however the ordering of each phase itself is important. Component initialization is provided by assembly level autocode, however, what gets initialized within each component is defined by each component's model. The different phases of component initialization are summarized below.

1. **Component Instantiation** - Every component is instantiated from a component type. Com-

ponents may specify a *discriminant* in their component models to specify parameters to be used during instantiation.

2. **Component Base Initialization** - A `Init_Base` procedure is autocoded for any component that requires heap allocation for base package objects. This includes queues and unconstrained arrayed connectors.
3. **Component Set ID Bases** - A `Set_Id_Bases` procedure is autocoded for any component that includes an IDed Entity Suite, see Section 6.8. This procedure is used to set the base ID for each entity suite.
4. **Component Implementation Initialization** - Components may specify an `Init` procedure with custom parameters in their model. This procedure is called in order to initialize any implementation package variables. This can include allocations on the heap, saving off input parameters, etc.
5. **Component Set Up** - Components are not allowed to make connector calls during any of the initialization phases described above, since it is not safe to assume that an invoked component has completed its initialization. A `Set_Up` procedure is provided for all components in order to provide a safe initialization phase to call connectors. By default, the implementation of this method is `null`, however it is commonly overridden to publish data products at startup or other perform other connector invocations that must occur at boot.

The following sections will go into detail on each of the initialization phases and provide an example component that demonstrates each feature.

6.7.1 Component Instantiation

By default, every component is initialized without parameters. Parameters may be provided at instantiation using the *discriminant*. The *discriminant* is an Ada language feature that allows parameterization of a record type.

Not all Ada types can be supplied as a discriminant parameter, and the list is even more restricted when using the Ravenscar profile. However, in Adamant, discriminants are commonly used to provide static values such as task priorities or interrupt identifiers. It is also common to pass access types to data that parameterize the behavior of the component. Note that lengths cannot be passed into the discriminant and used to size an array (or other such operations) as this causes implicit heap allocation, which is forbidden by the Ravenscar profile. Parameters included in the discriminant can be accessed in component implementation code just like any other field of the `Instance` record.

A discriminant can be defined in a component model as follows:

discriminated_component.component.yaml

```
1  ---
2  execution: passive
3  description: This is an example component with a discriminant provided.
4  # Optional - discriminant declaration
5  discriminant:
6  # Optional - discriminant description
7  description: This is the component's discriminant. It initializes parameters
8  ↪ during instantiation.
9  # Optional - list of parameters to pass to component's discriminant
10 parameters:
11   # Required - parameter name
12   - name: packets_Per_Tick
13   # Required - parameter type
14   type: Natural
15   # Optional - puts "not null" in front of the type. Defaults to false, and
     ↪ only should be used for access types
     not_null: false
```

```

16    # Optional - parameter description
17    description: The number of packets to send every time the Tick_T_Recv_Sync
18      ↳ connector is invoked.
19    - name: enabled_At_Startup
20      type: Boolean
21      description: If True, packets will be produced for every call to
22        ↳ Tick_T_Recv_Sync. If False, no packets will be produced.
23    connectors:
24      - type: Tick.T
25        kind: recv_sync
26        description: This connector provides the schedule tick for the component.
27      - type: Packet.T
28        kind: send
29        description: This connector is used to send out a telemetry packet.

```

In this model we define the component's discriminant with the discriminant tag. Note that all fields are commented to indicate whether they are *required* or *optional*.

This component is discriminated with two parameters, `Packets_Per_Tick`, which specifies the number of packets the component produces every time it receives a `Tick.T`, and `Enabled_At_Startup`, which defines whether or not the component produces packets at all when first initialized.

Let's take a look at how this model is reflected in the component autocode. We generate the component template specification file with the following commands:

```
> redo build/template/component-discriminated_component-implementation.ads
> cp build/template/* .    # copy template into component dir
```

Here is what the specification file looks like:

component-discriminated_component-implementation.ads:

```

1  -----
2  -- Discriminated_Component Component Implementation Spec
3  -----
4
5  -- Includes:
6  with Tick;
7
8  -- This is an example component with a discriminant provided.
9  package Component.Discriminated_Component.Implementation is
10
11    -- The component class instance record:
12    -- This is the component's discriminant. It initializes parameters during
13    -- instantiation.
14
15    -- Discriminant Parameters:
16    -- Packets_Per_Tick : Natural - The number of packets to send every time the
17    -- Tick_T_Recv_Sync connector is invoked.
18    -- Enabled_At_Startup : Boolean - If True, packets will be produced for every
19      ↳ call
20      ↳ to Tick_T_Recv_Sync. If False, no packets will be produced.
21
22    type Instance (Packets_Per_Tick : Natural; Enabled_At_Startup : Boolean) is
23      ↳ new Discriminated_Component.Base_Instance with private;
24
25  private

```

```

25  -- The component class instance record:
26  -- This is the component's discriminant. It initializes parameters during
27  -- instantiation.
28  --
29  -- Discriminant Parameters:
30  -- Packets_Per_Tick : Natural - The number of packets to send every time the
31  -- Tick_T_Recv_Sync connector is invoked.
32  -- Enabled_At_Startup : Boolean - If True, packets will be produced for every
33  -- call
34  -- to Tick_T_Recv_Sync. If False, no packets will be produced.
35  --
36  type Instance (Packets_Per_Tick : Natural; Enabled_At_Startup : Boolean) is
37    new Discriminated_Component.Base_Instance with record
38      null; -- TODO
39    end record;
40  --
41  -----
42  -- Set Up Procedure
43  -----
44  -- Null method which can be implemented to provide some component
45  -- set up code. This method is generally called by the assembly
46  -- main.adb after all component initialization and tasks have been started.
47  -- Some activities need to only be run once at startup, but cannot be run
48  -- safely until everything is up and running, ie. command registration,
49  -- initial
50  -- data product updates. This procedure should be implemented to do these
51  -- things
52  -- if necessary.
53  overriding procedure Set_Up (Self : in out Instance) is null;
54  --
55  -----
56  -- Invokee connector primitives:
57  -----
58  -- This connector provides the schedule tick for the component.
59  overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in
60    Tick.T);
61  --
62  -- Invoker connector primitives:
63  -----
64  -- This procedure is called when a Packet_T_Send message is dropped due to a
65  -- full queue.
66  overriding procedure Packet_T_Send_Dropped (Self : in out Instance; Arg : in
67    Packet.T) is null;
68
69 end Component.Discriminated_Component.Implementation;

```

Notice the definition of the component record `Instance` now includes the two discriminant parameters `Packets_Per_Tick` and `Enabled_At_Startup`. To instantiate the component these parameters now must be provided. This is done in an Adamant assembly model, but the autocode that instantiates the component would look something like this:

```

1  Discriminated_Component_Instance := Component.Discriminated_Component.Instance (
2    Packets_Per_Tick => 3,
3    Enabled_At_Startup => True
4  );

```

Note that the discriminant parameters can be accessed inside of the component implementation pack-

age by simply referencing them via `Self.Packets_Per_Tick` or `Self.Enabled_At_Startup`, as if they were normal record fields.

Similar functionality to a discriminant can be provided with the implementation package `Init` function discussed in Section 6.7.5. You should use an `Init` function instead of a discriminant if you need to use the parameters to allocate memory on the heap or provide additional computation at startup. Using the discriminant should be considered when a variable is so essential to the functioning of a component that it is required for the component to even be instantiated. One good example of this case is an interrupt handling component, which could not function without an interrupt ID. For handling interrupts specifically, see Section 6.18.2.

6.7.2 Component Base Initialization

The component base package is autocoded with an `Init_Base` procedure if any of the following are true:

- The component has at least one `recv_async` connector, which requires the component have an internal queue.
- The component has at least one unconstrained arrayed connector.

In either case, variable sized allocations need to be made on the heap. This is provided via a call to the component base package `Init_Base` procedure. Below is an example component model that satisfies both of the conditions above:

`init_base_component.component.yaml`

```
1  ---
2  execution: passive
3  description: This is an example component with an base package initialization.
4  connectors:
5    - type: Tick.T
6      # Any component with an asynchronous connector requires an internal queue.
7      # The sizing of this queue (in bytes) is done during the initialization
8      # of the base package in the init_base procedure.
9      kind: recv_async
10     description: This connector provides the schedule tick for the component.
11    - type: Packet.T
12      kind: send
13      description: This connector is used to send out a telemetry packet.
14      # A count of zero specifies this connector as "unconstrained" it must
15      # be constrained at runtime by allocating its size on the heap during
16      # initialization. This is achieved in the init_base procedure.
17      count: 0
```

In this model we define the component with `recv_async` connector for `Tick.T` and an unconstrained arrayed invoker connector for `Packet.T`. The first requires the allocation of the internal component queue, the second requires the allocation of the connector array.

Let's take a look at how this model is reflected in the component base package autocode. We generate the component base package specification with the following command:

```
> redo build/src/component-init_base_component.ads
```

The specification is quite large, so is not shown here. But the reader can verify that there is an autocoded public subprogram in `component-init_base_component.ads` that looks like:

```

1  not overriding procedure Init_Base (Self : in out Instance; Queue_Size : in,
→  Natural; Packet_T_Send_Count : in Natural);

```

This procedure is called by the assembly autocode in the following way:

```

1  Init_Base_Component_Instance.Init_Base (
2    Queue_Size => 1000,           -- bytes
3    Packet_T_Send_Count => 3
4 );

```

6.7.3 Component Set ID Bases

A `Set_Id_Bases` procedure gets autocoded into a component base package if the component contains any `IDed` entity model, see Section 6.8. For example, let's modify the example component in Section 6.1 to have two `IDed` entity models, one for events and another for commands. In this case, the autocoded base package specification could be generated with the following command:

```
> redo build/src/component-example_component.ads
```

The specification is quite large, so is not shown here. But the reader can verify that there is an autocoded public subprogram in `component-example_component.ads` that looks like:

```

1  not overriding procedure Set_Id_Bases (Self : in out Instance; Command_Id_Base :
→  in Command_Types.Command_Id_Base; Event_Id_Base : in
→  Event_Types.Event_Id_Base);

```

Notice that there are two parameters, each which sets the base identifier for the `IDed` entity. For example, if `Command_Id_Base` was set to 17, then the component's commands would start at 17. If the component had 3 commands, the IDs for those three commands would be 17, 18, and 19. In this way, the user can set the ID space for each component in the system. By default, Adamant will autocode the calls to these functions to produce the most compact ID space possible, with one component's ID set starting right at the end of the previous component's ID set. However, the assembly model can be used to override this behavior, and assign IDs manually. For more details see Section 7.3.

This procedure is called by the assembly autocode in the following way:

```

1  Example_Component_Instance.Set_Id_Bases (
2    Command_Id_Base => 17,
3    Event_Id_Base => 1
4 );

```

6.7.4 Component Map Data Dependencies

A `Map_Data_Dependencies` procedure gets autocoded into a component base package if the component contains a data dependency model, see Section 6.11. For example, let's look at the component

presented in Section 6.11, which has two data dependencies defined: Count and Temperature. In this case, the autocoded base package specification could be generated with the following command:

```
> redo build/src/component-data_dependency_component.ads
```

The specification is quite large, so is not shown here. But the reader can verify that there is an autocoded public subprogram in *component-data_dependency_component.ads* that looks like:

```
1 not overriding procedure Map_Data_Dependencies (
2     Self : in out Instance;
3     Counter_Id : in Data_Product_Types.Data_Product_Id;
4     Counter_Stale_Limit : in Ada.Real_Time.Time_Span;
5     Temperature_Id : in Data_Product_Types.Data_Product_Id;
6     Temperature_Stale_Limit : in Ada.Real_Time.Time_Span
7 );
```

Notice that there are four parameters, two which set the IDs of the dependencies and two which set the *stale limits*. The IDs must correspond to data products of the same type produced elsewhere in the system. During the construction of the assembly, these IDs are inserted automatically based on the data products specified in the assembly model, which is discussed in Section 7.3. The *stale limits* are of the Ada Time_Span type and are used to determine when a data dependency is too old to use. When fetched, a data dependency's timestamp is compared to a reference time (usually the current time) minus the *stale limit*. If the data dependency's timestamp is found to be earlier, then it is deemed stale. Sometimes a data dependency can never go stale, such as a system mode that is only updated upon change. To indicate this, the stale limit parameter must be set to zero, as shown in the example below.

It is important to understand that this procedure is generated when a data dependency model is present, and must be called by the assembly at initialization in the following way:

```
1 Data_Dependency_Component_Instance.Map_Data_Dependencies (
2     Counter_Id => 22,
3     -- The counter can never be stale. Set to zero to indicate this:
4     Counter_Stale_Limit => Ada.Real_Time.Time_Span_Zero,
5     Temperature_Id => 11,
6     -- The temperature is stale if more than 5 seconds old:
7     Temperature_Stale_Limit => Ada.Real_Time.Microseconds (5_000_000)
8 );
```

6.7.5 Component Implementation Initialization

Adamant provides a way to write a custom initialization procedure for a component's implementation package. The name of this procedure is always called `Init` and is modeled in a component model in a similar manner to the *discriminant*. Custom initialization procedures are commonly used to allocate component data structures on the heap, provide initial configuration values for component variables, or to perform startup calculations before the assembly tasks begin. Note, if you need to call connectors in your initialization procedure you should use the `Set_Up` procedure instead, see Section 6.7.6.

The following example shows how to configure a component with an `Init` procedure. It is very similar to the example shown for configuring a component with a *discriminant* in Section 6.7.1. In fact, either a discriminant or an initialization procedure may be used to accomplish similar goals. In general, you should use an `Init` function instead of a discriminant if you need to use the parameters to allocate memory on the heap or provide additional computation at startup. Both operations can only be achieved using a subprogram call, and cannot be achieved statically in a discriminant. Using the discriminant should be considered when a static variable is so essential to the functioning of a component that it is required for the component to even be instantiated. One good example of this case is an interrupt handling component, which could not function without an interrupt ID. For handling interrupts specifically, see Section 6.18.2. Sometimes it makes sense to have both a discriminant and an initialization function. Use your best judgement, but, in general, an initialization function provides more capability so is more often employed.

An `Init` procedure can be defined in a component model as follows:

initialized_component.component.yaml

```

1  ---
2  execution: passive
3  description: This is the example component, with an Init procedure.
4  # Optional - init declaration
5  init:
6  # Optional - init description
7  description: This is the component's init procedure. It initializes parameters
8  ↪ during initialization.
9  # Optional - list of parameters to pass to component's init
10 parameters:
11   # Required - parameter name
12   - name: Packets_Per_Tick
13   # Required - parameter type
14   type: Natural
15   # Optional - puts "not null" in front of the type. Defaults to false, and
16   ↪ only should be used for access types
17   not_null: false
18   # Optional - parameter description
19   description: The number of packets to send every time the Tick_T_Recv_Sync
20   ↪ connector is invoked.
21   - name: Enabled_At_Startup
22   type: Boolean
23   description: If True, packets will be produced for every call to
24   ↪ Tick_T_Recv_Sync. If False, no packets will be produced.
25   # Optional - default value
26   default: "True"
27 connectors:
28   - type: Tick.T
29   kind: recv_sync
30   description: This connector provides the schedule tick for the component.
31   - type: Packet.T
32   kind: send
33   description: This connector is used to send out a telemetry packet.

```

In this model we define the component's initialization procedure with the `init` tag. Note that all fields are commented to indicate whether they are *required* or *optional*.

This component is initialized with two parameters, `packets_Per_Tick`, which specifies the number of packets the component produces every time it receives a `Tick.T`, and `enabled_At_Startup`, which defines whether or not the component produces packets at all when first initialized. We have provided a default value for `enabled_At_Startup` which is "True". Note that an `Init` function may also be modeled that has no parameters. This might be useful if the component needs to run some special code at startup, but requires no parameters to do so.

Let's take a look at how this model is reflected in the component autocode. We generate the component template specification file with the following commands:

```
> redo build/template/component-initialized-implementation.ads
> redo build/template/component-initialized-implementation.adb
> cp build/template/* . # copy template into component dir
```

Here is what the specification file looks like. It has been modified to include some internal variables that will be set in our new initialization procedure:

component-initialized_component-implementation.ads:

```

1  -----
2  -- Initialized_Component Component Implementation Spec
3  -----
4
5  -- Includes:
6  with Tick;
7
8  -- This is the example component, with an Init procedure.
9  package Component.Initialized_Component.Implementation is
10
11    -- The component class instance record:
12    type Instance is new Initialized_Component.Base_Instance with private;
13
14    -----
15    -- Subprogram for implementation init method:
16    -----
17    -- This is the component's init procedure. It initializes parameters during
18    -- initialization.
19    --
20    -- Init Parameters:
21    -- packets_Per_Tick : Natural - The number of packets to send every time the
22    --   Tick_T_Recv_Sync connector is invoked.
23    -- enabled_At_Startup : Boolean - If True, packets will be produced for every
24    --   call to Tick_T_Recv_Sync. If False, no packets will be produced.
25    --
26    overriding procedure Init (Self : in out Instance; Packets_Per_Tick : in
27      Natural; Enabled_At_Startup : in Boolean := True);
28
29  private
30
31    -- The component class instance record:
32    type Instance is new Initialized_Component.Base_Instance with record
33      Packets_Per_Tick : Natural := 0;
34      Enabled_At_Startup : Boolean := False;
35      Pkt : Packet.T;
36    end record;
37
38    -----
39    -- Invokee connector primitives:
40    -----
41    -- This connector provides the schedule tick for the component.
42    overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in
43      Tick.T);
44
45    -----
46    -- Invoker connector primitives:
47    -----
```

```

43    -- This procedure is called when a Packet_T_Send message is dropped due to a
44    -- full queue.
45    overriding procedure Packet_T_Send_Dropped (Self : in out Instance; Arg : in
46    -- Packet.T) is null;
47
48 end Component.Initialized_Component.Implementation;

```

Notice the public definition of the `Init` procedure which includes the two parameters `packets_Per_Tick` and `enabled_At_Startup`, which is set to the default value `True`. The definition for this procedure is shown in the package body:

component-initialized_component-implementation.adb:

```

1  -----
2  -- Initialized_Component Component Implementation Body
3  -----
4
5  with Interfaces; use Interfaces;
6
7 package body Component.Initialized_Component.Implementation is
8
9  -----
10   -- Subprogram for implementation init method:
11   -----
12   -- This is the component's init procedure. It initializes parameters during
13   -- initialization.
14   --
15   -- Init Parameters:
16   -- packets_Per_Tick : Natural - The number of packets to send every time the
17   -- Tick_T_Recv_Sync connector is invoked.
18   -- enabled_At_Startup : Boolean - If True, packets will be produced for every
19   -- call to Tick_T_Recv_Sync. If False, no packets will be produced.
20   --
21   overriding procedure Init (Self : in out Instance; Packets_Per_Tick : in
22   -- Natural; Enabled_At_Startup : in Boolean := True) is
23 begin
24     -- Make sure this value is not too large!
25     pragma Assert (Packets_Per_Tick < 10);
26
27     -- Initialized the component instance variables:
28     Self.Packets_Per_Tick := Packets_Per_Tick;
29     Self.Enabled_At_Startup := Enabled_At_Startup;
30
31     -- Other computations can be performed here too!
32     -- Let's set the packet data to 5 + the packets_Per_Tick
33     Self.Pkt.Header.Buffer_Length := 7;
34     Self.Pkt.Buffer := [others => 5 + Unsigned_8 (Self.Packets_Per_Tick)];
35
36   end Init;
37
38   -----
39   -- Invokee connector primitives:
40   -----
41   -- This connector provides the schedule tick for the component.
42   overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in
43   -- Tick.T) is
44     Ignore : Tick.T renames Arg;
45
46 begin
47   -- Send out the configured number of packets every time
48   -- a tick is received.
49   for Idx in 1 .. Self.Packets_Per_Tick loop
50     Self.Packet_T_Send (Self.Pkt);

```

```

44     end loop;
45 end Tick_T_Recv_Sync;
46
47 end Component.Initialized_Component.Implementation;

```

We can see that all three of the component's internal variables are initialized by the `Init` procedure, and some basic computation and checking was also implemented.

To initialize the component, these parameters now must be provided. This is done in an Adamant assembly model, but the autocode that initializes the component would look something like this:

```

1 Initialized_Component_Instance.Init (
2     Packets_Per_Tick => 3,
3     Enabled_At_Startup => True -- May be omitted since a default is provided
4 );

```

6.7.6 Component Set Up

Components are not allowed to make connector calls during any of the initialization phases described in the previous sections, since it is not safe to assume that an invoked component has completed its initialization. A connector call could result in accessing data structures that have not yet been allocated or even calling connectors that are not yet connected. A `Set_Up` procedure is provided for all components in order to provide a safe initialization phase to call connectors. The `Set_Up` procedure is called after all other component initialization has completed.

The `Set_Up` procedure is defined in the Adamant core package for a component, `src/core/component`. The subprogram declaration looks like:

```

1 procedure Set_Up (Self : in out Instance) is null;

```

By default, this subprogram is defined as null, and so does nothing. However, a developer may override this subprogram in their implementation package in order to make connector calls on startup. A few common uses for this function are grabbing the system time at startup or emitting a data product at startup with its boot value. If a component has no use for this procedure, then it can be ignored.

6.8 Component IDed Entities

Adamant provides domain specific modeling capabilities for common data types used in spacecraft flight software. The supported capabilities are defined below:

- **Events** - Small asynchronously created packets that signify some "happening" has occurred in the software. These are usually produced in response to actions that the software takes as a result of stimulus from the outside world. As a result, they are often produced sporadically.
- **Data Products** - Small packets that contain data from onboard sensors, software state variables, etc. These are usually produced on a periodic schedule and are meant to reflect the state of the system.
- **Data Dependencies** - Data dependencies are data products from external components that need to be consumed/used/checked in order for a component to function.

- **Packets** - Similar to *data products*, but usually larger. Often, a packet contains many *data products* within. *Packets* are also often used for external communication.
- **Commands** - Packets created by an external system that are received by the embedded software and “executed”. *Commands* are used to direct the software to perform certain actions at certain times.
- **Parameters** - Values that are used to configure the system as a whole. Parameters are often stored in a single location on the embedded system, and can be changed by an external system to reconfigure it.
- **Faults** - Similar to events, these are generated as a result of the system detecting an anomalous condition. *Faults* usually cause the system to engage in a recovery or *safing* action.

As implemented in Adamant, these concepts share many features and are classified as *IDed entities*. In particular, each of these entities can be thought of as packets of data that each share the following features:

- **ID** - The unique identifier for the entity.
- **Length** - The number of bytes used to hold data associated with the entity.
- **Buffer** - A statically sized generic byte array buffer for holding the entity’s specific data.

Each of the entities above has a distinct ID space. No two items in each ID space may share an ID. So for example, an event and a command in the system may share the same ID, but their IDs should never be compared because they are in different ID spaces. Two different command should never share the same ID.

Most of the entities above also include a single unique *type* that is associated with their ID. This type is almost always a *packed type*, see Section 5, and is serialized into the generic buffer. In this way, every event, command, data product, etc. in the system can have a unique data type associated with their unique ID. However, because all of these data types are serialized into a generic buffer, entities can be passed around easily through Adamant connectors as elements of the entity type. For example, it is common in Adamant to see many components with a connector of type `Data_Product.T`. These data products can often be handled uniformly by downstream components that process them, keeping the code generic and simple, even through the specific type of each data product is completely unique. The benefit of this property will be explored in the following sections.

In Adamant, each of these IDed entities is implemented at the component level. Specifically, a component can create its own events, data products, packets, and faults, execute its own commands, and be configured by its parameters. In fact, each of these entity types has its own unique model file that can be *associated* with a component. In this way, commands can be modeled and then assigned to a component for execution or parameters can be modeled and then used within a component for configuration.

IDs are assigned dynamically at initialization though each component’s `Set_Id_Bases` procedure, see Section 6.7.3. When a group of components are combined into an assembly, Adamant ensures that there are no duplicate IDs from all the aggregated events, data products, commands, etc. By default, Adamant will automatically assign IDs to these entities at startup with as compact an ID space as possible. This usually means assigning IDs from 0 (or sometimes 1) to the maximum number of that entity found in the assembly. The ID space can be assigned and configured manually in the assembly model as well. See Section 6.7.3.

The following sections will walk through tutorials of how to model and use these entities within a component.

6.9 Component Events

Events are component *IDed entities*. Please read Section 6.8 before continuing on to this section.

An *event* in Adamant is a small asynchronously created packet that signify some “happening” has occurred. Events are usually produced in response to actions that the software takes as a result of stimulus from the outside world. As a result, they are often produced sporadically.

6.9.1 The Event Type

The event type in Adamant is defined in *src/core/types/event*. An event consists of a header followed by a byte array used to serialize the specific data, parameters, associated with an event. An event is a *variable length packed record*, see Section 5.1.11, whose length is specified by the length field in its header. The following tables show the bit layout of an event and its header.

Generic event packet for holding arbitrary events

Table 11: Event _ Header Packed Record : 88 bits

Name	Type	Range	Size (Bits)	Start Bit	End Bit
Time	Sys_Time.T	-	64	0	63
Id	Event_Types.Event_Id	0 to 65535	16	64	79
Param_Buffer_Length	Event_Types.Parameter_Buffer_Length_Type	0 to 32	8	80	87

Field Descriptions:

- **Time** - The timestamp for the event.
- **Id** - The event identifier
- **Param_Buffer_Length** - The number of bytes used in the param buffer

Generic event packet for holding arbitrary events

Table 12: Event Packed Record : 344 bits (*maximum*)

Name	Type	Range	Size (Bits)	Start Bit	End Bit	Variable Length
Header	Event_Header.T	-	88	0	87	-
Param_Buffer	Event_Types.Parameter_Buffer_Type	-	256	88	343	Header.Param_Buffer_Length

Field Descriptions:

- **Header** - The event header
- **Param_Buffer** - A buffer that contains the event parameters

As can be seen, in the event header there is a 64-bit timestamp which signifies when the event was created followed by a unique identifier for the event. There is also a byte array where the event’s parameters are serialized.

6.9.2 Creating an Event Model

In Adamant, events are produced by components. A component may produce many different events. To demonstrate this we will create a component called the *event_component* using the following commands:

```
> mkdir event_component # make component directory  
> cd event_component  
> touch .all_path          # add directory to build path
```

Next we will create the component by writing the following YAML file in this directory:

event_component.component.yaml:

```
1  ---  
2  execution: passive  
3  description: This is the event component, which sends events.  
4  connectors:  
5    - type: Tick.T  
6      kind: recv_sync  
7      description: This connector provides the schedule tick for the component.  
8    - return_type: Sys_Time.T  
9      kind: get  
10     description: This connector is used to fetch the current system time.  
11    - type: Event.T  
12      kind: send  
13      description: This connector is used to send out an event.
```

which looks like this:



To configure the component with events we use an event model file. To create an event model for this component, we simply need to name the event model appropriately. Event models will always be of the form *component_name.events.yaml* where *component_name* is the component that will be producing the events. Only a single event model can be configured for a component, and a single event model cannot be shared by more than one component. In this directory we create an event model named *event_component.events.yaml*. The contents are shown below:

event_component.events.yaml:

```
1  ---  
2  # Optional - description of event suite  
3  description: A set of events for the Event Component.  
4  # Required - list of events included in the event suite  
5  events:  
6    # Required - the name of the event  
7    - name: Tick_Received  
8      # Optional - a description of the event  
9      description: A tick was received by the component.  
10     # Optional - the parameter type to include with the event  
11     param_type: Tick.T  
12    - name: Ten_More_Ticks_Received  
13      description: This event is produced for every 10 ticks and includes the  
→ total number of ticks received by the component.
```

```

14     param_type: Packed_U16.T
15   - name: First_Tick_Received
16     description: This event is produced only when the first tick is received.

```

This model defines the *event suite* for the *event component*. It declares 3 different events that can be sent by the component. Comments are provided to explain whether or not each field is optional or required. In this case, you must supply a name for each event, and optionally a parameter that will be serialized with the event. Only a single parameter type can be provided with an event. However, that parameter can be a compound type such as an array or record. Parameters should always be *packed types*, see Section 5, since their bit layout is platform independent. This allows development machines to easily decode event parameters no matter what machine is running the Adamant embedded system.

The presence of an event model causes the Adamant build system to produce many more build rules (some output omitted for brevity):

```

> redo what
redo what
redo all
redo build/html/event_component_events.html
redo build/obj/Linux/event_component_events-representation.o
redo build/obj/Linux/event_component_events.o
redo build/src/event_component_events-representation.adb
redo build/src/event_component_events-representation.ads
redo build/src/event_component_events.adb
redo build/src/event_component_events.ads

```

We can see two object files that can be constructed, the event suite package, *build/obj/Linux/event_component_events.o*, and the event string representation package, *build/obj/Linux/event_component_events-representation.o*, which will be discussed in Section 6.9.5. To build the event suite package autocode we can run:

```
> redo build/obj/Linux/event_component_events.o
```

which will construct the package *Event_Component_Events* in *build/src/event_component_events.ads* and *build/src/event_component_events.adb*. Let's look at the specification.

event_component_events.ads:

```

1 -----
2 -- Event_Component_Events Spec
3 --
4 -- Generated from event_component.events.yaml on 2025-07-15 21:47.
5 -----
6
7 -- Standard Includes:
8 with Event;
9 with Event_Types;
10 with Sys_Time;
11
12 -- Parameter Includes
13 with Tick;
14 with Packed_U16;
15
16 -- A set of events for the Event Component.
17 package Event_Component_Events is
18

```

```

19   -- Object instance type:
20   type Instance is tagged limited private;
21
22   -----
23   -- Local Event Identifiers:
24   -----
25   Num_Events : constant Natural := 3;
26   type Local_Event_Id_Type is (
27     Tick_Received_Id,
28     Ten_More_Ticks_Received_Id,
29     First_Tick_Received_Id
30   );
31   for Local_Event_Id_Type use (
32     Tick_Received_Id => 0,
33     Ten_More_Ticks_Received_Id => 1,
34     First_Tick_Received_Id => 2
35   );
36
37   -----
38   -- Setter procedure for event ID base:
39   -----
40   not overriding function Get_Id_Base (Self : in Instance) return
41     Event_Types.Event_Id
42     with Inline => True;
43   not overriding procedure Set_Id_Base (Self : in out Instance; Id_Base : in
44     Event_Types.Event_Id)
45     with Inline => True;
46
47   -----
48   -- Getter function for global Event IDs:
49   -----
50   not overriding function Get_Tick_Received_Id (Self : in Instance) return
51     Event_Types.Event_Id
52     with Inline => True;
53   not overriding function Get_Ten_More_Ticks_Received_Id (Self : in Instance)
54     return Event_Types.Event_Id
55     with Inline => True;
56   not overriding function Get_First_Tick_Received_Id (Self : in Instance)
57     return Event_Types.Event_Id
58     with Inline => True;
59
60   -----
61   -- Event creation functions:
62   -----
63   -- A tick was received by the component.
64   not overriding function Tick_Received (Self : in Instance; Timestamp :
65     Sys_Time.T; Param : in Tick.T) return Event.T;
66
67   -- This event is produced for every 10 ticks and includes the total number
68   -- of
69   -- ticks received by the component.
70   not overriding function Ten_More_Ticks_Received (Self : in Instance;
71     Timestamp : Sys_Time.T; Param : in Packed_U16.T) return Event.T;
72
73   -- This event is produced only when the first tick is received.
74   not overriding function First_Tick_Received (Self : in Instance; Timestamp :
75     Sys_Time.T) return Event.T;
76
77   -- Compile time checks to make sure types do not serialize longer than the
78   -- event buffer size:
79   pragma Warnings (Off, "condition can only be True if invalid values
80   -- present");

```

```

70  pragma Compile_Time_Error (
71      Tick.Size_In_Bytes > Event_Types.Parameter_Buffer_Type'Length,
72      "Event 'Tick_Received' has argument of type 'Tick.T' which has a maximum
73      → serialized length larger than the buffer size of Event.T."
74  );
74  pragma Compile_Time_Error (
75      Packed_U16.Size_In_Bytes > Event_Types.Parameter_Buffer_Type'Length,
76      "Event 'Ten_More_Ticks_Received' has argument of type 'Packed_U16.T' which
77      → has a maximum serialized length larger than the buffer size of
78      → Event.T."
77  );
78  pragma Warnings (On, "condition can only be True if invalid values present");
79 private
80     type Instance is tagged limited record
81         Id_Base : Event_Types.Event_Id := 0;
82     end record;
83
84 end Event_Component_Events;

```

First we can see that there is a `Instance` type which is the event suite tagged type. There is also a public `local_Event_Id_Type` enumeration which declares the *local IDs* for the events. When instantiated in a component, these local IDs will be added to the component's *event ID base*. The event ID base can be set through the public function `set_Id_Base` which is called during component initialization, see Section 6.7.3. Note that the ID base for the event suite is stored in the `Instance` record in the private section in the `id_Base` variable. Next, there is a set of 3 functions which return the ID for each event in the event suite. The returned ID will already have the ID base offset applied to the local ID. Finally, there is a set of three functions that are used to create the three events.

The most commonly used functions are the event creation functions. Each function returns a type of `Event.T` and takes a timestamp and the event parameter type as input. The implementation of these functions are not shown here but can be explored by the reader in `event_component_events.adb`. These functions take the timestamp and event parameter and serialize them into the event header and buffer, respectively. The event is returned with the correct ID applied. To see how these functions are used within a component continue to the next section.

6.9.3 Using the Event Package

When an event model is found for a component, the component base package autocode is altered slightly to include a variable called `events` which is an instantiation of the event suite package instance, `Event_Component_Events.Instance`. The reader can verify this by building and inspecting `build/src/component-event_component.ads`. This `events` variable can then be used in the component implementation package to create events.

The following is the hand-coded implementation package body for the `event_component`. For directions on generating the templates for this file see Section 6.1.4.

`component-event_component-implementation.adb`:

```

1 -----
2 -- Event_Component Component Implementation Body
3 -----
4
5 package body Component.Event_Component.Implementation is
6
7     -----
8     -- Invokee connector primitives:
9     -----
10    -- This connector provides the schedule tick for the component.
11    overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in
12        → Tick.T) is

```

```

12      -- Get the timestamp:
13      Timestamp : constant Sys_Time.T := Self.Sys_Time_T_Get;
14  begin
15      -- Send the first-tick event once:
16      if not Self.First_Event_Sent then
17          Self.Event_T_Send (Self.Events.First_Tick_Received (Timestamp));
18          Self.First_Event_Sent := True;
19      else
20          -- Send the every-ten-ticks event every ten ticks:
21          if (Self.Count mod 10) = 0 then
22              Self.Event_T_Send (Self.Events.Ten_More_Ticks_Received (Timestamp,
23                  → (Value => Self.Count)));
24          end if;
25      end if;
26
27      -- Send the received event every time:
28      Self.Event_T_Send (Self.Events.Tick_Received (Timestamp, Arg));
29
30      -- Increment the count:
31      Self.Count := @ + 1;
32  end Tick_T_Recv_Sync;
33
34 end Component.Event_Component.Implementation;

```

As can be seen, the component record contains the variables `Self.Count` and `Self.First_Event_Sent` which are used to determine when to send out events. Events are constructed using the event creation functions provided by `Self.Events`, which is an instantiation of `Event_Component_Events.Instance`. Events are created and then sent out of the `Event_T_Send` connector.

6.9.4 Event Unit Testing

This section assumes you know how to unit test component which is described in Section 6.2. The details of creating a unit test model, generating tester component, and implementing unit tests are not repeated here. Instead, this section will show the important differences involved when unit testing a component that has events.

Below is a unit test implementation package that provides a single unit test, `Unit_Test`, for the `event_component`.

`event_component_tests-implementation.adb:`

```

1 -----
2 -- Event_Component Tests Body
3 -----
4
5 with Basic_Assertions; use Basic_Assertions;
6 with Packed_U16.Assertion; use Packed_U16.Assertion;
7 with Tick.Assertion; use Tick.Assertion;
8
9 package body Event_Component_Tests.Implementation is
10
11     -----
12     -- Fixtures:
13     -----
14
15     overriding procedure Set_Up_Test (Self : in out Instance) is
16     begin
17         -- Allocate heap memory to component:
18         Self.Tester.Init_Base;

```

```

19
20      -- Make necessary connections between tester and component:
21      Self.Tester.Connect;
22
23      -- Call the component set up method that the assembly would normally call.
24      Self.Tester.Component_Instance.Set_Up;
25  end Set_Up_Test;
26
27  overriding procedure Tear_Down_Test (Self : in out Instance) is
28  begin
29      -- Free component heap:
30      Self.Tester.Final_Base;
31  end Tear_Down_Test;
32
33  -----
34  -- Tests:
35  -----
36
37  overriding procedure Unit_Test (Self : in out Instance) is
38  begin
39      -- Send some ticks and check for events.
40      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
41      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
42      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
43      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
44      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
45      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
46      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
47      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
48      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
49      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
50      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
51
52      -- Make sure a total of 12 events were received.
53      Natural Assert.Eq (Self.Tester.Event_T_Recv_Sync_History.Get_Count, 13);
54
55      -- Check the individual event histories to make sure the correct
56      -- number of each event was sent:
57      Natural Assert.Eq (Self.Tester.First_Tick_Received_History.Get_Count, 1);
58      Natural Assert.Eq (Self.Tester.Tick_Received_History.Get_Count, 11);
59      Natural Assert.Eq (Self.Tester.Ten_More_Ticks_Received_History.Get_Count,
60                         1);
61
62      -- Check the event parameters and make sure they are as expected.
63      Tick Assert.Eq (
64          Self.Tester.Tick_Received_History.Get (1),
65          (Time => (0, 0), Count => 0)
66      );
67      Packed_U16 Assert.Eq (
68          Self.Tester.Ten_More_Ticks_Received_History.Get (1),
69          (Value => 10)
70      );
71  end Unit_Test;
72 end Event_Component_Tests.Implementation;

```

If a component has a event model, the autocoded tester package includes some modifications to make unit testing events simpler. Just as with connectors, a distinct history is included for every event defined in the component's event model within the tester component. This history can be queried and assertions can be run against it in the same way you would a connector history.

As can be seen in the unit test code above, some Tick.Ts are sent to the component which causes many events to be produced. The total number of events produced is checked using the event connector history. The amount and content of each individual event received is checked by asserting against the individual event histories provided in the tester component.

To run this unit test code we use:

```
> redo test
```

which produces the output:

```
OK Unit_Test
```

```
Total Tests Run: 1
Successful Tests: 1
Failed Assertions: 0
Unexpected Errors: 0
```

6.9.5 Event String Representation

Similar to *packed types*, Section 5.1.5, a string representation package can be produced for events which supplies “pretty printing” capabilities. To build this package run:

```
> redo build/obj/Linux/event_component_events-representation.o
```

This generates the source files *build/src/event_component_events-representation.ads* and *build/src/event_component_events-representation.adb*. The autocoded specification is shown below.

```

1 -----
2 -- Event_Component Events Representation Spec
3 --
4 -- Generated from event_component.events.yaml on 2025-07-15 21:51.
5 -----
6
7 -- Standard Includes:
8 with Event;
9 with Event_Types;
10 with Sys_Time;
11
12 -- A set of events for the Event Component.
13 package Event_Component_Events.Representation is
14
15 -----
16 -- Event to string functions:
17 -----
18 function Tick_Received_Image (Timestamp : in Sys_Time.T; Id : in
19   Event_Types.Event_Id; Param : in Tick.T; Instance_Name : String := "")
20   return String;
21 function Tick_Received_Image (The_Event : in Event.T; Instance_Name : in
22   String := "") return String;
23
24 function Ten_More_Ticks_Received_Image (Timestamp : in Sys_Time.T; Id : in
25   Event_Types.Event_Id; Param : in Packed_U16.T; Instance_Name : String :=
26   "") return String;
27 function Ten_More_Ticks_Received_Image (The_Event : in Event.T; Instance_Name
28   : in String := "") return String;
```

```

23
24     function First_Tick_Received_Image (Timestamp : in Sys_Time.T; Id : in
25         → Event_Types.Event_Id; Instance_Name : String := "") return String;
26     function First_Tick_Received_Image (The_Event : in Event.T; Instance_Name :
27         → in String := "") return String;
28
29 end Event_Component_Events.Representation;

```

The *_Image functions above, provided for each event defined in the event model, are useful for printing or logging events during component unit test.

6.9.6 Event Documentation

Adamant provides automatic generation of documentation for any event model. Currently two versions of documentation can be created: HTML, which is useful for presentation in meetings or for quick reference, and PDF (via L^AT_EX), which is useful for formal documentation. PDF documentation for events is created as part of component documentation and is not discussed here. See Section 6.16 for details.

To build HTML documentation run:

```
> redo build/html/event_component_events.html
```

which when opened with your favorite web browser looks something like this:

Event_Component Event Suite

Description: A set of events for the Event Component.

Event Name	Parameter Type	Description
Tick_Received	Tick.T	A tick was received by the component.
Ten_More_Ticks_Received	Packed_U16.T	This event is produced for every 10 ticks and includes the total number of ticks received by the component.
First_Tick_Received	-	This event is produced only when the first tick is received.

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/event_component/event_component.events.yaml on 2020-05-05 16:48.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

6.10 Component Data Products

Data products are component *IDed entities*. Please read Section 6.8 before continuing on to this section.

A *data product* in Adamant is a small packet that contains data from onboard sensors, software state variables, etc. These are usually produced on a periodic schedule and are meant to reflect the state of the system.

6.10.1 The Data Product Type

The data product type in Adamant is defined in *src/core/types/data_product*. A data product consists of a header followed by a byte array used to serialize the specific data, the type, associated with a data product. A data product is a *variable length packed record*, see Section 5.1.11, whose length is specified by the length field in its header. The following tables show the bit layout of a data product and its header.

Generic data_product packet for holding arbitrary data_product types

Table 13: Data_Product_Header Packed Record : 88 bits

Name	Type	Range	Size (Bits)	Start Bit	End Bit
Time	Sys_Time.T	-	64	0	63
Id	Data_Product_Types.Data_Product_Id	0 to 65535	16	64	79
Buffer_Length	Data_Product_Types.Data_Product_Buffer_Length_Type	0 to 32	8	80	87

Field Descriptions:

- **Time** - The timestamp for the data product item.
- **Id** - The data product identifier
- **Buffer_Length** - The number of bytes used in the data product buffer

Generic data product packet for holding arbitrary data types

Table 14: Data_Product Packed Record : 344 bits (*maximum*)

Name	Type	Range	Size (Bits)	Start Bit	End Bit	Variable Length
Header	Data_Product_Header.T	-	88	0	87	-
Buffer	Data_Product_Types.Data_Product_Buffer_Type	-	256	88	343	Header.Buffer_Length

Field Descriptions:

- **Header** - The data product header
- **Buffer** - A buffer that contains the data product type

As can be seen, in the header there is a 64-bit timestamp which signifies when the data product was created followed by a unique identifier for the data product. There is also a byte array where the data product's type is serialized.

6.10.2 Creating a Data Product Model

In Adamant, data products are produced by components. A component may produce many different data products. To demonstrate this we will create a component called the *data_product_component* using the following commands:

```
> mkdir data_product_component # make component directory
> cd data_product_component
> touch .all_path # add directory to build path
```

Next we will create the component by writing the following YAML file in this directory:

data_product_component.component.yaml:

```

1  ---
2  execution: passive
3  description: This is the data product component, which sends data products.
4  connectors:
5    - type: Tick.T
6      kind: recv_sync
7      description: This connector provides the schedule tick for the component.
8    - return_type: Sys_Time.T
9      kind: get
10     description: This connector is used to fetch the current system time.
11    - type: Data_Product.T
12      kind: send
13      description: This connector is used to send out data products.

```

which looks like this:



To configure the component with data products we use a data product model file. To create a data product model for this component, we simply need to name the data product model appropriately. Data product models will always be of the form `component_name.data_products.yaml` where `component_name` is the component that will be producing the data products. Only a single data product model can be configured for a component, and a single data product model cannot be shared by more than one component. In this directory we create a data product model named `data_product_component.data_products.yaml`. The contents are shown below:

`data_product_component.data_products.yaml`:

```

1  ---
2  # Optional - description of data product suite
3  description: A set of data products for the Data Product Component.
4  # Required - list of data products included in the data product suite
5  data_products:
6    # Required - the name of the data product
7    - name: Counter
8      # Optional - a description of the data product
9      description: A 16-bit counter.
10     # Required - the type of the data product
11     type: Packed_U16.T
12     - name: Last_Tick_Received
13       description: A last tick that was received by the component.
14       type: Tick.T

```

This model defines the *data product suite* for the `data_product_component`. It declares 2 different data products that can be sent by the component. Comments are provided to explain whether or not each field is optional or required. In this case, you must supply a name for each data product and the type that will be serialized into the data product's buffer. Only a single type can be provided with a data product. However, that type can be a compound type such as an array or record. Data product types should always be *packed types*, see Section 5, since their bit layout is platform independent. This allows development machines to easily decode data product values no matter what machine is running the Adamant embedded system.

The presence of a data product model causes the Adamant build system to produce many more build rules (some output omitted for brevity):

```

> redo what
redo  what
redo all
redo build/html/data_product_component_data_products.html
redo build/obj/Linux/data_product_component_data_products-representation.o
redo build/obj/Linux/data_product_component_data_products.o
redo build/src/data_product_component_data_products-representation.adb
redo build/src/data_product_component_data_products-representation.ads
redo build/src/data_product_component_data_products.adb
redo build/src/data_product_component_data_products.ads

```

We can see two object files that can be constructed, the data product suite package, *build/obj/Linux/data_product_component_data_products.o*, and the data product string representation package, *build/obj/Linux/data_product_component_data_products-representation.o*, which will be discussed in Section 6.10.5. To build the data product suite package autocode we can run:

```
> redo build/obj/Linux/data_product_component_data_products.o
```

which will construct the package *Data_Product_Component_Data_Products* in *build/src/data_product_component_data_products.ads* and *build/src/data_product_component_data_products.adb*. Let's look at the specification.

data_product_component_data_products.ads:

```

1 -----
2 -- Data_Product_Component_Data_Products Spec
3 --
4 -- Generated from data_product_component.data_products.yaml on 2025-07-15 21:48.
5 -----
6
7 -- Standard Includes:
8 with Data_Product;
9 with Data_Product_Types;
10 with Sys_Time;
11
12 -- Data Product Type Includes
13 with Packed_U16;
14 with Tick;
15
16 -- A set of data products for the Data Product Component.
17 package Data_Product_Component_Data_Products is
18
19   -- Object instance type:
20   type Instance is tagged limited private;
21
22   -----
23   -- Local Data Product Identifiers:
24   -----
25   Num_Data_Products : constant Natural := 2;
26   type Local_Data_Product_Id_Type is (
27     Counter_Id,
28     Last_Tick_Received_Id
29   );
30   for Local_Data_Product_Id_Type use (
31     Counter_Id => 0,
32     Last_Tick_Received_Id => 1
33   );
34
```

```

35 -----  

36 -- Setter procedure for data product ID base:  

37 -----  

38 not overriding function Get_Id_Base (Self : in Instance) return  

   → Data_Product_Types.Data_Product_Id  

   with Inline => True;  

39 not overriding procedure Set_Id_Base (Self : in out Instance; Id_Base : in  

   → Data_Product_Types.Data_Product_Id)  

   with Inline => True;  

40  

41 -----  

42 -- Getter function for global data product IDs:  

43 -----  

44 not overriding function Get_Counter_Id (Self : in Instance) return  

   → Data_Product_Types.Data_Product_Id  

   with Inline => True;  

45 not overriding function Get_Last_Tick_Received_Id (Self : in Instance) return  

   → Data_Product_Types.Data_Product_Id  

   with Inline => True;  

46  

47 -----  

48 -- Data product creation functions:  

49 -----  

50  

51 -- A 16-bit counter.  

52 not overriding function Counter (Self : in Instance; Timestamp : Sys_Time.T;  

   → Item : in Packed_U16.T) return Data_Product.T;  

53  

54 -- A last tick that was received by the component.  

55 not overriding function Last_Tick_Received (Self : in Instance; Timestamp :  

   → Sys_Time.T; Item : in Tick.T) return Data_Product.T;  

56  

57 -- Compile time checks to make sure types do not serialize longer than the  

   → data product buffer size:  

58 pragma Warnings (Off, "condition can only be True if invalid values  

   → present");  

59 pragma Compile_Time_Error (  

60   Packed_U16.Size_In_Bytes >  

   → Data_Product_Types.Data_Product_Buffer_Type'Length,  

61   "Data Product 'Counter' has argument of type 'Packed_U16.T' which has a  

   → maximum serialized length larger than the buffer size of  

   → Data_Product.T."  

62 );  

63 pragma Compile_Time_Error (  

64   Tick.Size_In_Bytes > Data_Product_Types.Data_Product_Buffer_Type'Length,  

65   "Data Product 'Last_Tick_Received' has argument of type 'Tick.T' which has  

   → a maximum serialized length larger than the buffer size of  

   → Data_Product.T."  

66 );  

67 pragma Warnings (On, "condition can only be True if invalid values present");
68  

69 private  

70   type Instance is tagged limited record  

71     Id_Base : Data_Product_Types.Data_Product_Id := 0;  

72   end record;  

73  

74 end Data_Product_Component_Data_Products;

```

First we can see that there is a `Instance` type which is the data product suite tagged type. There is also a public `local_Data_Product_Id_Type` enumeration which declares the *local IDs* for the data products. When instantiated in a component, these local IDs will be added to the component's *data product ID base*. The data product ID base can be set through the public function

`set_Id_Base` which is called during component initialization, see Section 6.7.3. Note that the ID base for the data product suite is stored in the `Instance` record in the private section in the `id_Base` variable. Next, there is a set of two functions which return the ID for each data product in the data product suite. The returned ID will already have the ID base offset applied to the local ID. Finally, there is a set of two functions that are used to create the two data products.

The most commonly used functions are the data product creation functions. Each function returns a type of `Data_Product.T` and takes a timestamp and the data product type as input. The implementation of these functions are not shown here but can be explored by the reader in `data_product_component_data_products.adb`. These functions take the timestamp and data product type and serialize them into the data product header and buffer, respectively. The data product is returned with the correct ID applied. To see how these functions are used within a component continue to the next section.

6.10.3 Using the Data Product Package

When a data product model is found for a component, the component base package autocode is altered slightly to include a variable called `data_products` which is an instantiation of the data product suite package instance, `Data_Product_Component_Data_Products.Instance`. The reader can verify this by building and inspecting `build/src/component-data_product_component.ads`. This `data_products` variable can then be used in the component implementation package to create `data_products`.

The following is the hand-coded implementation package body for the `data_product_component`. For directions on generating the templates for this file see Section 6.1.4.

component-data_product_component-implementation.adb:

```

1  -----
2  -- Data_Product_Component Component Implementation Body
3  -----
4
5  package body Component.Data_Product_Component.Implementation is
6
7  -----
8  -- Invokee connector primitives:
9  -----
10 -- This connector provides the schedule tick for the component.
11 overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in
12   → Tick.T) is
13   -- Get the timestamp:
14   Timestamp : constant Sys_Time.T := Self.Sys_Time_T_Get;
15 begin
16   -- Send the counter data product:
17   Self.Data_Product_T_Send (Self.Data_Products.Counter (Timestamp, (Value =>
18     → Self.Count)));
19   -- Send the last tick data product:
20   Self.Data_Product_T_Send (Self.Data_Products.Last_Tick_Received
21     → (Timestamp, Arg));
22   -- Increment the count:
23   Self.Count := @ + 1;
24 end Tick_T_Recv_Sync;
25

```

As can be seen, the component record contains the variable `Self.Count` which is sent out as a data product. Data products are constructed using the data product creation functions provided by `Self.Data_Products`, which is an instantiation of

`Data_Product_Component_Data_Products.Instance`. Data products are created and then sent out of the `Data_Product_T_Send` connector.

6.10.4 Data Product Unit Testing

This section assumes you know how to unit test component which is described in Section 6.2. The details of creating a unit test model, generating tester component, and implementing unit tests are not repeated here. Instead, this section will show the important differences involved when unit testing a component that has data products.

Below is a unit test implementation package that provides a single unit test, `Unit_Test`, for the *data_product_component*.

data_product_component_tests-implementation.adb:

```
1 -----  
2 -- Data_Product_Component Tests Body  
3 -----  
4  
5 with Basic_Assertions; use Basic_Assertions;  
6 with Packed_U16.Assertion; use Packed_U16.Assertion;  
7 with Tick.Assertion; use Tick.Assertion;  
8  
9 package body Data_Product_Component_Tests.Implementation is  
10  
11 -----  
12 -- Fixtures:  
13 -----  
14  
15 overriding procedure Set_Up_Test (Self : in out Instance) is  
16 begin  
17     -- Allocate heap memory to component:  
18     Self.Tester.Init_Base;  
19  
20     -- Make necessary connections between tester and component:  
21     Self.Tester.Connect;  
22  
23     -- Call the component set up method that the assembly would normally call.  
24     Self.Tester.Component_Instance.Set_Up;  
25 end Set_Up_Test;  
26  
27 overriding procedure Tear_Down_Test (Self : in out Instance) is  
28 begin  
29     -- Free component heap:  
30     Self.Tester.Final_Base;  
31 end Tear_Down_Test;  
32  
33 -----  
34 -- Tests:  
35 -----  
36  
37 overriding procedure Unit_Test (Self : in out Instance) is  
38 begin  
39     -- Send some ticks and check for data products.  
40     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));  
41     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));  
42  
43     -- Make sure 4 total data products were sent out:  
44     Natural Assert.Eq (Self.Tester.Data_Product_T_Recv_Sync_History.Get_Count,  
45     => 4);
```

```

46    -- Make sure 2 Counter data products were sent:
47    NaturalAssert.Eq (Self.Tester.Counter_History.Get_Count, 2);
48    Packed_U16Assert.Eq (Self.Tester.Counter_History.Get (1), (Value => 0));
49    Packed_U16Assert.Eq (Self.Tester.Counter_History.Get (2), (Value => 1));
50
51    -- Make sure 2 Last_Tick_Received data products were sent:
52    NaturalAssert.Eq (Self.Tester.Last_Tick_Received_History.Get_Count, 2);
53    TickAssert.Eq (Self.Tester.Last_Tick_Received_History.Get (1), (
54        Time => (0, 0),
55        Count => 0
56    ));
57    TickAssert.Eq (Self.Tester.Last_Tick_Received_History.Get (2), (
58        Time => (0, 0),
59        Count => 0
60    ));
61 end Unit_Test;
62
63 end Data_Product_Component_Tests.Implementation;

```

If a component has a data product model, the autocoded tester package includes some modifications to make unit testing data products simpler. Just as with connectors, a distinct history is included for every data product defined in the component's data product model within the tester component. This history can be queried and assertions can be run against it in the same way you would a connector history.

As can be seen in the unit test code above, some `Tick.Ts` are sent to the component which causes data products to be produced. The total number of data products produced is checked using the data product connector history. The amount and content of each individual data product received is checked by asserting against the individual data product histories provided in the tester component.

To run this unit test code we use:

```
> redo test
```

which produces the output:

```
OK Unit_Test
```

```
Total Tests Run: 1
Successful Tests: 1
Failed Assertions: 0
Unexpected Errors: 0
```

6.10.5 Data Product String Representation

Similar to *packed types*, Section 5.1.5, a string representation package can be produced for data products which supplies "pretty printing" capabilities. To build this package run:

```
> redo build/obj/Linux/data_product_component_data_products-representation.o
```

This generates the source files `build/src/data_product_component_data_products-representation.ads` and `build/src/data_product_component_data_products-representation.adb`. The autocoded specification is shown below.

```

1 -----
2 -- Data_Product_Component Data_Products Representation Spec
3 --
4 -- Generated from data_product_component.data_products.yaml on 2025-07-15 21:51.
5 -----
6
7 -- Standard Includes:
8 with Data_Product;
9 with Data_Product_Types;
10 with Sys_Time;
11
12 -- A set of data products for the Data Product Component.
13 package Data_Product_Component_Data_Products.Representation is
14
15 -----
16 -- Data product to string functions:
17 -----
18 function Counter_Image (Timestamp : in Sys_Time.T; Id : in
19   Data_Product_Types.Data_Product_Id; Item : in Packed_U16.T; Instance_Name
20   : in String := "") return String;
21 function Counter_Image (Dp : in Data_Product.T; Instance_Name : in String :=
22   "") return String;
23
24 function Last_Tick_Received_Image (Timestamp : in Sys_Time.T; Id : in
25   Data_Product_Types.Data_Product_Id; Item : in Tick.T; Instance_Name : in
26   String := "") return String;
27 function Last_Tick_Received_Image (Dp : in Data_Product.T; Instance_Name : in
28   String := "") return String;
29
30 end Data_Product_Component_Data_Products.Representation;

```

The *_Image functions above, provided for each data product defined in the data product model, are useful for printing or logging data products during component unit test.

6.10.6 Data Product Documentation

Adamant provides automatic generation of documentation for any data product model. Currently two versions of documentation can be created: HTML, which is useful for presentation in meetings or for quick reference, and PDF (via L^AT_EX), which is useful for formal documentation. PDF documentation for data products is created as part of component documentation and is not discussed here. See Section 6.16 for details.

To build HTML documentation run:

```
> redo build/html/data_product_component_data_products.html
```

which when opened with your favorite web browser looks something like this:

Data_Product_Component Data Product Suite

Description: A set of data products for the Data Product Component.

Data Product Name	Type	Description
Counter	Packed_U16.T	A 16-bit counter.
Last_Tick_Received	Tick.T	A last tick that was received by the component.

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/data_product_component/data_product_component.data_products.yaml on 2020-05-05 17:27.

© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

6.11 Component Data Dependencies

Data dependencies are component *IDed entities*. Please read Section 6.8 before continuing on to this section.

A *data dependency* in Adamant is simply a *data product* produced by an external component that is needed by the component that declares the dependency. *Data products* are produced by components and often stored in some central database. *Data dependencies* are consumed/used/checkered by components by fetching these *data products* from the central store. *Data dependencies* are often used by components to check the outside state of the system prior to making decisions or to perform computations based on values that are not produced inside the component.

Each *data dependency* must declare its *type* and, in the assembly, be mapped to a corresponding *data product* of the same type. In this way, the producer component and consumer component of the data are decoupled, removing any compile-time dependence within the code, and type safety is enforced by the assembly model mapping.

6.11.1 The Data Product Fetch and Return Types

Data dependencies are designed to be fetched from a central store of data products upon request from the calling component. This is implemented by a request connector with the *type* `Data_Product_Fetch.T` and *return_type* `Data_Product_Return.T` (found in `src/core/type-s/data_product`) which are shown below.

A packed record which holds information for a data product request.

Table 15: Data_Product_Fetch Packed Record : 16 bits

Name	Type	Range	Size (Bits)	Start Bit	End Bit
Id	<code>Data_Product_Types.Data_Product_Id</code>	0 to 65535	16	0	15

Field Descriptions:

- **Id** - The data product identifier

This record holds data returned from a data product fetch request.

Table 16: Data_Product_Return Packed Record : 352 bits (*maximum*)

Name	Type	Range	Size (Bits)	Start Bit	End Bit	Variable Length
The_Status	<code>Data_Product.Enums.Fetch_Status.E</code>	0 => Success 1 => Not_Available 2 => Id_Out_Of_Range	8	0	7	-
The_Data_Product	<code>Data_Product.T</code>	-	344	8	351	-

Field Descriptions:

- **The_Status** - A status relating whether or not the data product fetch was successful or not.
- **The_Data_Product** - The data product item returned.

As can be seen, the `Data_Product_Fetch.T` type simply includes the data product identifier to fetch. The `Data_Product_Return.T` includes the returned data product and a status relating whether or not the fetch was successful. A data product is a compound record type that is detailed in Section 6.10.1. As will be seen in the next section, to use data dependencies within a component, the component must instantiate a request connector using these types.

6.11.2 Creating a Data Dependency Model

In Adamant, data dependencies are fetched by components. A component may fetch many different data dependencies of many different types, so long as those data dependencies are produced as data products of the same type by some other component in the system. To demonstrate how to use data dependencies within a component we will create a component called the `data_dependency_component` using the following commands:

```
> mkdir data_dependency_component # make component directory
> cd data_dependency_component
> touch .all_path # add directory to build path
```

Next we will create the component by writing the following YAML file in this directory:

`data_dependency_component.component.yaml`:

```

1  ---
2  execution: passive
3  description: This is the data dependency component, which fetches data
4    ↳ dependencies.
5  connectors:
6    - type: Tick.T
7      kind: recv_sync
8      description: This connector provides the schedule tick for the component.
9    - return_type: Sys_Time.T
10     kind: get
11     description: This connector is used to fetch the current system time.
12    - description: Fetch a data product item from the database. This is used to
13      ↳ grab the component's data dependencies.
14      type: Data_Product_Fetch.T
15      return_type: Data_Product_Return.T
16      kind: request

```

which looks like this:



Note the inclusion of the `Data_Product_Fetch.T`/`Data_Product_Return.T` request connector which will be used to fetch the data dependencies.

To configure the component with data dependencies we use a data dependency model file. To create a data dependency model for this component, we simply need to name the data dependency model appropriately. Data dependency models will always be of the form `component_name.data_dependencies.yaml` where `component_name` is the component that will be using the data dependencies. Only a single data dependency model can be configured for a component, and a single data dependency model cannot be shared by more than one component. In this directory

we create a data dependency model named `data_dependency_component.data_dependencies.yaml`. The contents are shown below:

`data_dependency_component.data_dependencies.yaml`:

```
1  ---
2  # Optional - description of data dependency suite
3  description: A set of data dependencies for the Data Dependency Component.
4  # Required - list of data dependencies included in the data dependency suite
5  data_dependencies:
6      # Required - the name of the data dependency
7      - name: Counter
8          # Optional - a description of the data dependency
9          description: A 16-bit counter. The value of this counter is reported every
10             → tick.
11          # Required - the type of the data dependency
12          type: Packed_U16.T
13          - name: Temperature
14              description: A temperature produced by an external component that is limit
15                 → checked.
16              type: Packed_F32.T
```

This model defines the *data dependency suite* for the *data_dependency_component*. It declares 2 different data dependencies that can be fetched by the component. Comments are provided to explain whether or not each field is optional or required. In this case, you must supply a name for each data dependency and its type. Only a single type can be provided with a data dependency. However, that type can be a compound type such as an array or record. Data dependency types should always be *packed types*, see Section 5, since they must correspond to data products, which are required to be *packed types*.

The presence of a data dependency model causes the Adamant build system to produce many more build rules (some output omitted for brevity):

```
> redo what
redo what
redo all
redo build/html/data_dependency_component_data_dependencies.html
redo build/obj/Linux/data_dependency_component_data_dependencies.o
redo build/src/data_dependency_component_data_dependencies.adb
redo build/src/data_dependency_component_data_dependencies.ads
```

We can see one object file that can be constructed, the data dependency suite package, `build/obj/Linux/data_dependency_component_data_dependencies.o`. To build the data dependency suite package autocode we can run:

```
> redo build/obj/Linux/data_dependency_component_data_dependencies.o
```

which will construct the package `Data_Dependency_Component_Data_Dependencies` in `build/src/data_dependency_component_data_dependencies.ads` and `build/src/-data_dependency_component_data_dependencies.adb`. Let's look at the specification.

`data_dependency_component_data_dependencies.ads`:

```
1  -----
2  -- Data_Dependency_Component_Data_Dependencies Spec
3  --
4  -- Generated from data_dependency_component.data_dependencies.yaml on 2025-07-15
5  → 21:48.
```

```

5 -----
6
7 -- Standard Includes:
8 with Data_Product;
9 with Data_Product_Types;
10 with Sys_Time;
11 with Ada.Real_Time;
12
13 -- Data Product Type Includes
14 with Packed_U16;
15 with Packed_F32;
16
17 -- A set of data dependencies for the Data Dependency Component.
18 package Data_Dependency_Component_Data_Dependencies is
19
20     -- Object instance type:
21     type Instance is tagged limited private;
22
23     -- Status type for extraction functions:
24     type Status is (Success, Id_Error, Length_Error, Stale);
25
26     -- Constants:
27     Num_Data_Dependencies : constant Natural := 2;
28
29 -----
30     -- Setter procedure for data product IDs:
31
32     not overriding procedure Set_Ids_And_Limits (
33         Self : in out Instance;
34         Counter_Id : in Data_Product_Types.Data_Product_Id;
35         Counter_Stale_Limit : in Ada.Real_Time.Time_Span;
36         Temperature_Id : in Data_Product_Types.Data_Product_Id;
37         Temperature_Stale_Limit : in Ada.Real_Time.Time_Span
38     );
39
40 -----
41     -- Getter function for data product IDs:
42
43     not overriding function Get_Counter_Id (Self : in Instance) return
44         Data_Product_Types.Data_Product_Id
45         with Inline => True;
46     not overriding function Get_Temperature_Id (Self : in Instance) return
47         Data_Product_Types.Data_Product_Id
48         with Inline => True;
49
50 -----
51     -- Data dependency extraction functions:
52
53     -- A 16-bit counter. The value of this counter is reported every tick.
54     not overriding function Extract_Counter (Self : in Instance; Product : in
55         Data_Product.T; Stale_Reference : in Sys_Time.T; Timestamp : out
56         Sys_Time.T; Item : out Packed_U16.T) return Status;
57
58     -- A temperature produced by an external component that is limit checked.
59     not overriding function Extract_Temperature (Self : in Instance; Product : in
60         Data_Product.T; Stale_Reference : in Sys_Time.T; Timestamp : out
61         Sys_Time.T; Item : out Packed_F32.T) return Status;
62
63     -- Compile time checks to make sure types do not serialize longer than the
64     -- data product buffer size:
65     pragma Warnings (Off, "condition can only be True if invalid values
66         present");

```

```

59  pragma Compile_Time_Error (
60      Packed_U16.Size_In_Bytes >
61          => Data_Product_Types.Data_Product_Buffer_Type'Length,
62      "Data Product 'Counter' has argument of type 'Packed_U16.T' which has a
63          => maximum serialized length larger than the buffer size of
64          => Data_Product.T."
65  );
66  pragma Compile_Time_Error (
67      Packed_F32.Size_In_Bytes >
68          => Data_Product_Types.Data_Product_Buffer_Type'Length,
69      "Data Product 'Temperature' has argument of type 'Packed_F32.T' which has
70          => a maximum serialized length larger than the buffer size of
71          => Data_Product.T."
72  );
73  pragma Warnings (On, "condition can only be True if invalid values present");
74
75 private
76     type Instance is tagged limited record
77         -- Default data dependency IDs just increment from zero for unit testing.
78         -- These will get overridden by the Set_Ids_And_Limits procedure in a
79             -- running assembly.
80         -- The stale limits will be set to 1 second, which is just a general
81             -- assumption
82         -- we can make for unit testing. This will also get overridden by the
83         -- Set_Ids_And_Limits procedure in a running assembly.
84     Counter_Id : Data_Product_Types.Data_Product_Id := 0;
85     Counter_Stale_Limit : Ada.Real_Time.Time_Span := Ada.Real_Time.Seconds
86         -- (1);
87     Temperature_Id : Data_Product_Types.Data_Product_Id := 1;
88     Temperature_Stale_Limit : Ada.Real_Time.Time_Span := Ada.Real_Time.Seconds
89         -- (1);
90     end record;
91
92 end Data_Dependency_Component_Data_Dependencies;

```

Unlike for data products, packets, and events, this package is not intended to be used directly by the component implementation that it is associated with. This package is instead used within the component's base package autocode, which wraps these functions with a more friendly interface for the developer, detailed in the next section.

However, it is important to know what is included in this package. First, we can see that there is a `Instance` type, which is the data dependency suite tagged type. Stored in this `Instance` record is the ID to fetch for each data dependency. By default these are simply enumerated started at zero to make unit testing simpler. In a deployed assembly, however, these values are overridden to the appropriate IDs for their corresponding data products via the `Set_Ids_And_Limits` procedure. This gets called as part of the assembly ided entity initialization and is discussed further in Section 6.7.4.

Also provided as an input parameter to the `Set_Ids_And_Limits` initialization procedure are *stale limits* for each data dependency. The stale limit is an Ada `Time_Span` type which is used to determine if a fetched data dependency's timestamp is too old to be considered valid. For example, if the stale limit is set to 1 second, and the timestamp of the data dependency is more than 1 second older than the current time, the data product is considered to be stale, or too old to use. The numerical values used for the stale limits are set at the assembly modeling level, and eventually get passed to the `Set_Ids_And_Limits` procedure during initialization. These stale limits are stored in the `Instance` record. Note that a stale limit value of zero indicates that the data dependency can never go stale. A periodically produced data product like a temperature may reasonably go stale if it stops being produced, however some data products such as a system mode may only be produced when they are changed. Thus some data products may be considered valid, even if they are days,

months, or years older than the current time. Due to this mission specific nature, stale limits must necessarily be set at the assembly modeling level. See 7.1.2 for more details on how these limits are set at the assembly level.

For each data dependency there are two functions provided. The first retrieves the data dependency's ID, ie. `Get_Counter_Id`, and the other parses out the data dependency type from a generic data product, ie. `Extract_Counter`. Again, these functions rarely need to be called directly by component implementation code, but it is good to know they exist. We will discuss how they get used in the following section.

6.11.3 Fetching Data Dependencies

When a data dependency model is found for a component, the component base package autocode is altered to include helper functions for fetching data dependencies. When a data dependency request is issued by the component the base package does the following:

1. The data dependency ID is retrieved using the data dependency autocoded package discussed in the previous section.
2. The data dependency ID is passed out the `Data_Product_Fetch.T` connector.
3. The data dependency is returned synchronously within a `Data_Product_Return.T` type that also includes the status from the request.
4. If the status is not set to `Success`, then an `Error` or `Not_Available` is returned to the implementation package, whichever is appropriate.
5. If the status is set to `Success`, then the data dependency is extracted from the data product using the extraction functions discussed in the previous section.
6. Extracting a data dependency may fail and, if so, an `Error` status is returned to the caller.
7. The data dependency timestamp may be too old and, if so, a `Stale` status is returned to the caller.
8. If extraction succeeds, then the extracted data dependency is returned to the caller with its timestamp and a status as to whether or not the operation failed or not.

The reader can verify this logic by building and inspecting `build/src/component-data_dependency_component.ads` and `build/src/component-data_dependency_component.adb` packages. To see the data dependency interface that can be used in the implementation package, let's look at the base package specification.

component-data_dependency_component.ads:

```
1 -----  
2 -- Data_Dependency_Component Component Base Spec  
3 --  
4 -- Generated from data_dependency_component.component.yaml on 2025-07-15 21:48.  
5 -----  
6  
7 -- Includes:  
8 with Connector_Types;  
9 use Connector_Types;  
10 with Tick;  
11 with Common_Connectors;  
12 with Sys_Time;  
13 with Data_Product_Return;  
14 with Data_Product_Fetch;  
15 with Data_Dependency_Component_Data_Dependencies;  
16 with Data_Product_Types;  
17 with Ada.Real_Time;
```

```

18  with Data_Product.Enums;
19  with Packed_U16;
20  with Packed_F32;
21
22  -- This is the data dependency component, which fetches data dependencies.
23  package Component.Data_Dependency_Component is
24
25      -- Base class instance:
26      type Base_Instance is abstract new Component.Core_Instance with private;
27      type Base_Class_Access is access all Base_Instance'Class;
28
29  -----
30  -- Initialize the IDs for the component's data dependencies
31  -----
32  -- This procedure maps the component's data dependencies to external data
33  -- product
34  -- IDs and initializes the stale limits for each.
35
36  -- Set Data Dependency Ids Parameters:
37  -- Counter_Id : Data_Product_Types.Data_Product_Id - The external data
38  -- product ID
39  -- used to fetch the value for the 'Counter' data dependency.
40  -- Counter_Stale_Limit : Ada.Real_Time.Time_Span - The stale limit used for
41  -- the
42  -- 'Counter' data dependency. If the fetched data product is older than the
43  -- given
44  -- Time_Span then it is deemed stale and unfit for use by the component.
45  -- Temperature_Id : Data_Product_Types.Data_Product_Id - The external data
46  -- product
47  -- ID used to fetch the value for the 'Temperature' data dependency.
48  -- Temperature_Stale_Limit : Ada.Real_Time.Time_Span - The stale limit used
49  -- for
50  -- the 'Temperature' data dependency. If the fetched data product is older
51  -- than
52  -- the given Time_Span then it is deemed stale and unfit for use by the
53  -- component.
54
55  not overriding procedure Map_Data_Dependencies (Self : in out Base_Instance;
56  -- Counter_Id : in Data_Product_Types.Data_Product_Id; Counter_Stale_Limit :
57  -- in Ada.Real_Time.Time_Span; Temperature_Id : in
58  -- Data_Product_Types.Data_Product_Id; Temperature_Stale_Limit : in
59  -- Ada.Real_Time.Time_Span);
60
61  -----
62  -- Invokee connector primitives:
63  -----
64  -- This connector provides the schedule tick for the component.
65  -- Define connector package:
66  package Tick_T_Recv_Sync_Connector renames
67  -- Common_Connectors.Tick_T_In_Connector;
68  -- Define index type for invokee connector.
69  subtype Tick_T_Recv_Sync_Index is Connector_Index_Type range
70  -- Connector_Index_Type'First .. Connector_Index_Type'First;
71  -- Abstract connector procedure to be overridden by child:
72  not overriding procedure Tick_T_Recv_Sync (Self : in out Base_Instance; Arg :
73  -- in Tick.T) is abstract;
74  -- Function which returns the hook for this connector. Used when attaching
75  -- this connector to an invoker connector:
76  not overriding function Tick_T_Recv_Sync_Access (Self : in Base_Instance;
77  -- Index : in Connector_Index_Type := Connector_Index_Type'First) return not
78  -- null Tick_T_Recv_Sync_Connector.Invokee_Hook

```

```

61      with Inline => True;
62
63  -----
64  -- Invoker connector primitives:
65  -----
66  -- This connector is used to fetch the current system time.
67  -- Define connector package:
68  package Sys_Time_T_Get_Connector renames
69    <-- Common_Connectors.Sys_Time_T_Return_Connector;
70  -- Function to attach this invoker connector to an invokee connector:
71  not overriding procedure Attach_Sys_Time_T_Get (Self : in out Base_Instance;
72    <-- To_Component : in not null Component.Class_Access; Hook : in not null
73    <-- Sys_Time_T_Get_Connector.Invokee_Hook; To_Index : in Connector_Index_Type
74    <-- := Connector_Index_Type'First)
75      with Inline => True;
76
77  -- Fetch a data product item from the database. This is used to grab the
78  -- component's data dependencies.
79  -- Define connector package:
80  package Data_Product_Fetch_T_Request_Connector renames
81    <-- Common_Connectors.Data_Product_Fetch_T_In_Return_Connector;
82  -- Function to attach this invoker connector to an invokee connector:
83  not overriding procedure Attach_Data_Product_Fetch_T_Request (Self : in out
84    <-- Base_Instance; To_Component : in not null Component.Class_Access; Hook :
85    <-- in not null Data_Product_Fetch_T_Request_Connector.Invokee_Hook; To_Index
86    <-- : in Connector_Index_Type := Connector_Index_Type'First)
87      with Inline => True;
88
89  -----
90  -- Abstract data dependency procedures to be overridden:
91  -----
92  -- Description:
93  -- A set of data dependencies for the Data Dependency Component.
94  -- Function which retrieves a data dependency. This should be overridden by
95  -- the implementation to call the correct connector.
96  not overriding function Get_Data_Dependency (Self : in out Base_Instance; Id
97    <-- : in Data_Product_Types.Data_Product_Id) return Data_Product_Return.T is
98    <-- abstract;
99
100  -- Invalid data dependency handler. This procedure is called when a data
101  -- dependency's id or length are found to be invalid:
102  not overriding procedure Invalid_Data_Dependency (Self : in out
103    <-- Base_Instance; Id : in Data_Product_Types.Data_Product_Id; Ret : in
104    <-- Data_Product_Return.T) is abstract;
105
106  -----
107  -- Private invokee connector hooks which
108  -- dispatch invokee calls to the correct
109  -- abstract function defined in the
110  -- child package:
111  -----
112  -- This connector provides the schedule tick for the component.
113  function Tick_T_Recv_Sync_Hook (Class_Self : in out
114    <-- Component.Core_Instance'Class; Arg : in Tick.T; Index : in
115    <-- Connector_Index_Type := Connector_Index_Type'First; Full_Queue_Behavior :
116    <-- in Full_Queue_Action := Drop) return Connector_Types.Connector_Status;

```

```

108
-----
110 -- Private invoker connector functions
111 -- for use in the child package:
112 -----
113 -- This connector is used to fetch the current system time.
114 not overriding function Sys_Time_T_Get (Self : in Base_Instance) return
115   Sys_Time.T
116   with Inline => True;
117 not overriding function Is_Sys_Time_T_Get_Connected (Self : in Base_Instance)
118   return Boolean
119   with Inline => True;
120 -- Fetch a data product item from the database. This is used to grab the
121 -- component's data dependencies.
122 not overriding function Data_Product_Fetch_T_Request (Self : in
123   Base_Instance; Arg : in Data_Product_Fetch.T) return
124   Data_Product_Return.T
125   with Inline => True;
126 not overriding function Is_Data_Product_Fetch_T_Request_Connected (Self : in
127   Base_Instance) return Boolean
128   with Inline => True;

129 -----
130 -- Private subprograms for data dependencies
131 -----
132 -- Private helper functions to get data dependencies:
133 not overriding function Get_Counter (Self : in out Base_Instance;
134   Stale_Reference : in Sys_Time.T; Timestamp : out Sys_Time.T; Value : out
135   Packed_U16.T) return Data_Product.Enums.Data_Dependency_Status.E;
136 not overriding function Get_Counter (Self : in out Base_Instance;
137   Stale_Reference : in Sys_Time.T; Value : out Packed_U16.T) return
138   Data_Product.Enums.Data_Dependency_Status.E
139   with Inline => True;
140 not overriding function Get_Temperature (Self : in out Base_Instance;
141   Stale_Reference : in Sys_Time.T; Timestamp : out Sys_Time.T; Value : out
142   Packed_F32.T) return Data_Product.Enums.Data_Dependency_Status.E;
143 not overriding function Get_Temperature (Self : in out Base_Instance;
144   Stale_Reference : in Sys_Time.T; Value : out Packed_F32.T) return
145   Data_Product.Enums.Data_Dependency_Status.E
146   with Inline => True;

147 -----
148 -- The base class instance record:
149 type Base_Instance is abstract new Component.Core_Instance with record
150   -- Invoker connector objects:
151   Connector_Sys_Time_T_Get : Sys_Time_T_Get_Connector.Instance;
152   Connector_Data_Product_Fetch_T_Request :
153     Data_Product_Fetch_T_Request_Connector.Instance;
154   -- Data dependencies instance:
155   Data_Dependencies : Data_Dependency_Component_Data_Dependencies.Instance;
156 end record;
157
158 end Component.Data_Dependency_Component;

```

The bottom section of the specification is what we are most interested in. Note that for each data dependency there are two helper functions defined. The each do the same thing, except one does not pass back the data dependency timestamp to the caller. As an example, for the Counter data dependency we can see there is a Get_Counter function. This is meant to be called directly from the implementation package to fetch data dependencies. The Get_Counter function returns the counter value, its timestamp, and a status relating the success or failure of the operation. The

`stale_Reference` parameter is used to check if the data dependency is stale with respect to some time reference. Usually the current time is passed as the `stale_Reference`. A stale return value indicates that the data product is too old to be considered valid to use within the component. The measure of “too old” is determined by the value of `stale_Reference`, which is the time reference (usually the current time), and *stale limit*, which is defined for each data dependency at initialization (in the assembly model, as we will see in Section 7.1.2). The *stale limit* is a timespan that determines how much older than `stale_Reference` is still considered an unstale data dependency. In other words, any data dependency fetched with a timestamp that is older than `stale_Reference` minus `stale limit` is considered stale and will produce a return status of `Stale` to the caller.

Next, let us discuss how we can use these functions within the component implementation. The following is the hand-coded implementation package specification and body for the `data_dependency_component`. For directions on generating the templates for these file see Section 6.1.4. Looking at the specification first, we can see a few new procedures of note.

component-data_dependency_component-implementation.ads:

```

1  -----
2  -- Data_Dependency_Component Component Implementation Spec
3  -----
4
5  -- Includes:
6  with Tick;
7
8  -- This is the data dependency component, which fetches data dependencies.
9  package Component.Data_Dependency_Component.Implementation is
10
11    -- The component class instance record:
12    type Instance is new Data_Dependency_Component.Base_Instance with private;
13
14  private
15
16    -- The component class instance record:
17    type Instance is new Data_Dependency_Component.Base_Instance with record
18      null; -- TODO
19    end record;
20
21  -----
22  -- Set Up Procedure
23  -----
24  overriding procedure Set_Up (Self : in out Instance) is null;
25
26  -----
27  -- Invokee connector primitives:
28  -----
29  -- This connector provides the schedule tick for the component.
30  overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in
31  --> Tick.T);
32
33  -----
34  -- Data dependency primitives:
35  -----
36  -- Description:
37  --   A set of data dependencies for the Data Dependency Component.
38  -- Function which retrieves a data dependency.
39  -- The default implementation is to simply call the
40  --> Data_Product_Fetch_T_Request connector. Change the implementation if this
41  -- component
42  -- needs to do something different.
43  overriding function Get_Data_Dependency (Self : in out Instance; Id : in
44  --> Data_Product_Types.Data_Product_Id) return Data_Product_Return.T is
45  --> (Self.Data_Product_Fetch_T_Request ((Id => Id)));

```

```

41
42    -- Invalid data dependency handler. This procedure is called when a data
43    -- dependency's id or length are found to be invalid:
44    overriding procedure Invalid_Data_Dependency (Self : in out Instance; Id : in
45        Data_Product_Types.Data_Product_Id; Ret : in Data_Product_Return.T);

```

The data dependency primitives `Get_Data_Dependency` and `Invalid_Data_Dependency` have been included. The first is the user defined implementation for how to fetch a data dependency given a data product ID. In almost all cases this is usually just implemented via the call `Self.Data_Product_Fetch_T_Request((id => id))`, which is why it is implemented that way by default. As a general rule, the Adamant component base package does not make connector calls, and reserves that right for the implementation package only, which is why it is defined in the implementation specification, and not in the base package. If your component fetches data dependencies in a different manner, than this function can be replaced by your custom implementation.

The `Invalid_Data_Dependency` procedure is called whenever a data dependency is received that is considered invalid. In a well formed assembly, invalid data dependencies should never be encountered, as checks are performed at the modeling level to ensure that this cannot happen. Nevertheless, if one of these issues should be detected by the autocoded base package, this procedure will be called. Usually the implementation of this procedure sends out an informational event alerting of the failure. Below are the possible reasons that this procedure might be called:

- The data dependency ID requested is out of range as determined by the data product central store.
- The data product ID returned does not match the data dependency ID requested.
- The data product returned has an unexpected length that does not match the data dependency type's length.

Since data product and data dependency IDs are assigned by the assembly model and checked for type consistency, the following errors can only result if their is a misconfigured assembly, or a bug in either the modeling autocode or the component.

Now let's look at the implementation package body to see how to fetch data dependencies.

component-data_dependency_component-implementation.adb:

```

1 -----
2 -- Data_Dependency_Component Component Implementation Body
3 -----
4
5 with Ada.Text_IO; use Ada.Text_IO;
6 with Interfaces; use Interfaces;
7 with Sys_Time;
8
9 package body Component.Data_Dependency_Component.Implementation is
10
11   -----
12   -- Invokee connector primitives:
13   -----
14   -- This connector provides the schedule tick for the component.
15   overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in
16     Tick.T) is
17     use Data_Product.Enums;
18     use Data_Product.Enums.Data_Dependency_Status;
19     -- Get the current time:

```

```

19      Current_Time : constant Sys_Time.T := Self.Sys_Time_T_Get;
20      -- Define the temperature limit we are checking against:
21      Temperature_Limit : constant Short_Float := 100.0;
22      -- Data dependency value.
23      Temp : Packed_F32.T;
24      -- Data dependency timestamp:
25      Ignore_Timestamp : Sys_Time.T;
26      -- Data dependency value.
27      Cnt : Packed_U16.T;
28      Status : Data_Dependency_Status.E;
29 begin
30     -- First let's grab the counter data dependency and report its value.
31     Status := Self.Get_Counter (
32         Stale_Reference => Current_Time,
33         Timestamp => Ignore_Timestamp,
34         Value => Cnt
35     );
36
37     -- Check the status:
38     case Status is
39         when Success =>
40             Put_Line ("The counter value is: " & Unsigned_16'Image (Cnt.Value));
41         when Not_Available =>
42             Put_Line ("The counter value is unavailable!");
43         when Stale =>
44             Put_Line ("The counter value is stale!");
45         when Error =>
46             null; -- Invalid_Data_Dependency will be called, no need to do
47             -- anything more.
48     end case;
49
50     -- Grab the temperature and see if it is over the limit:
51     Status := Self.Get_Temperature (
52         Stale_Reference => Current_Time,
53         Timestamp => Ignore_Timestamp,
54         Value => Temp
55     );
56
57     -- Check the status:
58     case Status is
59         when Success =>
60             if Temp.Value > Temperature_Limit then
61                 Put_Line ("The temperature value is too hot!");
62             else
63                 Put_Line ("The temperature value is just right.");
64             end if;
65         when Not_Available =>
66             Put_Line ("The temperature value is unavailable!");
67         when Stale =>
68             Put_Line ("The temperature value is stale!");
69         when Error =>
70             null; -- Invalid_Data_Dependency will be called, no need to do
71             -- anything more.
72     end case;
73 end Tick_T_Recv_Sync;
74
75 -----
76 -- Data dependency handlers:
77 -----
78 -- Description:
79 --   A set of data dependencies for the Data Dependency Component.

```

```

78  -- Invalid data dependency handler. This procedure is called when a data
79  -- dependency's id or length are found to be invalid:
80  overriding procedure Invalid_Data_Dependency (Self : in out Instance; Id : in
81  -- Data_Product_Types.Data_Product_Id; Ret : in Data_Product_Return.T) is
82  begin
83      Put_Line ("Oh, no! A data dependency was received that was malformed!");
84  end Invalid_Data_Dependency;
85
86  end Component.Data_Dependency_Component.Implementation;

```

Besides the `Invalid_Data_Dependency` procedure, which just prints out an error message if an invalid data dependency is received, the only subprogram included is the `Tick.T` connector handler. This would likely be tied to a rate group and get called periodically in a running assembly. The implementation of the handler does two things: 1) It grabs the Counter data dependency and reports its value via a print message, and 2) It grabs the Temperature data dependency and checks it against a static limit if it is both valid. Both data dependencies are also checked to make sure they are not stale.

The Counter data dependency and its corresponding timestamp is requested via the `Get_Counter` function. Note that the return status of this function should be checked prior to using the returned counter value, as it may not be valid. If `Success` is returned, then the counter is valid and can be used. If `Not_Available` is returned, then there is not corresponding data product available yet in the central store to relate the value of the counter. In this case, the returned value for `cnt` should not be used. If `Stale` is returned then the data dependency's timestamp was older than the `stale_Reference` (the current time in this case) minus the data dependency's *stale limit*. Using a data dependency that is stale can be problematic, and should be avoided. Finally, if `Error` is returned, then something unexpected happened, and you should also not use `cnt`. Note that an `Error` return value will only be returned after a call to `Invalid_Data_Dependency` has been made by the base package. In this way, the component developer can encapsulate all error handling for invalid data dependencies in this single subprogram.

The Temperature data dependency is requested via the `Get_Temperature` function and is checked in a similar fashion. The next section will describe how to unit test component that uses data dependencies.

6.11.4 Data Dependency Unit Testing

This section assumes you know how to unit test component which is described in Section 6.2. The details of creating a unit test model, generating tester component, and implementing unit tests are not repeated here. Instead, this section will show the important differences involved when unit testing a component that has data dependencies.

Below is a unit test implementation package that provides a single unit test, `Unit_Test`, for the `data_dependency_component`.

`data_dependency_component_tests-implementation.adb:`

```

1  -----
2  -- Data_Dependency_Component Tests Body
3  -----
4
5  with Ada.Text_IO; use Ada.Text_IO;
6  with Data_Product.Enums;
7
8  package body Data_Dependency_Component_Tests.Implementation is
9
10   -----
11   -- Fixtures:

```

```

12 -----
13
14 overriding procedure Set_Up_Test (Self : in out Instance) is
15 begin
16     -- Allocate heap memory to component:
17     Self.Tester.Init_Base;
18
19     -- Make necessary connections between tester and component:
20     Self.Tester.Connect;
21
22     -- Call the component set up method that the assembly would normally call.
23     Self.Tester.Component_Instance.Set_Up;
24
25     -- System time for test. Make this non-zero to data dependencies don't
26     -- report as stale.
27     Self.Tester.System_Time := (10_000, 0);
28 end Set_Up_Test;
29
30 overriding procedure Tear_Down_Test (Self : in out Instance) is
31 begin
32     -- Free component heap:
33     Self.Tester.Final_Base;
34 end Tear_Down_Test;
35
36 -----
37 -- Tests:
38 -----
39
40 overriding procedure Unit_Test (Self : in out Instance) is
41     use Data_Product.Enums.Fetch_Status;
42 begin
43     -- Set data dependency values in tester component.
44     Self.Tester.Counter := (Value => 10);
45     Self.Tester.Temperature := (Value => 15.0);
46
47     -- Send tick and expect nominal response.
48     Put_Line ("Nominal test...");
49     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
50     Put_Line ("");
51
52     -- Send tick and expect over temp response.
53     Put_Line ("Over temperature test...");
54     Self.Tester.Temperature := (Value => 101.0);
55     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
56     Put_Line ("");
57
58     -- Set the data dependency time to something too old:
59     Put_Line ("Stale time test...");
60     -- Set timestamp of data dependency.
61     Self.Tester.Data_Dependency_Timestamp_Override := (90, 0);
62     -- Set simulated system time.
63     Self.Tester.System_Time := (101, 0);
64     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
65     Put_Line ("");
66
67     -- Set the data dependency ID to something nonsensical:
68     Put_Line ("Bad ID test...");
69     Self.Tester.Data_Dependency_Return_Id_Override := 99;
70     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
71     Put_Line ("");

```

```

72      -- Set the return status to Not_Available:
73      Self.Tester.Data_Dependency_Return_Status_Override := Not_Available;
74      Put_Line ("Not available test...");
75      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
76      Put_Line ("");
77  end Unit_Test;
78
79 end Data_Dependency_Component_Tests.Implementation;

```

If a component has a data dependency model, the autocoded tester package includes some modifications to make unit testing data products simpler. First, a variable for each data dependency is included within the tester package, ie. `Self.Tester.Counter` and `Self.Tester.Temperature`. The value of these variables is returned directly to the component when the data dependencies are requested. Note that by default, during unit testing the *stale limits* of all data dependencies is initialized to 1 second. So if a data dependency is received that is older than one second when compared to the unit test time, `Self.Tester.System_Time` then it will look stale to the component.

In the unit test, the first thing we do is set these variables to reasonable value. Next, we send a `Tick.T` to the component to tell it to grab these data dependencies. Next, we set the temperature to a value above the static limit and sent the `Tick.T` again.

The subsequent tests are used to cover the different paths a component can take when checking the data dependency statuses. In the third section we set the special `Self.Tester.Data_Dependency_Timestamp_Override` variable to a non-zero timestamp. This tells the tester to send this timestamp with any requested data dependencies instead of the simulated system time, `Self.Tester.System_Time`. In this case we also set `Self.Tester.System_Time` such that the returned Temperature data dependency will be determined to be stale by the component under test.

The fourth section sets another special tester variable `Self.Tester.Data_Dependency_Return_Id_Override`. When non-zero, the tester component will return the value of this variable with every requested data dependency. In this way you can simulate a misconfigured system that should cause the component to receive an `Error` status.

In the final section we set the special tester variable `Self.Tester.Data_Dependency_Return_Status_Override` which allows us to simulate a different response than `Success` when a component requests a data dependency. In this case we set it to `Not_Available` to see how the component responds.

Note that there is one additional special tester variable available (but not demonstrated) called `data_Dependency_Return_Length_Override`. When non-zero, the tester component will return the value of this variable as the data dependency length. You can use this to simulate a misconfigured system in addition to the `Self.Tester.Data_Dependency_Return_Id_Override` variable.

To run this unit test code we use:

```
> redo test
```

which produces the output:

```
Nominal test...
The counter value is: 10
The temperature value is just right.
```

```
Over temperature test...
The counter value is: 10
```

```
The temperature value is too hot!
```

```
Stale time test...
```

```
The counter value is stale!
```

```
The temperature value is stale!
```

```
Bad ID test...
```

```
Oh, no! A data dependency was received that was malformed!
```

```
Oh, no! A data dependency was received that was malformed!
```

```
Not available test...
```

```
The counter value is unavailable!
```

```
The temperature value is unavailable!
```

```
OK Unit_Test
```

```
Total Tests Run: 1
```

```
Successful Tests: 1
```

```
Failed Assertions: 0
```

```
Unexpected Errors: 0
```

We can see that we have exercised full coverage of all the paths that this component can execute in its implementation package.

6.11.5 Data Dependency Documentation

Adamant provides automatic generation of documentation for any data dependency model. Currently two versions of documentation can be created: HTML, which is useful for presentation in meetings or for quick reference, and PDF (via L^AT_EX), which is useful for formal documentation. PDF documentation for data dependencies is created as part of component documentation and is not discussed here. See Section 6.16 for details.

To build HTML documentation run:

```
> redo build/html/data_dependency_component_data_dependencies.html
```

which when opened with your favorite web browser looks something like this:

Data_Dependency_Component Data Dependency Suite

Description: A set of data dependencies for the Data Dependency Component.

Data Dependency Name	Type	Description
Counter	Packed_U16.T	A 16-bit counter. The value of this counter is reported every tick.
Temperature	Packed_F32.T	A temperature produced by an external component that is limit checked.

*This file was autogenerated from
/vagrant/adamant/doc/example_architecture/data_dependency_component/data_dependency_component.data_dependencies.yaml on 2021-05-27 22:13.

© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

6.12 Component Packets

Packets are component *IDed entities*. Please read Section 6.8 before continuing on to this section.

Packets are similar to *data products*, but usually larger. Often, a packet contains many *data products* within. *Packets* are also often used for external communication.

6.12.1 The Packet Type

The packet type in Adamant is defined in *src/core/types/packet*. A packet consists of a header followed by a byte array used to serialize the specific data, the type, associated with a packet. A packet is a *variable length packed record*, see Section 5.1.11, whose length is specified by the length field in its header. The following tables show the bit layout of a packet and its header.

Generic packet header for holding arbitrary data

Table 17: Packet _ Header Packed Record : 112 bits

Name	Type	Range	Size (Bits)	Start Bit	End Bit
Time	Sys_Time.T	-	64	0	63
Id	Packet_Types. Packet_Id	0 to 65535	16	64	79
Sequence_Count	Packet_Types. Sequence_Count_Mod_Type	0 to 16383	16	80	95
Buffer_Length	Packet_Types. Packet_Buffer_Length_Type	0 to 1246	16	96	111

Field Descriptions:

- **Time** - The timestamp for the packet item.
- **Id** - The packet identifier
- **Sequence_Count** - Packet Sequence Count
- **Buffer_Length** - The number of bytes used in the packet buffer

Generic packet for holding arbitrary data

Table 18: Packet Packed Record : 10080 bits (*maximum*)

Name	Type	Range	Size (Bits)	Start Bit	End Bit	Variable Length
Header	Packet_Header.T	-	112	0	111	-
Buffer	Packet_Types.Packet_Buffer_Type	-	9968	112	10079	Header.Buffer_Length

Field Descriptions:

- **Header** - The packet header
- **Buffer** - A buffer that contains the packet data

As can be seen, in the header there is a 64-bit timestamp which signifies when the packet was created followed by a unique identifier for the packet. A packet also contains a Sequence_Count which

increments with every packet that is created. This can be used by a packet receiver to determine if any packets have been dropped or to correctly order received data. There is also a byte array where the packet's type is serialized.

6.12.2 Creating a Packet Model

In Adamant, packets are produced by components. A component may produce many different packets. To demonstrate this we will create a component called the *packet_component* using the following commands:

```
> mkdir packet_component # make component directory
> cd packet_component
> touch .all_path         # add directory to build path
```

Next we will create the component by writing the following YAML file in this directory:

packet_component.component.yaml:

```
1  ---
2  execution: passive
3  description: This is the packet component, which sends packets.
4  connectors:
5    - type: Tick.T
6      kind: recv_sync
7      description: This connector provides the schedule tick for the component.
8    - return_type: Sys_Time.T
9      kind: get
10     description: This connector is used to fetch the current system time.
11    - type: Packet.T
12      kind: send
13      description: This connector is used to send out packets.
```

which looks like this:



To configure the component with packets we use a packet model file. To create a packet model for this component, we simply need to name the packet model appropriately. Packet models will always be of the form *component_name.packets.yaml* where *component_name* is the component that will be producing the packets. Only a single packet model can be configured for a component, and a single packet model cannot be shared by more than one component. In this directory we create a packet model named *packet_component.packets.yaml*. The contents are shown below:

packet_component.packets.yaml:

```
1  ---
2  # Optional - description of packet suite
3  description: A set of packets for the Packet Component.
4  # Required - list of packets included in the packet suite
5  packets:
6    # Required - the name of the packet
7    - name: Last_Tick_Received
8      # Optional - a description of the packet
9      description: A last tick that was received by the component.
```

```

10  # Optional - the type of the packet
11  type: Tick.T
12  # Optional - a statically defined ID for the packet
13  # If this is specified, then IDs must be specified for
14  # EVERY packet in this packet suite. The Set_Id_Base
15  # procedure will then NOT be used to set the packet IDs.
16  #
17  # If this is not specified, then NO packets in this packet
18  # suite may specify an ID. In this case the Set_Id_Base
19  # procedure will be used to set the packed IDs.
20  id: 17
21  - name: Counter
22  description: A packet containing an incrementing 16-bit counter.
23  id: 5

```

This model defines the *packet suite* for the *packet component*. It declares 2 different packets that can be sent by the component. Comments are provided to explain whether or not each field is optional or required. In this case, you must supply a name for each packet and the type that will be serialized into the packet's buffer. Only a single type can be provided with a packet. However, that type can be a compound type such as an array or record. Packet types should always be *packed types*, see Section 5, since their bit layout is platform independent. This allows development machines to easily decode packet values no matter what machine is running the Adamant embedded system. The type for the packet is optional. If not specified, the user must fill in the packet data using a generic byte array, which will be shown in the following section.

Unique to packets (and faults, as we will see later), a developer may also specify the ID for each packet. If specified, the ID refers to the global ID of the packet as defined in the assembly. In this case, there is no packet *id base*. This feature should only be used if you are constructing a mission-specific component, where you want to define static IDs for all the packets it creates. For reusable components, it is advised to not specify the ID for any packets, and to use the normal *Set_Id_Base* method for setting packet IDs, see Section 6.7.3. By not statically defining the packet IDs, the component remains reusable for future systems.

Note that if you define a static global ID for one packet, you MUST define the ID for every packet in the suite, as we have done here. Similarly, if you wish to not define the static IDs for the packets, you must NOT define the packet IDs for every packet in the packet suite.

The presence of a packet model causes the Adamant build system to produce many more build rules (some output omitted for brevity):

```

> redo what
redo  what
redo  all
redo build/html/packet_component_packets.html
redo build/obj/Linux/packet_component_packets-representation.o
redo build/obj/Linux/packet_component_packets.o
redo build/src/packet_component_packets-representation.adb
redo build/src/packet_component_packets-representation.ads
redo build/src/packet_component_packets.adb
redo build/src/packet_component_packets.ads

```

We can see two object files that can be constructed, the packet suite package, *build/obj/Linux/packet_component_packets.o*, and the packet string representation package, *build/obj/Linux/packet_component_packets-representation.o*, which will be discussed in Section 6.12.5. To build the packet suite package autocode we can run:

```
> redo build/obj/Linux/packet_component_packets.o
```

which will construct the package `Packet_Component_Packets` in `build/src/`-`packet_component_packets.ads` and `build/src/``packet_component_packets.adb`. Let's look at the specification.

`packet_component_packets.ads`:

```
1 -----  
2 -- Packet_Component Packets Spec  
3 --  
4 -- Generated from packet_component.packets.yaml on 2025-07-15 21:48.  
5 -----  
6  
7 -- Standard Includes:  
8 with Packet_Types;  
9 with Packet;  
10 with Sys_Time;  
11  
12 -- Packet Type Includes  
13 with Tick;  
14 with Basic_Types;  
15 with Serializer_Types; use Serializer_Types;  
16  
17 -- A set of packets for the Packet Component.  
18 package Packet_Component_Packets is  
19  
20     -- Object instance type:  
21     type Instance is tagged limited private;  
22  
23 -----  
24     -- Local Packet Identifiers:  
25 -----  
26     Num_Packets : constant Natural := 2;  
27     type Local_Packet_Id_Type is (  
28         Counter_Id,  
29         Last_Tick_Received_Id  
30     );  
31     for Local_Packet_Id_Type use (  
32         Counter_Id => 5,  
33         Last_Tick_Received_Id => 17  
34     );  
35  
36 -----  
37     -- Getter function for global packet IDs:  
38 -----  
39     not overriding function Get_Counter_Id (Self : in Instance) return  
40         Packet_Types.Packet_Id  
41         with Inline => True;  
42     not overriding function Get_Last_Tick_Received_Id (Self : in Instance) return  
43         Packet_Types.Packet_Id  
44         with Inline => True;  
45  
46 -----  
47     -- Packet creation functions:  
48 -----  
49     -- A packet containing an incrementing 16-bit counter.  
50     not overriding function Counter (Self : in out Instance; Timestamp : in  
51         Sys_Time.T; Buf : in Basic_Types.Byte_Array; Pkt : out Packet.T) return  
52         Serialization_Status;  
53     -- Special function, which will fill the Packet.T type as much as possible,  
54     -- and then drop any remaining bytes found in the  
55     -- input item. Sometimes this is desirable over returning a  
56     -- Serialization_Error, like what would occur with the above function.
```

```

51  not overriding function Counter_Truncate (Self : in out Instance; Timestamp :
52    => in Sys_Time.T; Buf : in Basic_Types.Byte_Array) return Packet.T;
53  -- This function returns a packet with the timestamp, id, and sequence number
54  -- filled in, but with a data length of zero and no
55  -- zeroed buffered data. Sometimes this is more convenient to use than the
56  -- functions above.
57  not overriding function Counter_Empty (Self : in out Instance; Timestamp : in
58    => Sys_Time.T) return Packet.T;
59
60  -- A last tick that was received by the component.
61  not overriding function Last_Tick_Received (Self : in out Instance; Timestamp
62    => : in Sys_Time.T; Item : in Tick.T) return Packet.T;
63
64  -- Compile time checks to make sure types do not serialize longer than the
65  -- packet buffer size:
66  pragma Warnings (Off, "condition can only be True if invalid values
67  -- present");
68  pragma Warnings (Off, "condition can only be True if invalid values
69  -- present");
70  pragma Compile_Time_Error (
71    Tick.Size_In_Bytes > Packet_Types.Packet_Buffer_Type'Length,
72    "Packet 'Last_Tick_Received' has argument of type 'Tick.T' which has a
73    -- maximum serialized length larger than the buffer size of Packet.T."
74  );
75  pragma Warnings (On, "condition can only be True if invalid values present");
76 private
77
78  type Instance is tagged limited record
79    Counter_Sequence_Count : Packet_Types.Sequence_Count_Mod_Type := 0;
80    Last_Tick_Received_Sequence_Count : Packet_Types.Sequence_Count_Mod_Type
81      => := 0;
82  end record;
83
84 end Packet_Component_Packets;

```

First, we can see that there is a `Instance` type which is the packet suite tagged type. There is also a public `local_Packet_Id_Type` enumeration which declares the *local IDs* for the packets. In this case, the IDs are actually the global IDs that were defined in the packet model. Notice that the `Instance` type does not have an `id_Base` variable and there is no `set_Id_Base` procedure defined. The local IDs will be directly used within the packets and cannot be modified at initialization. If the IDs were omitted from the packet model, then the autocoder would generate an `id_Base` variable and a `set_Id_Base` procedure and the packet IDs would be set normally at initialization, see Section 6.7.3.

Also take note of the two variables defined within the `Instance` record. These are used to keep track of the sequence number for each packet. These numbers are incremented and stored in the packet header every time a new packet is created, so the user does not have to worry about filling in that field.

Next is a set of two functions which return the ID for each packet in the packet suite.

Following, are the packet creation functions. The most commonly used functions are the packet creation functions. Each function returns a type of `Packet.T` and takes a timestamp and the packet type as input. The implementation of these functions are not shown here but can be explored by the reader in `packet_component_packets.adb`. These functions take the timestamp and packet type and serialize them into the packet header and buffer, respectively. The packet is returned with the correct ID and sequence count applied.

The `Last_Tick_Received` function looks similar to the data product creation functions that we saw in the previous sections. The `Counter` function comes in three varieties, due to the fact that

a packet type was not declared in the packet model for the Counter packet. The first function, Counter, constructs the packet using a supplied byte array. If the byte array is too large to fit in the packet then a Failure status is returned through the `Serialization_Status` return type. Otherwise, the byte array is copied into the packet, the length is set appropriately, and the constructed packet is returned through the `out` parameter. The `Counter_Truncate` function is the same as the `Counter` function, except that if the supplied byte array is too large, then any bytes larger than the packet buffer are simply discarded and a maximum sized packed is returned. Finally, the `Counter_Empty` function returns a packet with the header set with the correct ID, sequence count, and timestamp. The packet's data is not filled in, and the user is expected to populate it manually before sending the packet out. This function should be used if filling the packet with data is more complicated than the other functions allow.

To see how these functions are used within a component continue to the next section.

6.12.3 Using the Packet Package

When a packet model is found for a component, the component base package autocode is altered slightly to include a variable called `packets` which is an instantiation of the packet suite package instance, `Packet_Component_Packets.Instance`. The reader can verify this by building and inspecting `build/src/component-packet_component.ads`. This `packets` variable can then be used in the component implementation package to create packets.

The following is the hand-coded implementation package body for the `packet_component`. For directions on generating the templates for this file see Section 6.1.4.

component-packet_component-implementation.adb:

```

1  -----
2  -- Packet_Component Component Implementation Body
3  -----
4
5  with Packed_U16;
6
7  package body Component.Packet_Component.Implementation is
8
9      -----
10     -- Invokee connector primitives:
11     -----
12     -- This connector provides the schedule tick for the component.
13     overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in
14         => Tick.T) is
15         -- Get the timestamp:
16         Timestamp : constant Sys_Time.T := Self.Sys_Time_T_Get;
17     begin
18         -- Send the counter packet:
19         Self.Packet_T_Send (Self.Packets.Counter_Truncate (
20             Timestamp,
21             Packed_U16.Serialization.To_Byte_Array ((Value => Self.Count)))
22         );
23
24         -- Send the last tick data product:
25         Self.Packet_T_Send (Self.Packets.Last_Tick_Received (Timestamp, Arg));
26
27         -- Increment the count:
28         Self.Count := @ + 1;
29     end Tick_T_Recv_Sync;
30 end Component.Packet_Component.Implementation;

```

As can be seen, the component record contains the variable `Self.Count` which is sent out as a

packet. Packets are constructed using the packet creation functions provided by `Self.Packets`, which is an instantiation of `Packet_Component_Packets.Instance`. Packets are created and then sent out of the `packet_T_Send` connector. Note that since no type was provided for the Counter packet in the model, we have to supply our own byte array to populate the packet. In this case, the byte array is generated by serializing the `Self.Count` value using the `Packed_U16` packed record serialization function.

6.12.4 Packet Unit Testing

This section assumes you know how to unit test component which is described in Section 6.2. The details of creating a unit test model, generating tester component, and implementing unit tests are not repeated here. Instead, this section will show the important differences involved when unit testing a component that has packets.

Below is a unit test implementation package that provides a single unit test, `Unit_Test`, for the *packet_component*.

packet_component_tests-implementation.adb:

```

1  -----
2  -- Packet_Component Tests Body
3  -----
4
5  with Basic_Assertions; use Basic_Assertions;
6  with Tick.Assertion; use Tick.Assertion;
7
8  package body Packet_Component_Tests.Implementation is
9
10
11  -- Fixtures:
12  -----
13
14  overriding procedure Set_Up_Test (Self : in out Instance) is
15  begin
16      -- Allocate heap memory to component:
17      Self.Tester.Init_Base;
18
19      -- Make necessary connections between tester and component:
20      Self.Tester.Connect;
21
22      -- Call the component set up method that the assembly would normally call.
23      Self.Tester.Component_Instance.Set_Up;
24  end Set_Up_Test;
25
26  overriding procedure Tear_Down_Test (Self : in out Instance) is
27  begin
28      -- Free component heap:
29      Self.Tester.Final_Base;
30  end Tear_Down_Test;
31
32
33  -- Tests:
34  -----
35
36  overriding procedure Unit_Test (Self : in out Instance) is
37  begin
38      -- Send some ticks and check for data products.
39      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
40      Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
41
42      -- Make sure 4 total packets were sent out:

```

```

43     Natural.Assert.Eq (Self.Tester.Packet_T_Recv_Sync_History.Get_Count, 4);
44
45     -- Make sure 2 Counter packets were sent:
46     Natural.Assert.Eq (Self.Tester.Counter_History.Get_Count, 2);
47     Byte_Array Assert.Eq (Self.Tester.Counter_History.Get (1).Buffer (0 .. 1),
48     => [0, 0]);
49     Byte_Array Assert.Eq (Self.Tester.Counter_History.Get (2).Buffer (0 .. 1),
50     => [0, 1]);
51
52     -- Make sure 2 Last_Tick_Received packets were sent:
53     Natural.Assert.Eq (Self.Tester.Last_Tick_Received_History.Get_Count, 2);
54     Tick Assert.Eq (
55         Self.Tester.Last_Tick_Received_History.Get (1),
56         (Time => (0, 0), Count => 0)
57     );
58     Tick Assert.Eq (Self.Tester.Last_Tick_Received_History.Get (2),
59         (Time => (0, 0), Count => 0)
60     );
61 end Unit_Test;
62
63 end Packet_Component_Tests.Implementation;

```

If a component has a packet model, the autocoded tester package includes some modifications to make unit testing packets simpler. Just as with connectors, a distinct history is included for every packet defined in the component's packet model within the tester component. This history can be queried and assertions can be run against it in the same way you would a connector history.

As can be seen in the unit test code above, some `Tick.Ts` are sent to the component which causes some packets to be produced. The total number of packets produced is checked using the packet connector history. The amount and content of each individual packet received is checked by asserting against the individual packet histories provided in the tester component. Note that because the Counter packet does not have a defined packet type, the history simply stores the entire packet itself. In this case, we just check the first two bytes of the packet buffer to ensure that the data inside matches the values we expect.

To run this unit test code we use:

```
> redo test
```

which produces the output:

```
OK Unit_Test
```

```
Total Tests Run:    1
Successful Tests:  1
Failed Assertions: 0
Unexpected Errors: 0
```

6.12.5 Packet String Representation

Similar to *packed types*, Section 5.1.5, a string representation package can be produced for packets which supplies "pretty printing" capabilities. To build this package run:

```
> redo build/obj/Linux/packet_component_packets-representation.o
```

This generates the source files `build/src/packet_component_packets-representation.ads` and

`build/src/packet_component_packets-representation.adb`. The autocoded specification is shown below.

```
1 -----  
2 -- Packet_Component Packets Representation Spec  
3 --  
4 -- Generated from packet_component.packets.yaml on 2025-07-15 21:51.  
5 -----  
6  
7 -- Standard Includes:  
8 with Packet;  
9 with Packet_Types;  
10 with Sys_Time;  
11  
12 -- A set of packets for the Packet Component.  
13 package Packet_Component_Packets.Representation is  
14  
15 -----  
16 -- Packet to string functions:  
17 -----  
18 function Counter_Image (Timestamp : in Sys_Time.T; Id : in  
→   Packet_Types.Packet_Id; Buf : in Basic_Types.Byte_Array; Instance_Name :  
→   in String := "") return String;  
19 function Counter_Image (P : in Packet.T; Instance_Name : in String := "")  
→   return String;  
20  
21 function Last_Tick_Received_Image (Timestamp : in Sys_Time.T; Id : in  
→   Packet_Types.Packet_Id; Item : in Tick.T; Instance_Name : in String :=  
→   "") return String;  
22 function Last_Tick_Received_Image (P : in Packet.T; Instance_Name : in String  
→   := "") return String;  
23  
24 end Packet_Component_Packets.Representation;
```

The `*_Image` functions above, provided for each packet defined in the packet model, are useful for printing or logging packets during component unit test.

6.12.6 Packet Documentation

Adamant provides automatic generation of documentation for any packet model. Currently two versions of documentation can be created: HTML, which is useful for presentation in meetings or for quick reference, and PDF (via L^AT_EX), which is useful for formal documentation. PDF documentation for packets is created as part of component documentation and is not discussed here. See Section 6.16 for details.

To build HTML documentation run:

```
> redo build/html/packet_component_packets.html
```

which when opened with your favorite web browser looks something like this:

Packet_Component Packet Suite

Description: A set of packets for the Packet Component.

Packet Name	Global Packet Id	Type	Description
Counter	0x0005 (5)	-	A packet containing an incrementing 16-bit counter.
Last_Tick_Received	0x0011 (17)	Tick.T	A last tick that was received by the component.

*This file was autogenerated from [/vagrant/adamant/doc/example_architecture/packet_component/packet_component.packets.yaml](#) on 2020-05-06 16:11.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

6.13 Component Commands

Commands are component *IDed entities*. Please read Section 6.8 before continuing on to this section.

Commands are packets created by an external system that are received by the embedded software and “executed”. *Commands* are used to direct the software to perform certain actions at certain times.

6.13.1 The Command Type

The command type in Adamant is defined in *src/core/types/command*. A command consists of a header followed by a byte array used to serialize the specific data, the command arguments, associated with a command. A command is a *variable length packed record*, see Section 5.1.11, whose length is specified by the length field in its header. The following tables show the bit layout of a command and its header.

Generic command header for holding arbitrary commands

Table 19: Command_Header Packed Record : 40 bits

Name	Type	Range	Size (Bits)	Start Bit	End Bit
Source_Id	Command_Types.Command_Source_Id	0 to 65535	16	0	15
Id	Command_Types.Command_Id	0 to 65535	16	16	31
Arg_Buffer_Length	Command_Types.Command_Arg_Buffer_Length_Type	0 to 255	8	32	39

Field Descriptions:

- **Source_Id** - The source ID. An ID assigned to a command sending component.
- **Id** - The command identifier
- **Arg_Buffer_Length** - The number of bytes used in the command argument buffer

Generic command packet for holding arbitrary commands

Table 20: Command Packed Record : 2080 bits (*maximum*)

Name	Type	Range	Size (Bits)	Start Bit	End Bit	Variable Length

Header	Command_Header.T	-	40	0	39	-
Arg_Buffer	Command_Types.Command_Arg_Buffer_Type	-	2040	40	2079	Header.Arg_Buffer_Length

Field Descriptions:

- **Header** - The command header
- **Arg_Buffer** - A buffer to that contains the command arguments

As can be seen, in the header there is a unique identifier for the command, commonly referred to as the “opcode”. There is also a `Source_Id` field which indicates which component in the system (or externally) the command originated from. Finally, there is a byte array where the command’s argument is serialized.

6.13.2 Creating a Command Model

In Adamant, commands are executed by components, and in some rare cases, also produced by components. This section will discuss the first case. A component may execute many different commands. To demonstrate this we will create a component called the `command_component` using the following steps:

```
> mkdir command_component # make component directory
> cd command_component
> touch .all_path           # add directory to build path
```

Next we will create the component by writing the following YAML file in this directory:

`command_component.component.yaml`:

```

1  ---
2  execution: passive
3  description: This is the command component, which executes commands.
4  connectors:
5    - description: This is the command receive connector.
6      type: Command.T
7      kind: recv_sync
8    - description: This connector is used to register the components commands at
9      ↪ initialization, and send command responses after execution.
10     type: Command_Response.T
11     kind: send

```

which looks like this:



Note that the component has two connectors. In the Adamant command system, every component has one connector to receive commands and a second connector on which to send back a *command response*, which indicates whether or not the command succeeded or failed during execution.

To configure the component with commands we use a command model file. To create a command model for this component, we simply need to name the command model appropriately. Command models will always be of the form `component_name.commands.yaml` where `component_name` is the component that will be executing the commands. Only a single command model can be configured for a component, and a single command model cannot be shared by more than one component. In this directory we create a command model named `command_component.commands.yaml`. The contents are shown below:

`command_component.commands.yaml`:

```

1  ---
2  # Optional - description of command suite
3  description: A set of commands for the Command Component.
4  # Required - list of commands included in the command suite
5  commands:
6      # Required - the name of the command
7      - name: Hello_World
8          # Optional - a description of the command
9          description: Prints Hello World when received.
10         - name: Was_Five_Sent
11             description: A command that when received checks to see if the commanded
12                 ↳ argument is equal to 5. If so the command succeeds, otherwise it fails.
13             # Optional - the argument type included with the command
              arg_type: Packed_U16.T

```

This model defines the *command suite* for the *command_component*. It declares 2 different commands that can be sent to the component for execution. Comments are provided to explain whether or not each field is optional or required. In this case, you must supply a name for each command. Optionally you can provide an argument type that will be serialized into the command's buffer. Only a single argument type can be provided with a command. However, that type can be a compound type such as an array or record. Command types should always be *packed types*, see Section 5, since their bit layout is platform independent. This allows development machines to easily construct properly formatted commands no matter what machine is running the Adamant embedded system.

The presence of a command model causes the Adamant build system to produce many more build rules (some output omitted for brevity):

```

> redo what
redo what
redo all
redo build/html/command_component_commands.html
redo build/obj/Linux/command_component_commands.o
redo build/src/command_component_commands.adb
redo build/src/command_component_commands.ads

```

We can see one object file that can be constructed, the command suite package, `build/obj/Linux/command_component_commands.o`. To build the command suite package autocode we can run:

```
> redo build/obj/Linux/command_component_commands.o
```

which will construct the package `Command_Component_Commands` in `build/src/command_component_commands.ads` and `build/src/command_component_commands.adb`. Let's look at the specification.

`command_component_commands.ads`:

```

1 -----  

2 -- Command_Component Commands Spec  

3 --  

4 -- Generated from command_component.commands.yaml on 2025-07-15 21:47.  

5 -----  

6  

7 -- Standard Includes  

8 with Command;  

9 with Command_Types;  

10  

11 -- Argument Includes  

12 with Packed_U16;  

13  

14 -- A set of commands for the Command Component.  

15 package Command_Component_Commands is  

16  

17 -- Object instance type:  

18 type Instance is tagged limited private;  

19  

20 -----  

21 -- Local Command Identifiers:  

22 -----  

23 Num_Commands : constant Natural := 2;  

24 type Local_Command_Id_Type is (  

25     Hello_World_Id,  

26     Was_Five_Sent_Id  

27 );  

28 for Local_Command_Id_Type use (  

29     Hello_World_Id => 0,  

30     Was_Five_Sent_Id => 1  

31 );  

32  

33 -----  

34 -- Setter procedure for command source ID:  

35 -----  

36 not overriding function Get_Source_Id (Self : in out Instance) return  

37     Command_Types.Command_Source_Id  

38     with Inline => True;  

39 not overriding procedure Set_Source_Id (Self : in out Instance; Source_Id :  

40     in Command_Types.Command_Source_Id)  

41     with Inline => True;  

42  

43 -----  

44 -- Setter procedure for command ID base:  

45 -----  

46 not overriding function Get_Id_Base (Self : in Instance) return  

47     Command_Types.Command_Id  

48     with Inline => True;  

49 not overriding procedure Set_Id_Base (Self : in out Instance; Id_Base : in  

50     Command_Types.Command_Id)  

51     with Inline => True;  

52  

53 -----  

54 -- Getter function for global command IDs:  

55 -----  

56 not overriding function Get_Hello_World_Id (Self : in Instance) return  

57     Command_Types.Command_Id  

58     with Inline => True;  

59 not overriding function Get_Was_Five_Sent_Id (Self : in Instance) return  

60     Command_Types.Command_Id  

61     with Inline => True;

```

```

56
57 -----  

58 -- Command creation functions:  

59 -----  

60  

61 -- Prints Hello World when received.  

62 not overriding function Hello_World (Self : in Instance) return Command.T;  

63 -- A command that when received checks to see if the commanded argument is  

64 -- equal  

65 -- to 5. If so the command succeeds, otherwise it fails.  

66 not overriding function Was_Five_Sent (Self : in Instance; Arg : in  

67 -- Packed_U16.T) return Command.T;  

68  

69 -- Compile time checks to make sure types do not serialize longer than the  

70 -- command buffer size:  

71 pragma Warnings (Off, "condition can only be True if invalid values  

72 -- present");  

73 pragma Compile_Time_Error (
74     Packed_U16.Size_In_Bytes > Command_Types.Command_Arg_Buffer_Type'Length,  

75     "Command 'Was_Five_Sent' has argument of type 'Packed_U16.T' which has a  

76     maximum serialized length larger than the buffer size of Command.T."  

77 );  

78 pragma Warnings (On, "condition can only be True if invalid values present");  

79  

80 private
81     type Instance is tagged limited record
82         Id_Base : Command_Types.Command_Id := 1;
83         Source_Id : Command_Types.Command_Source_Id := 0;
84     end record;
85 end Command_Component_Commands;

```

Unlike for data products, packets, and events, this package is not intended to be used by the component that it is associated with. This package is used to *create* commands, not *execute* them, and thus is most commonly used to generate commands while unit testing the component. The commands are executed within the component itself using special autocode within the component base package, see the following section. In rare cases, an external component may use this package to create and send commands to the destination component. This pattern is commonly used for a fault response component in spacecraft flight software. Let's explore the package's contents, remembering that it will most likely be utilized by a tester component during unit test.

First, we can see that there is a `Instance` type, which is the command suite tagged type. There is also a public `local_command_Id_Type` enumeration which declares the *local IDs* for the commands. When instantiated in a component, these local IDs will be added to the component's *command ID base*. The command ID base can be set through the public function `set_Id_Base` which is called during component initialization, see Section 6.7.3. Note that the ID base for the command suite is stored in the `Instance` record in the `id_Base` variable.

Also stored in the `Instance` record is a `source_Id` variable. This variable can be set or retrieved through the `set_Source_Id` and `get_Source_Id` subprograms. The `Source_Id` gets populated in the command header and signifies where the command is originating from. This value is generally not used by the component executing command, but is used by the Adamant standard command routing and command response forwarding system provided by the Command Router component, see `src/components/command_router`.

Next, there is a set of two functions which return the ID for each command in the command suite. The returned ID will already have the ID base offset applied to the local ID. Finally, there is a set of two functions that are used to create the two commands.

The most commonly used functions are the command creation functions. Each function returns a type of `Command.T` and takes the command argument (if there is one) as input. The im-

plementation of these functions are not shown here but can be explored by the reader in *command_component_commands.adb*. These functions take the command argument type and serialize it into the command buffer. The command is returned with the correct ID and source ID applied. To see how these functions are used within a unit test see Section 6.13.5.

6.13.3 Implementing Command Handlers

When a command model is found for a component, the component base package autocode is altered to include command handling logic. When a command is received by a component the component base package does the following:

1. The command ID is checked for validity. If it is out of range an `Id_Error` return status is set.
2. The command length is checked for validity. If it does not match the expected length a `Length_Error` return status is set.
3. The command argument is checked for validity by making sure it is within the Ada range(s) defined for the argument type. If it is not valid, then a `Validation_Error` return status is set.
4. If all the previous checks pass, the command is passed to the implementation package command handler for execution, otherwise proceed to the last step.
5. The command handler executes and either sets the command return status to `Success` or `Failure`.
6. A command response is constructed using the return status and sent out of the component.

The reader can verify this logic by building and inspecting *build/src/component-command_component.ads* and *build/src/component-command_component.adb*.

Also defined in the base package is an abstract subprogram for each command that must be executed. These subprograms ensure that the command handlers are implemented within the implementation package, otherwise a compilation error will result.

The developer must implement the abstract command handler functions for each command defined in the base package. The following is the hand-coded implementation package body for the *command_component* with the command handlers implemented. For directions on generating the templates for this file see Section 6.1.4.

component-command_component-implementation.adb:

```

1-----  

2-- Command_Component Component Implementation Body  

3-----  

4  

5 with Ada.Text_IO; use Ada.Text_IO;  

6  

7 package body Component.Command_Component.Implementation is  

8-----  

9-----  

10-- Invokee connector primitives:  

11-----  

12-- This is the command receive connector.  

13 overriding procedure Command_T_Recv_Sync (Self : in out Instance; Arg : in  

14    → Command.T) is  

15      -- Execute the command:  

16      Stat : constant Command_Response_Status.E := Self.Execute_Command (Arg);  

17 begin  

18      -- Send the return status:  

19      Self.Command_Response_T_Send_If_Connected ((Source_Id =>  

20          → Arg.Header.Source_Id, Registration_Id => Self.Command_Reg_Id,  

21          → Command_Id => Arg.Header.Id, Status => Stat));
```

```

19    end Command_T_Recv_Sync;
20
21  -----
22  -- Command handler primitives:
23  -----
24  -- Description:
25  --   A set of commands for the Command Component.
26  -- Prints Hello World when received.
27  overriding function Hello_World (Self : in out Instance) return
28    → Command_Execution_Status.E is
29      use Command_Execution_Status;
30      begin
31        Put_Line ("Hello, World!");
32        return Success;
33      end Hello_World;
34
35  -- A command that when received checks to see if the commanded argument is
36  -- equal to 5. If so the command succeeds, otherwise it fails.
37  overriding function Was_Five_Sent (Self : in out Instance; Arg : in
38    → Packed_U16.T) return Command_Execution_Status.E is
39    use Command_Execution_Status;
40    begin
41      Put_Line (Unsigned_16'Image (Arg.Value) & " was sent.");
42      if Arg.Value = 5 then
43        return Success;
44      else
45        return Failure;
46      end if;
47    end Was_Five_Sent;
48
49  -- Invalid command handler. This procedure is called when a command's
50  -- arguments are found to be invalid:
51  overriding procedure Invalid_Command (Self : in out Instance; Cmd : in
52    → Command.T; Errant_Field_Number : in Unsigned_32; Errant_Field : in
      → Basic_Types.Poly_Type) is
53    begin
54      Put_Line ("Oh, no! A command was sent that was malformed.");
55    end Invalid_Command;
56
57  end Component.Command_Component.Implementation;

```

The first thing to notice is the implementation of the Command_T_Recv_Sync connector handler. This code is completely generated by the template autocoder, and rarely needs to be modified by the developer. It first calls execute_Command which calls an autocoded function in the base package that performs the procedure described above, validating and executing the command. Returned from this call is the command status, which is copied into a command response record and sent out of the component through a call to Command_Response_T_Send connector. Note that the call to execute_Command is going to, in turn, call the command handlers that are implemented just below the Command_T_Recv_Sync procedure.

The two command handler functions are named after the commands that they service. The Hello_World command simply prints out the famous text and then returns a Success status. The Was_Five_Sent handler is a bit more complicated. It checks the command argument to see if it equals 5. If so, then a Success status is returned, otherwise a Failure status is returned.

Note, all command handler functions return a type of Command_Execution_Status.E, which is an enumeration with only two states, Success and Failure. One of the two must be returned by the developer. The base package autocode will copy the return value into the command response message that gets forwarded out after command execution. The command response type itself is described in the following section.

Finally, there is a `Invalid_Command` procedure defined. This procedure must be implemented anytime a component includes a command model. The procedure gets called by the autocoded base package if the command is not valid, ie. it does not have a valid ID, length, or command arguments. Usually this function sends out an event alerting the operator of the system that their command was invalid. In this case we just print some text to standard out.

6.13.4 The Command Response Type

The command response type in Adamant is defined in `src/core/types/command`. Adamant requires a command response be sent out after any command is executed. The type itself is shown below.

Record for holding command response data.

Table 21: Command_Response Packed Record : 56 bits

Name	Type	Range	Size (Bits)	Start Bit	End Bit
Source_Id	Command_Types.Command_Source_Id	0 to 65535	16	0	15
Registration_Id	Command_Types.Command_Registration_Id	0 to 65535	16	16	31
Command_Id	Command_Types.Command_Id	0 to 65535	16	32	47
Status	Command.Enums.Command_Response_Status.E	0 => Success 1 => Failure 2 => Id_Error 3 => Validation_Error 4 => Length_Error 5 => Dropped 6 => Register 7 => Register_Source	8	48	55

Field Descriptions:

- **Source_Id** - The source ID. An ID assigned to a command sending component.
- **Registration_Id** - The registration ID. An ID assigned to each registered component at initialization.
- **Command_Id** - The command ID for the command response.
- **Status** - The command execution status.

The record contains `Source_Id` and `Command_Id` which are copied from the command itself. The `Registration_Id` is used for command routing within Adamant and is not discussed here. See the Command Router component for more details, `src/components/command_router`. The most important part of the response is the `Status` field which relates if the command was successfully executed or not. The possible status enumeration literals and their numerical representations are shown. As a developer, you are only responsible for returning a command's status as `Success` or `Failure` via a return value in the command's handler function. All other status values are returned, when necessary, by the autocode within the component base package.

6.13.5 Command Unit Testing

This section assumes you know how to unit test component which is described in Section 6.2. The details of creating a unit test model, generating tester component, and implementing unit tests are not repeated here. Instead, this section will show the important differences involved when unit testing a component that has commands.

Below is a unit test implementation package that provides a single unit test, `Unit_Test`, for the *command_component*.

command_component_tests-implementation.adb:

```
1 -----  
2 -- Command_Component Tests Body  
3 -----  
4  
5 with Command_Response.Assertion; use Command_Response.Assertion;  
6 with Command.Enums; use Command.Enums.Command_Response_Status;  
7 with Basic_Assertions; use Basic_Assertions;  
8  
9 package body Command_Component_Tests.Implementation is  
10  
11 -----  
12 -- Fixtures:  
13 -----  
14  
15 overriding procedure Set_Up_Test (Self : in out Instance) is  
16 begin  
17     -- Allocate heap memory to component:  
18     Self.Tester.Init_Base;  
19  
20     -- Make necessary connections between tester and component:  
21     Self.Tester.Connect;  
22  
23     -- Call the component set up method that the assembly would normally call.  
24     Self.Tester.Component_Instance.Set_Up;  
25 end Set_Up_Test;  
26  
27 overriding procedure Tear_Down_Test (Self : in out Instance) is  
28 begin  
29     -- Free component heap:  
30     Self.Tester.Final_Base;  
31 end Tear_Down_Test;  
32  
33 -----  
34 -- Tests:  
35 -----  
36  
37 overriding procedure Unit_Test (Self : in out Instance) is  
38 begin  
39     -- Send hello world command:  
40     Self.Tester.Command_T_Send (Self.Tester.Commands.Hello_World);  
41  
42     -- Make sure a command response was sent back and that is was Success.  
43     Natural Assert.Eq  
44         --> (Self.Tester.Command_Response_T_Recv_Sync_History.Get_Count, 1);  
45     Command_Response Assert.Eq  
46         --> (Self.Tester.Command_Response_T_Recv_Sync_History.Get (1), (  
47             Source_Id => 0,  
48             Registration_Id => 0,  
49             Command_Id => Self.Tester.Commands.Get_Hello_World_Id,
```

```

48         Status => Success
49     ) );
50
51     -- Send was five sent command:
52     Self.Tester.Command_T_Send (Self.Tester.Commands.Was_Five_Sent ((Value =>
53     ↳ 5)));
54
55     -- Make sure a command response was sent back and that is was Success.
56     Natural Assert.Eq
57     ↳ (Self.Tester.Command_Response_T_Recv_Sync_History.Get_Count, 2);
58     Command_Response Assert.Eq
59     ↳ (Self.Tester.Command_Response_T_Recv_Sync_History.Get (2), (
60         Source_Id => 0,
61         Registration_Id => 0,
62         Command_Id => Self.Tester.Commands.Get_Was_Five_Sent_Id,
63         Status => Success
64     ) );
65
66     -- Send was five sent command:
67     Self.Tester.Command_T_Send (Self.Tester.Commands.Was_Five_Sent ((Value =>
68     ↳ 6)));
69
70     -- Make sure a command response was sent back and that is Failure.
71     Natural Assert.Eq
72     ↳ (Self.Tester.Command_Response_T_Recv_Sync_History.Get_Count, 3);
73     Command_Response Assert.Eq
74     ↳ (Self.Tester.Command_Response_T_Recv_Sync_History.Get (3), (
75         Source_Id => 0,
76         Registration_Id => 0,
77         Command_Id => Self.Tester.Commands.Get_Was_Five_Sent_Id,
78         Status => Failure
79     ) );
80 end Unit_Test;
81
82 end Command_Component_Tests.Implementation;

```

If a component has a command model, the autocoded tester package includes some modifications to make unit testing commands simpler. Included in the tester base package `Instance` record is a variable called `Commands` which is of the type `Command_Component_Commands.Instance`. As can be seen in the unit test code above, the `Self.Tester.Commands` variable is used to construct commands that are then sent to the component on the `Command.T` send connector. After the command is sent, the unit test makes sure that a command response is returned by the component into the command response connector history. The command responses are checked for Success or Failure where appropriate.

To run this unit test code we use:

```
> redo test
```

which produces the output:

```
Hello, World!
5 was sent.
6 was sent.
```

```
OK Unit_Test
```

```
Total Tests Run:    1
Successful Tests:   1
```

```
Failed Assertions: 0
Unexpected Errors: 0
```

6.13.6 Command Documentation

Adamant provides automatic generation of documentation for any command model. Currently two versions of documentation can be created: HTML, which is useful for presentation in meetings or for quick reference, and PDF (via L^AT_EX), which is useful for formal documentation. PDF documentation for commands is created as part of component documentation and is not discussed here. See Section 6.16 for details.

To build HTML documentation run:

```
> redo build/html/command_component_commands.html
```

which when opened with your favorite web browser looks something like this:

Command_Component Command Suite

Description: A set of commands for the Command Component.		
Command Name	Argument Type	Description
Hello_World	-	Prints Hello World when received.
Was_Five_Sent	Packed_U16_T	A command that when received checks to see if the commanded argument is equal to 5. If so the command succeeds, otherwise it fails.
<small>*This file was autogenerated from /vagrant/adamant/doc/example_architecture/command_component/command_component.commands.yaml on 2020-05-06 16:40. © The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)</small>		

6.14 Component Parameters

Parameters are component *IDed entities*. Please read Section 6.8 before continuing on to this section.

Parameters are values that are used to configure a component. Parameters are often stored in a single location on the embedded system, and can be changed by an external system to reconfigure it. Changes to these parameters are then “pushed” to components to change their behavior.

6.14.1 The Parameter Type

The parameter type in Adamant is defined in *src/core/types/parameter*. A parameter consists of a header followed by a byte array used to serialize the specific data, the type, associated with a parameter. A parameter is a *variable length packed record*, see Section 5.1.11, whose length is specified by the length field in its header. The following tables show the bit layout of a parameter and its header.

Generic parameter header for holding arbitrary parameters

Table 22: Parameter_Header Packed Record : 24 bits

Name	Type	Range	Size (Bits)	Start Bit	End Bit
Id	Parameter_Types.Parameter_Id	0 to 65535	16	0	15
Buffer_Length	Parameter_Types.Parameter_Buffer_Length_Type	0 to 32	8	16	23

Field Descriptions:

- **Id** - The parameter identifier
- **Buffer_Length** - The number of bytes used in the parameter type buffer

Generic parameter packet for holding a generic parameter

Table 23: Parameter Packed Record : 280 bits (*maximum*)

Name	Type	Range	Size (Bits)	Start Bit	End Bit	Variable Length
Header	Parameter_Header.T	-	24	0	23	-
Buffer	Parameter_Types.Parameter_Buffer_Type	-	256	24	279	Header.Buffer_Length

Field Descriptions:

- **Header** - The parameter header
- **Buffer** - A buffer to that contains the parameter type

As can be seen, in the header there is a unique identifier for the parameter. There is also a byte array where the parameter's serialized type is stored.

6.14.2 The Parameter Update Type

The parameter type itself does not contain enough information to allow effective updating of parameters through connectors in the system, so a Parameter_Update type has been created for this purpose. The type is located in *src/core/types/parameter/parameter_update.record.yaml* and has the following layout.

A record intended to be used as a provide/modify connector type for updating/fetching parameters.

Table 24: Parameter_Update Packed Record : 296 bits (*maximum*)

Name	Type	Range	Size (Bits)	Start Bit	End Bit	Variable Length
Operation	Parameter.Enums.Parameter_Operation_Type.E	0 => Stage 1 => Update 2 => Fetch 3 => Validate	8	0	7	-
Status	Parameter.Enums.Parameter_Update_Status.E	0 => Success 1 => Id_Error 2 => Validation_Error 3 => Length_Error	8	8	15	-

Param	Parameter.T	-	280	16	295	-
-------	-------------	---	-----	----	-----	---

Field Descriptions:

- **Operation** - The parameter operation to perform.
- **Status** - The parameter return status.
- **Param** - The parameter that has been updated or fetched.

As can be seen the Parameter_Update type contains an additional Operation field which states whether or not to *Stage*, *Update*, or *Fetch* the parameter, concepts that will be discussed in the following sections. It also contains a Status field which returns the status of the operation as executed by the receiving component. The possible status enumeration literals and their numerical representations are shown. As a developer, you never need to set the parameter update status, as this is completely handled by the base package autocode. More details on updating parameters are discussed in the following sections.

6.14.3 Creating a Parameter Model

In Adamant, parameters are used by components to configure their behavior. A component may contain many different parameters. To demonstrate this we will create a component called the *parameter_component* using the following steps:

```
> mkdir parameter_component # make component directory
> cd parameter_component
> touch .all_path           # add directory to build path
```

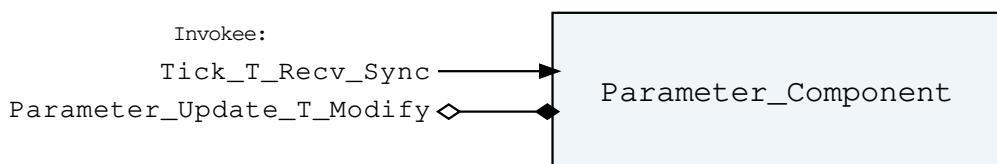
Next we will create the component by writing the following YAML file in this directory:

parameter_component.component.yaml:

```

1  ---
2  execution: passive
3  description: This is the parameter component, which is configured via
4    ↳ parameters.
5  connectors:
6    - type: Tick.T
7      kind: recv_sync
8      description: This connector provides the schedule tick for the component.
9    - type: Parameter_Update.T
10      kind: modify
11      description: The parameter update connector.
```

which looks like this:



Note that the component has two connectors. The first is the *Tick.T* connector which drives the execution of the component. The second is the *Parameter_Update.T* modify connector which allows the component's parameters to be updated during execution. The *Parameter_Update.T* modify connector will return a status that relates whether or not the parameter was successfully

updated or not.

To configure the component with parameters we use a parameter model file. To create a parameter model for this component, we simply need to name the parameter model appropriately. Parameter models will always be of the form *component_name.parameters.yaml* where *component_name* is the component that will be executing the parameters. Only a single parameter model can be configured for a component, and a single parameter model cannot be shared by more than one component. In this directory we create a parameter model named *parameter_component.parameters.yaml*. The contents are shown below:

parameter_component.parameters.yaml:

```
1  ---
2  # Optional - description of parameter suite
3  description: A set of parameters for the Parameter Component.
4  # Required - list of parameters included in the parameter suite
5  parameters:
6      # Required - the name of the parameter
7      - name: Start_Count
8          # Optional - a description of the parameter
9          description: The value to start counting at.
10         # Required - the parameter type
11         type: Packed_U16.T
12         # Required - the default value of the parameter. This is the value that
13         # is used until the component's parameters have been set by an external
14         # parameter managing component.
15         default: "(Value => 0)"
16     - name: Hello_World_Value
17         description: The value of count at which the Hello, World! string is
18             ↳ printed.
19         type: Packed_U16.T
20         default: "(Value => 16)"
```

This model defines the *parameter suite* for the *parameter_component*. It declares 2 different parameters that are used to configure the component. Comments are provided to explain whether or not each field is optional or required. In this case, you must supply a name for each parameter. You must also provide the type for each parameter. Only a single type can be used to described a parameter. However, that type can be a compound type such as an array or record. Parameter types should always be *packed types*, see Section 5, since their bit layout is platform independent. This allows development machines to easily construct properly formatted parameters no matter what machine is running the Adamant embedded system. A default value must also be provided for each parameter. The parameter within the component will be initialized to this value on startup and will only be overridden when a new value is set from a parameter managing component externally.

The presence of a parameter model causes the Adamant build system to produce many more build rules (some output omitted for brevity):

```
> redo what
redo  what
redo all
redo build/html/parameter_component_parameters.html
redo build/obj/Linux/parameter_component_parameters.o
redo build/src/parameter_component_parameters.adb
redo build/src/parameter_component_parameters.ads
```

We can see one object file that can be constructed, the parameter suite package, *build/obj/Linux/parameter_component_parameters.o*. To build the parameter suite package autocode we can run:

```
> redo build/obj/Linux/parameter_component_parameters.o
```

which will construct the package `Parameter_Component_Parameters` in `build/src/parameter_component_parameters.ads` and `build/src/parameter_component_parameters.adb`. Let's look at the specification.

parameter_component_parameters.ads:

```
1 -----  
2 -- Parameter_Component Parameters Spec  
3 --  
4 -- Generated from parameter_component.parameters.yaml on 2025-07-15 21:46.  
5 -----  
6  
7 -- Standard Includes  
8 with Parameter;  
9 with Parameter_Types;  
10  
11 -- Argument Includes  
12 with Packed_U16;  
13  
14 -- A set of parameters for the Parameter Component.  
15 package Parameter_Component_Parameters is  
16  
17     -- Object instance type:  
18     type Instance is tagged limited private;  
19  
20 -----  
21     -- Local Parameter Identifiers:  
22 -----  
23     Num_Parameters : constant Natural := 2;  
24     type Local_Parameter_Id_Type is (  
25         Start_Count_Id,  
26         Hello_World_Value_Id  
27     );  
28     for Local_Parameter_Id_Type use (  
29         Start_Count_Id => 0,  
30         Hello_World_Value_Id => 1  
31     );  
32  
33 -----  
34     -- Setter procedure for parameter ID base:  
35 -----  
36     not overriding function Get_Id_Base (Self : in Instance) return  
37         Parameter_Types.Parameter_Id  
38             with Inline => True;  
39     not overriding procedure Set_Id_Base (Self : in out Instance; Id_Base : in  
40         Parameter_Types.Parameter_Id)  
41             with Inline => True;  
42  
43 -----  
44     -- Getter function for global parameter IDs:  
45 -----  
46     not overriding function Get_Start_Count_Id (Self : in Instance) return  
47         Parameter_Types.Parameter_Id  
48             with Inline => True;  
49     not overriding function Get_Hello_World_Value_Id (Self : in Instance) return  
49         Parameter_Types.Parameter_Id  
49             with Inline => True;  
50  
51 -----  
52     -- Parameter creation functions:  
53 -----
```

```

50   -----
51   -- The value to start counting at.
52   not overriding function Start_Count (Self : in Instance; Arg : Packed_U16.T)
53     => return Parameter.T;
54   -- The value of count at which the Hello, World! string is printed.
55   not overriding function Hello_World_Value (Self : in Instance; Arg :
56     => Packed_U16.T) return Parameter.T;
57
58   -- Compile time checks to make sure types do not serialize longer than the
59   -- parameter buffer size:
60   pragma Warnings (Off, "condition can only be True if invalid values
61   -- present");
62   pragma Compile_Time_Error (
63     Packed_U16.Size_In_Bytes > Parameter_Types.Parameter_Buffer_Type'Length,
64     "Parameter 'Start_Count' has argument of type 'Packed_U16.T' which has a
65     -- maximum serialized length larger than the buffer size of Parameter.T.");
66   pragma Compile_Time_Error (
67     Packed_U16.Size_In_Bytes > Parameter_Types.Parameter_Buffer_Type'Length,
68     "Parameter 'Hello_World_Value' has argument of type 'Packed_U16.T' which
69     -- has a maximum serialized length larger than the buffer size of
70     -- Parameter.T."
71   );
72   pragma Warnings (On, "condition can only be True if invalid values present");
73
74 private
75   type Instance is tagged limited record
76     Id_Base : Parameter_Types.Parameter_Id := 1;
77   end record;
78 end Parameter_Component_Parameters;

```

Unlike for data products, packets, and events, this package is not intended to be used by the component that it is associated with. This package is used to *create* parameters, not *use* them, and thus is most commonly used to generate parameters while unit testing the component. The parameters are stored and updated within the component itself using special autocode within the component base package, see the following section. Let's explore the package's contents, remembering that it will most likely be utilized by a tester component during unit test.

First, we can see that there is a `Instance` type, which is the parameter suite tagged type. There is also a public `local_Parameter_Id_Type` enumeration which declares the *local IDs* for the parameters. When instantiated in a component, these local IDs will be added to the component's *parameter ID base*. The parameter ID base can be set through the public function `set_Id_Base` which is called during component initialization, see Section 6.7.3. Note that the ID base for the parameter suite is stored in the `Instance` record in the private section in the `id_Base` variable.

Next, there is a set of two functions which return the ID for each parameter in the parameter suite. The returned ID will already have the ID base offset applied to the local ID. Finally, there is a set of two functions that are used to create the two parameters.

The most commonly used functions are the parameter creation functions. Each function returns a type of `Parameter.T` and takes the parameter type as input. The implementation of these functions are not shown here but can be explored by the reader in `parameter_component_parameters.adb`. These functions take the parameter type and serialize it into the parameter buffer. The parameter is returned with the correct ID applied. To see how these functions are used within a unit test see Section 6.14.5.

6.14.4 Updating and Using Parameters

When a parameter model is found for a component, the component base package `Instance` record is altered to contain those parameter variables. For our `parameter_component`, two variables are added to the base package `Instance` record: `Start_Count` and `Hello_World_Value`. These variables can be accessed just like any other component variable, ie. via `Self.Start_Count` and `Self>Hello_World_Value`. The only difference is that these variables can be updated or fetched by a call to the component's `Parameter_Update.T` modify connector.

Note that if the type of the parameter is a *packed type*, then the `U` version of the type is used within the component base package `Instance` record for the best performance. Since the `U` type uses the machine native representation of the variable, it often provides faster access than the packed type.

The two variables added to the base class are called the *working* parameters, because these are the values that are directly used by the component during execution. In addition to the *working* parameters, there is also a set of protected *staged* parameters that assist in the updating of the *working* parameter values in a thread-safe and performant manner. The *staged* parameters act as a temporary holding place for updated parameter values, and are copied to the *working* values atomically, when the component deems it is safe to do so. The details of this process are discussed in the remainder of this section.

To handle parameter updating, the component base package autocode is altered to include additional logic to process the `Parameter_Update.T` data. As seen in Section 6.14.2, the `Parameter_Update.T` type contains a field `Operation` which determines the type of processing that the component base package needs to perform. The processing performed for the different Operations are as follows:

- **Stage** - The incoming parameter value is checked for validity in ID, length, and range and then stored within a temporary *staged* variable within the component.
- **Update** - This operation indicates that all new parameter values have been *staged* and are ready to be copied to the *working* variables within the component.
- **Fetch** - The *staged* value of the parameter with the indicated ID is returned to the invoker.

The `Fetch` operation is the most straightforward. The current *staged* value of the parameter is simply returned to the caller. The `Stage` and `Update` operations are a bit more sophisticated and are described further.

When a parameter `Stage` operation is received by a component, the component base package does the following:

1. The parameter ID is checked for validity. If it is out of range an `Id_Error` return status is set.
2. The parameter length is checked for validity. If it does not match the expected length a `Length_Error` return status is set.
3. The parameter type is checked for validity by making sure it is within the Ada range(s) defined for that type. If it is not valid, then a `Validation_Error` return status is set.
4. If all the previous checks pass, the parameter is “staged”, ie. stored in a *staged* variable.
5. The set parameter status is returned to the parameter update invoker.

The reader can verify this logic by building and inspecting `build/src/component-parameter_component.ads` and `build/src/component-parameter_component.adb`.

Note that after validation, a parameters is not immediately copied over to the component `Instance` variables holding the parameters. Instead the parameters are “staged”, ie. copied to a temporary staging variable. This is done for a few reasons:

1. We do not want to update the component's parameters at the same time the component is executing. This could cause unintended behavior. Instead, we want to let the component implementation itself determine when the best time to update parameters is.
2. We want to make sure that updating all of the component's parameters is atomic, that is, all parameters are updated together. We don't want to get into a state where one parameter is updated, but another is still not updated. This could also cause unintended behavior.

To circumvent these issues, parameters are staged. A component is alerted that all new values for parameters have been staged when the `Update` operation is received on the `Parameter_Update.T` connector. Once alerted, the component is able to atomically update its local parameter variables stored in `Instance` by making a call to the autocoded base package procedure `Update_Parameters`. This procedure will copy all the parameters from the staging area to their working local copy in `Instance`. The call to this function will only execute this logic after all parameters for the component have been staged, ie. once the `Update` operation has been received, making sure the parameters are updated atomically.

The following is the hand-coded implementation package body for the `parameter_component` which demonstrates how to properly use parameters within a component. For directions on generating the templates for this file see Section 6.1.4.

component-parameter_component-implementation.adb:

```

1-----  

2-- Parameter_Component Component Implementation Body  

3-----  

4  

5 with Ada.Text_IO; use Ada.Text_IO;  

6  

7 package body Component.Parameter_Component.Implementation is  

8  

9-----  

10-- Invokee connector primitives:  

11-----  

12-- The parameter update connector.  

13 overriding procedure Parameter_Update_T_Modify (Self : in out Instance; Arg :  

14  → in out Parameter_Update.T) is  

15 begin  

16   -- Process the parameter update, staging or fetching parameters as  

17   -- requested.  

18   Self.Process_Parameter_Update (Arg);  

19 end Parameter_Update_T_Modify;  

20  

21-- This connector provides the schedule tick for the component.  

22 overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in  

23  → Tick.T) is  

24   Ignore : Tick.T renames Arg;  

25 begin  

26   -- If there are any staged updates to our parameters, then perform the  

27   -- copy now to update  

28   -- our local parameter values.  

29   Self.Update_Parameters;  

30  

31   -- Print hello world if our count has reached the parameter  

32   -- value.  

33   if Self.Count = Self.Hello_World_Value.Value then  

34     Put_Line ("Hello, World!");  

35   end if;  

36  

37   -- Increment the count:  


```

```

34     Self.Count := @ + 1;
35 end Tick_T_Recv_Sync;
36
37 -- This procedure is called when the parameters of a component have been
-- updated. The default implementation of this
38 -- subprogram in the implementation package is a null procedure. However,
-- this procedure can, and should be implemented if
39 -- something special needs to happen after a parameter update. Examples of
-- this might be copying certain parameters to
40 -- hardware registers, or performing other special functionality that only
-- needs to be performed after parameters have
41 -- been updated.
42 overriding procedure Update_Parameters_Action (Self : in out Instance) is
43 begin
44     -- Set the current count to the start count value every time the
        -- component's parameters
        -- are updated.
45     Self.Count := Self.Start_Count.Value;
46 end Update_Parameters_Action;
47
48 -----
49 -- Parameter handlers:
50 -----
51 -- Description:
52 -- A set of parameters for the Parameter Component.
53 -- Invalid Parameter handler. This procedure is called when a parameter's
        -- type is found to be invalid:
54 overriding procedure Invalid_Parameter (Self : in out Instance; Par : in
        -- Parameter.T; Errant_Field_Number : in Unsigned_32; Errant_Field : in
        -- Basic_Types.Poly_Type) is
55 begin
56     Put_Line ("Oh, no! A parameter was sent that was malformed.");
57 end Invalid_Parameter;
58
59 end Component.Parameter_Component.Implementation;

```

The first thing to notice is the implementation of the Parameter_Update_T_Modify connector handler. This code is completely generated by the template autocoder, and rarely needs to be modified by the developer. It calls Process_Parameter_Update which is an autocoded procedure in the base package that performs the procedure described above, processing the parameter update based on the specified Operation. Returned from this call is the parameter status which is returned to the connector caller (via in out procedure parameter).

The next procedure, Tick_T_Recv_Sync represents the main execution of the component. In this procedure, we first call Self.Update_Parameters, which is implemented in the component base package. As described earlier, if all the parameters for the component have been staged through calls to Parameter_Update_T_Modify from an external component (as indicated by the Update operation), then this function will atomically copy those staged parameters to their working copies in the component Instance record. If an Update has not yet been received, then the call to Update_Parameters does nothing and returns instantly.

Next, we access one of the component's parameters, Self.Hello_World_Value.Value, and compare it against our Self.Count variable, which is incremented every time a Tick.T is received. If Self.Count is equal to our parameter value, then the famous string is printed.

The standard practice when using parameters within components is to call Self.Update_Parameters at the beginning of execution to make sure that all parameters are up to date before continuing on. Once the parameters are atomically updated, they can be used freely.

The next procedure that is defined is `Update_Parameters_Action`. This procedure gets called any time the parameters have been successfully updated via a call to `Self.Update_Parameters`. Note that it is not called if not all the parameters have been staged yet, ie. an `Update` operation has not yet been received. By default, the implementation of this procedure is `null`, and it rarely needs to actually be implemented by the developer. The purpose of the procedure is to allow the developer to implement an action that runs whenever the parameters have been updated. This is often used to keep hardware state in sync with software parameters. For instance, one can use this procedure to write parameter values to hardware registers. In this way, you can always ensure that if the software parameter values get updated, so do the analogous hardware registers.

In this case, we use the `Update_Parameters_Action` procedure to update our local variable `Self.Count` with the parameter value `Self.Start_Count.Value` anytime the parameters are updated. In this way, the parameters can be used to set `Self.Count` to any value during execution.

6.14.5 Parameter Unit Testing

This section assumes you know how to unit test component which is described in Section 6.2. The details of creating a unit test model, generating tester component, and implementing unit tests are not repeated here. Instead, this section will show the important differences involved when unit testing a component that has parameters.

Below is a unit test implementation package that provides a single unit test, `Unit_Test`, for the *parameter_component*.

parameter_component_tests-implementation.adb:

```

1  -----
2  -- Parameter_Component Tests Body
3  -----
4
5  with Parameter.Enums; use Parameter.Enums.Parameter_Update_Status;
6  with Parameter.Enums.Assertion; use Parameter.Enums.Assertion;
7  with Parameter.Assertion; use Parameter.Assertion;
8
9  package body Parameter_Component_Tests.Implementation is
10
11  -----
12  -- Fixtures:
13  -----
14
15  overriding procedure Set_Up_Test (Self : in out Instance) is
16  begin
17      -- Make necessary connections between tester and component:
18      Self.Tester.Connect;
19
20      -- Call the component set up method that the assembly would normally call.
21      Self.Tester.Component_Instance.Set_Up;
22  end Set_Up_Test;
23
24  overriding procedure Tear_Down_Test (Self : in out Instance) is
25  begin
26      null;
27  end Tear_Down_Test;
28
29  -----
30  -- Tests:
31  -----
32
```

```

33 overriding procedure Unit_Test (Self : in out Instance) is
34     Status : Parameter.Enums.Parameter_Update_Status.E;
35     Param : Parameter.T;
36 begin
37     -- Send some ticks:
38     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
39     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
40
41     -- Stage the unit test parameter:
42     Status := Self.Tester.Stage_Parameter (Self.Tester.Parameters.Start_Count
43         => ((Value => 5)));
44     Parameter_Update_Status_Assert.Eq (Status, Success);
45
46     -- Fetch the parameter to make sure the parameter value
47     -- inside the component is what we just set it to.
48     Status := Self.Tester.Fetch_Parameter
49         => (Self.Tester.Parameters.Get_Start_Count_Id, Param);
50     Parameter_Update_Status_Assert.Eq (Status, Success);
51     Parameter_Assert.Eq (Param, Self.Tester.Parameters.Start_Count ((Value =>
52         => 5)));
53
54     -- Update the next unit test parameter:
55     Status := Self.Tester.Stage_Parameter
56         => (Self.Tester.Parameters.Hello_World_Value ((Value => 7)));
57     Parameter_Update_Status_Assert.Eq (Status, Success);
58
59     -- Fetch the parameter to make sure the parameter value
60     -- inside the component is what we just set it to.
61     Status := Self.Tester.Fetch_Parameter
62         => (Self.Tester.Parameters.Get_Hello_World_Value_Id, Param);
63     Parameter_Update_Status_Assert.Eq (Status, Success);
64     Parameter_Assert.Eq (Param, Self.Tester.Parameters.Hello_World_Value
65         => ((Value => 7)));
66
67     -- Now tell the component that we are done staging parameters
68     -- so that they can be updated by the component.
69     -- This will send a parameter to the component with an "Update"
70     -- operation. This tells the base package that all parameters
71     -- have been sent and it is safe to do an update.
72     Parameter_Update_Status_Assert.Eq (Self.Tester.Update_Parameters,
73         => Success);
74
75     -- After 3 more ticks, hello world should be printed.
76     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
77     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
78     Self.Tester.Tick_T_Send ((Time => (0, 0), Count => 0));
79 end Unit_Test;
80
81 end Parameter_Component_Tests.Implementation;

```

If a component has a parameter model, the autocoded tester package includes some modifications to make unit testing parameters simpler. Included in the tester base package `Instance` record is a variable called `Parameters` which is of the type `Parameter_Component_Parameters.Instance`. As can be seen in the unit test code above, the `Self.Tester.Parameters` variable is used to construct parameters.

In addition to `Self.Tester.Parameters`, three helper functions are autocoded into the tester component implementation package to aid in the unit testing of parameters. They are described below:

- **Stage_Parameter** - Stage a parameter within the component under test.
- **Update_Parameters** - Send the Update operation to the component under test, allowing it to copy parameter values from *staged* to *active* upon a call to `Update_Parameters`.
- **Fetch_Parameter** - Get the value of a parameter (the *staged* copy) within the component under test.

If you look at the autocoded tester implementation package body (not shown here for brevity), you can see that the implementation of these helper functions simply calls the tester component's `Parameter_Update.T` provide connector with the appropriate `Operation` set, and then returns the resulting status to the caller. Using these helper functions keeps unit test code that deals with parameters small and concise.

The unit test above begins by staging the component's two parameters, and subsequently fetching them to ensure that the stage worked correctly. This is achieved through calls to `Stage_Parameter` and `Fetch_Parameter`. For each, the status is checked to ensure that `Success` is returned.

After the parameters are staged, the special procedure `Self.Tester.Update_Parameters` is called. This special parameter sends the `Update` operation to the component, signaling that the component's *working* parameters can now be atomically updated by copying from the *staged* via the `Update_Parameters` procedure within the component.

Finally, we can verify the parameters were successfully updated by the unit test code by running it:

```
> redo test
```

which produces the output:

```
Hello, World!
```

```
OK Unit_Test
```

```
Total Tests Run:    1
Successful Tests:  1
Failed Assertions: 0
Unexpected Errors: 0
```

6.14.6 Parameter Documentation

Adamant provides automatic generation of documentation for any parameter model. Currently two versions of documentation can be created: HTML, which is useful for presentation in meetings or for quick reference, and PDF (via `LATEX`), which is useful for formal documentation. PDF documentation for parameters is created as part of component documentation and is not discussed here. See Section 6.16 for details.

To build HTML documentation run:

```
> redo build/html/parameter_component_parameters.html
```

which when opened with your favorite web browser looks something like this:

Parameter_Component Parameter Suite

Description: A set of parameters for the Parameter Component.

Parameter Name	Type	Default Value	Description
Start_Count	Packed_U16.T	(Value => 0)	The value to start counting at.
Hello_World_Value	Packed_U16.T	(Value => 16)	The value of count at which the Hello, World! string is printed.

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/parameter_component/parameter_component.parameters.yaml on 2020-05-06 19:25.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

6.15 Component Faults

Faults are component *IDed entities*. Please read Section 6.8 before continuing on to this section.

A *fault* in Adamant is a small asynchronously created packet that signifies that some off-nominal behavior or error condition has been encountered. Faults are usually produced in response to limit checking of telemetry, monitoring of loss on a communication channel, or due to some other anomalous condition. As a result, they are usually produced rarely and sporadically.

6.15.1 The Fault Type

The fault type in Adamant is defined in *src/core/types/fault*. A fault consists of a header followed by a byte array used to serialize the specific data, parameters, associated with the fault. A fault is a *variable length packed record*, see Section 5.1.11, whose length is specified by the length field in its header. The following tables show the bit layout of a fault and its header.

Generic fault header.

Table 25: Fault_Header Packed Record : 88 bits

Name	Type	Range	Size (Bits)	Start Bit	End Bit
Time	Sys_Time.T	-	64	0	63
Id	Fault_Types.Fault_Id	0 to 65535	16	64	79
Param_Buffer_Length	Fault_Types.Parameter_Buffer_Length_Type	0 to 8	8	80	87

Field Descriptions:

- **Time** - The timestamp for the fault.
- **Id** - The fault identifier
- **Param_Buffer_Length** - The number of bytes used in the param buffer

Generic fault packet for holding arbitrary faults.

Table 26: Fault Packed Record : 152 bits (*maximum*)

Name	Type	Range	Size (Bits)	Start Bit	End Bit	Variable Length
Header	Fault_Header.T	-	88	0	87	-

Param_Buffer	Fault_Types.Parameter_Buffer_Type	-	64	88	151	Header.Param_Buffer_Length
--------------	-----------------------------------	---	----	----	-----	----------------------------

Field Descriptions:

- **Header** - The fault header
- **Param_Buffer** - A buffer that contains the fault parameters

As can be seen, in the fault header there is a 64-bit timestamp which signifies when the fault was encountered followed by a unique identifier for the fault. There is also a byte array where the fault's parameters are serialized.

6.15.2 Creating a Fault Model

In Adamant, faults are produced by components. A component may produce many different faults. To demonstrate this we will create a component called the *fault_component* using the following commands:

```
> mkdir fault_component # make component directory
> cd fault_component
> touch .all_path          # add directory to build path
```

Next we will create the component by writing the following YAML file in this directory:

fault_component.component.yaml:

```

1  ---
2  execution: passive
3  description: This is the fault component, which sends faults.
4  connectors:
5    - type: Tick.T
6      kind: recv_sync
7      description: This connector provides the schedule tick for the component.
8    - return_type: Sys_Time.T
9      kind: get
10     description: This connector is used to fetch the current system time.
11    - type: Fault.T
12      kind: send
13      description: This connector is used to send out a fault.
```

which looks like this:



To configure the component with faults we use a fault model file. To create a fault model for this component, we simply need to name the fault model appropriately. Fault models will always be of the form *component_name.faults.yaml* where *component_name* is the component that will be producing the faults. Only a single fault model can be configured for a component, and a single fault model cannot be shared by more than one component. In this directory we create a fault model named *fault_component.faults.yaml*. The contents are shown below:

fault_component.faults.yaml:

```
1  ---
2  # Optional - description of fault suite
3  description: A set of faults for the Fault Component.
4  # Required - list of faults included in the fault suite
5  faults:
6    # Required - the name of the fault
7  - name: Discontinuous_Time_Fault
8    # Optional - a description of the fault
9    description: A discontinuous time was detected by the component.
10   # Optional - the parameter type to include with the fault
11   param_type: Packed_U32.T
12   # Optional - a statically defined ID for the fault
13   # If this is specified, then IDs must be specified for
14   # EVERY fault in this fault suite. The Set_Id_Base
15   # procedure will then NOT be used to set the fault IDs.
16   #
17   # If this is not specified, then NO faults in this fault
18   # suite may specify an ID. In this case the Set_Id_Base
19   # procedure will be used to set the packed IDs.
20   # id: 17
21   # ^ Note: We are not specifying global IDs in this example
22 - name: Zero_Time_Fault
23   description: A time restart at zero seconds epoch was detected by the
     ↳ component.
```

This model defines the *fault suite* for the *fault component*. It declares 2 different faults that can be sent by the component. Comments are provided to explain whether or not each field is optional or required. In this case, you must supply a name for each fault, and optionally a parameter that will be serialized with the fault. Only a single parameter type can be provided with a fault. However, that parameter can be a compound type such as an array or record. Parameters should always be *packed types*, see Section 5, since their bit layout is platform independent. This allows development machines to easily decode fault parameters no matter what machine is running the Adamant embedded system.

Similar to packets, a developer may also specify the ID for each fault. If specified, the ID refers to the global ID of the fault as defined in the assembly. In this case, there is no fault *id base*. This feature should only be used if you are constructing a mission-specific component, where you want to define static IDs for all the faults it creates. For reusable components, it is advised to not specify the ID for any faults, and to use the normal *Set_Id_Base* method for setting fault IDs, see Section 6.7.3. By not statically defining the fault IDs, the component remains reusable for future systems.

Note that if you define a static global ID for one fault, you MUST define the ID for every fault in the suite. Similarly, if you wish to not define the static IDs for the faults, you must NOT define the fault IDs for every fault in the fault suite.

The presence of a fault model causes the Adamant build system to produce many more build rules (some output omitted for brevity):

```
> redo what
redo  what
redo all
redo build/html/fault_component_faults.html
redo build/obj/Linux/fault_component_faults.o
redo build/src/fault_component_faults.adb
redo build/src/fault_component_faults.ads
```

We can see one object files that can be constructed, the fault suite package, *build/obj/Linux/fault_component_faults.o*. To build the fault suite package autocode we can run:

```
> redo build/obj/Linux/fault_component_faults.o
```

which will construct the package `Fault_Component_Faults` in `build/src/-fault_component_faults.ads` and `build/src/fault_component_faults.adb`. Let's look at the specification.

fault_component_faults.ads:

```
1 -----  
2 -- Fault_Component_Faults Spec  
3 --  
4 -- Generated from fault_component_faults.yaml on 2025-07-15 21:48.  
5 -----  
6  
7 -- Standard Includes:  
8 with Fault;  
9 with Fault_Types;  
10 with Sys_Time;  
11  
12 -- Parameter Includes  
13 with Packed_U32;  
14  
15 -- A set of faults for the Fault Component.  
16 package Fault_Component_Faults is  
17  
18     -- Object instance type:  
19     type Instance is tagged limited private;  
20  
21 -----  
22     -- Local Fault Identifiers:  
23  
24     Num_Faults : constant Natural := 2;  
25     type Local_Fault_Id_Type is (  
26         Discontinuous_Time_Fault_Id,  
27         Zero_Time_Fault_Id  
28     );  
29     for Local_Fault_Id_Type use (  
30         Discontinuous_Time_Fault_Id => 0,  
31         Zero_Time_Fault_Id => 1  
32     );  
33  
34 -----  
35     -- Setter procedure for fault ID base:  
36  
37     not overriding function Get_Id_Base (Self : in Instance) return  
38         & Fault_Types.Fault_Id  
39             with Inline => True;  
40     not overriding procedure Set_Id_Base (Self : in out Instance; Id_Base : in  
41         & Fault_Types.Fault_Id)  
42             with Inline => True;  
43  
44 -----  
45     -- Getter function for global Fault IDs:  
46  
47     not overriding function Get_Discontinuous_Time_Fault_Id (Self : in Instance)  
48         & return Fault_Types.Fault_Id  
49             with Inline => True;  
50     not overriding function Get_Zero_Time_Fault_Id (Self : in Instance) return  
51         & Fault_Types.Fault_Id  
52             with Inline => True;
```

```

50  -----
51  -- Fault creation functions:
52  -----
53  -- A discontinuous time was detected by the component.
54  not overriding function Discontinuous_Time_Fault (Self : in Instance;
55   →   Timestamp : Sys_Time.T; Param : in Packed_U32.T) return Fault.T;
56
57  -- A time restart at zero seconds epoch was detected by the component.
58  not overriding function Zero_Time_Fault (Self : in Instance; Timestamp :
59   →   Sys_Time.T) return Fault.T;
60
61  -- Compile time checks to make sure types do not serialize longer than the
62   →   fault buffer size:
63  pragma Warnings (Off, "condition can only be True if invalid values
64   →   present");
65  pragma Compile_Time_Error (
66    Packed_U32.Size_In_Bytes > Fault_Types.Parameter_Buffer_Type'Length,
67    "Fault 'Discontinuous_Time_Fault' has argument of type 'Packed_U32.T'
68    →   which has a maximum serialized length larger than the buffer size of
69    →   Fault.T."
70  );
71  pragma Warnings (On, "condition can only be True if invalid values present");
72
73  private
74    type Instance is tagged limited record
75      Id_Base : Fault_Types.Fault_Id := 0;
76    end record;
77
78  end Fault_Component_Faults;

```

First we can see that there is a `Instance` type which is the fault suite tagged type. There is also a public `local_Fault_Id_Type` enumeration which declares the *local IDs* for the faults. When instantiated in a component, these local IDs will be added to the component's *fault ID base*. The fault ID base can be set through the public function `set_Id_Base` which is called during component initialization, see Section 6.7.3. Note that the ID base for the fault suite is stored in the `Instance` record in the private section in the `id_Base` variable. Next, there is a set of 3 functions which return the ID for each fault in the fault suite. The returned ID will already have the ID base offset applied to the local ID. Finally, there is a set of three functions that are used to create the three faults.

The most commonly used functions are the fault creation functions. Each function returns a type of `Fault.T` and takes a timestamp and the fault parameter type as input. The implementation of these functions are not shown here but can be explored by the reader in `fault_component_faults.adb`. These functions take the timestamp and fault parameter and serialize them into the fault header and buffer, respectively. The fault is returned with the correct ID applied. To see how these functions are used within a component continue to the next section.

6.15.3 Using the Fault Package

When a fault model is found for a component, the component base package autocode is altered slightly to include a variable called `faults` which is an instantiation of the fault suite package instance, `Fault_Component_Faults.Instance`. The reader can verify this by building and inspecting `build/src/component-fault_component.ads`. This `faults` variable can then be used in the component implementation package to create faults.

The following is the hand-coded implementation package body for the `fault_component`. For directions on generating the templates for this file see Section 6.1.4.

component-fault_component-implementation.adb:

```

1 -----  

2 -- Fault_Component Component Implementation Body  

3 -----  

4  

5 package body Component.Fault_Component.Implementation is  

6  

7 -----  

8 -- Invokee connector primitives:  

9 -----  

10 -- This connector provides the schedule tick for the component.  

11 overriding procedure Tick_T_Recv_Sync (Self : in out Instance; Arg : in  

12   → Tick.T) is  

13   -- Get the system time:  

14   Timestamp : constant Sys_Time.T := Self.Sys_Time_T_Get;  

15   -- Extract the seconds from the tick:  

16   Seconds : Interfaces.Unsigned_32 renames Arg.Time.Seconds;  

17 begin  

18   -- Check the timestamp in the tick. If it has zero seconds, this is  

19   -- unexpected and  

20   -- we should send a fault:  

21   if Seconds = 0 then  

22     Self.Fault_T_Send (Self.Faults.Zero_Time_Fault (Self.Sys_Time_T_Get));  

23   end if;  

24  

25   -- If the last seconds value we received was not zero, then we  

26   -- should check for a time discontinuity. We expect the received  

27   -- time to be 1-2 seconds ahead of the last tick we received.  

28   if Self.Last_Received_Seconds /= Interfaces.Unsigned_32'First and then  

29     (Seconds > Self.Last_Received_Seconds + 2 or else  

30     Seconds < Self.Last_Received_Seconds + 1)  

31   then  

32     -- Discontinuous time detected, send fault.  

33     Self.Fault_T_Send (Self.Faults.Discontinuous_Time_Fault (Timestamp,  

34     → (Value => Seconds)));  

35   end if;  

36  

37   -- Save off the last seconds value received:  

38   Self.Last_Received_Seconds := Seconds;  

39 end Tick_T_Recv_Sync;  

40  

41 end Component.Fault_Component.Implementation;

```

As can be seen, the component record contains the variable `Self.Last_Received_Seconds` in which it saves off the last received `Seconds` value from the system time contained within the received `Tick.T`. The component does various error checking on this received time to determine if it is valid or not. The two error cases it looks for are a zero time (meaning time has reset) and a discontinuous time. If either of these conditions are encountered then the component will produce and send out a fault. Faults are constructed using the fault creation functions provided by `Self.Faults`, which is an instantiation of `Fault_Component_Faults.Instance`. Faults are created and then sent out of the `Fault_T_Send` connector.

6.15.4 Fault Unit Testing

This section assumes you know how to unit test component which is described in Section 6.2. The details of creating a unit test model, generating tester component, and implementing unit tests are not repeated here. Instead, this section will show the important differences involved when unit testing a component that has faults.

Below is a unit test implementation package that provides a single unit test, `Unit_Test`, for the *fault_component*.

fault_component_tests-implementation.adb:

```
1  -----
2  -- Fault_Component Tests Body
3  -----
4
5  with Basic_Assertions; use Basic_Assertions;
6  with Packed_U32.Assertion; use Packed_U32.Assertion;
7
8  package body Fault_Component_Tests.Implementation is
9
10   -----
11  -- Fixtures:
12  -----
13
14  overriding procedure Set_Up_Test (Self : in out Instance) is
15  begin
16    -- Allocate heap memory to component:
17    Self.Tester.Init_Base;
18
19    -- Make necessary connections between tester and component:
20    Self.Tester.Connect;
21
22    -- Call the component set up method that the assembly would normally call.
23    Self.Tester.Component_Instance.Set_Up;
24  end Set_Up_Test;
25
26  overriding procedure Tear_Down_Test (Self : in out Instance) is
27  begin
28    -- Free component heap:
29    Self.Tester.Final_Base;
30  end Tear_Down_Test;
31
32  -----
33  -- Tests:
34  -----
35
36  overriding procedure Unit_Test (Self : in out Instance) is
37  begin
38    -- A tick with a zero seconds timestamp should always produce a fault:
39    Self.Tester.Tick_T_Send ((Time => (0, 14), Count => 12));
40
41    -- Make sure we got a fault:
42    Natural Assert.Eq (Self.Tester.Fault_T_Recv_Sync_History.Get_Count, 1);
43    Natural Assert.Eq (Self.Tester.Zero_Time_Fault_History.Get_Count, 1);
44
45    -- If we send a non-zero seconds value we should not get a fault:
46    Self.Tester.Tick_T_Send ((Time => (1, 14), Count => 12));
47    Natural Assert.Eq (Self.Tester.Fault_T_Recv_Sync_History.Get_Count, 1);
48
49    -- If we send continuous seconds values we should not get a fault,
50    -- (within 1-2 seconds):
51    Self.Tester.Tick_T_Send ((Time => (2, 14), Count => 12));
52    Self.Tester.Tick_T_Send ((Time => (3, 14), Count => 12));
53    Self.Tester.Tick_T_Send ((Time => (5, 14), Count => 12));
54    Self.Tester.Tick_T_Send ((Time => (7, 14), Count => 12));
55    Natural Assert.Eq (Self.Tester.Fault_T_Recv_Sync_History.Get_Count, 1);
56
57    -- If we send non-continuous seconds value we should get a fault:
58    Self.Tester.Tick_T_Send ((Time => (7, 14), Count => 12));
59    Natural Assert.Eq (Self.Tester.Fault_T_Recv_Sync_History.Get_Count, 2);
```

```

60     Natural Assert.Eq (Self.Tester.Discontinuous_Time_Fault_History.Get_Count,
61     ↪  1);
62     Packed_U32 Assert.Eq (Self.Tester.Discontinuous_Time_Fault_History.Get
63     ↪  (1), (Value => 7));
64
65     -- Again.
66     Self.Tester.Tick_T_Send ((Time => (10, 14), Count => 12));
67     Natural Assert.Eq (Self.Tester.Fault_T_Recv_Sync_History.Get_Count, 3);
68     Natural Assert.Eq (Self.Tester.Discontinuous_Time_Fault_History.Get_Count,
69     ↪  2);
70     Packed_U32 Assert.Eq (Self.Tester.Discontinuous_Time_Fault_History.Get
71     ↪  (2), (Value => 10));
72   end Unit_Test;
73
74 end Fault_Component_Tests.Implementation;

```

If a component has a fault model, the autocoded tester package includes some modifications to make unit testing faults simpler. Just as with connectors, a distinct history is included for every fault defined in the component's fault model within the tester component. This history can be queried and assertions can be run against it in the same way you would a connector history.

As can be seen in the unit test code above, some `Tick.Ts` are sent to the component. The component is checking the `Seconds` value of the `Tick.T` timestamp for validity. For certain sequences of `Seconds` the unit test makes sure that the correct fault is produced. The total number of faults produced is checked using the fault connector history. The amount and content of each individual fault received is checked by asserting against the individual fault histories provided in the tester component.

To run this unit test code we use:

```
> redo test
```

which produces the output:

```

OK Unit_Test

Total Tests Run:    1
Successful Tests:  1
Failed Assertions: 0
Unexpected Errors: 0

```

6.15.5 Fault Documentation

Adamant provides automatic generation of documentation for any fault model. Currently two versions of documentation can be created: HTML, which is useful for presentation in meetings or for quick reference, and PDF (via L^AT_EX), which is useful for formal documentation. PDF documentation for faults is created as part of component documentation and is not discussed here. See Section 6.16 for details.

To build HTML documentation run:

```
> redo build/html/fault_component_faults.html
```

which when opened with your favorite web browser looks something like this:

Fault_Component Fault Suite

Description: A set of faults for the Fault Component.

Fault Name	Parameter Type	Description
Discontinuous_Time_Fault	Packed_U32.T	A discontinuous time was detected by the component.
Zero_Time_Fault	-	A time restart at zero seconds epoch was detected by the component.

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/fault_component/fault_component.faults.yaml on 2021-08-03 21:21.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

6.16 Component Documentation

One of the benefits of the extensive modeling required by Adamant is that rich, informative documentation can be generated with little extra effort. This documentation is beneficial at all stages of the development process, from working out the details of the design, to reviewing the implementation.

It is advisable to generate component documentation as soon as a solid component model has been developed for a component. In this way, the document can be used to explain the design to other engineers and gain feedback before the implementation has even begun. During the development process, the design document should periodically be regenerated to reflect any changes. Finally, when the component is complete, the design document should be reviewed to ensure that it contains all the information necessary for another engineer to use or make changes to the component. Sometimes this requires adding custom sections to the design document that the document autocoder would not be able to create on its own.

This section walks through how to generate documentation for components and points to good examples of component documentation that exists in the Adamant repository.

6.16.1 Component Requirements

One of the most important aspects to designing good software is to understand what the software needs to do. These features are often captured in written “requirements” which often stem from customer needs, design principles, or environmental constraints. When a good set of requirements has been distilled for a software component, the engineer can ensure that their design decisions are in line with what the software needs to accomplish. Finally, during test, requirements can be verified with the actual executing system to prove that the software does indeed adhere to functionality it is required to provide.

Currently in Adamant, requirements can be placed into a requirements model and included in the component design document. This is the currently supported functionality, but plans exist in the future to have a richer requirement feature set that includes tying requirements to the unit tests that fulfill them.

To specify requirements for a component we use a requirements model file. To create a requirements model for the `example_component` presented in previous sections, we simply need to name the model appropriately. Requirement models will always be of the form `component_name.requirements.yaml` where `component_name` is the component that will be fulfilling the requirements. Only a single requirements model can be configured for a component, and a single requirements model cannot be shared by more than one component. In the `example_component` directory we create a requirements model named `example_component.requirements.yaml`. The contents are shown below:

`example_component.requirements.yaml`:

```

1  ---
2  # Optional - description of requirements suite
3  description: The requirements for the Example Component are specified below.
4  # Required - list of requirements
5  requirements:
6    # Required - the requirement text, usually contains a "shall" statement
7    - text: The component shall be passive and have less than three connectors.
8      # Optional - a description of the requirement usually elaborating on the
9      # statement and
10     # providing rational
11     description: We want to be able to show a reader of the User Guide how a
12       ↳ passive component is put together to show the important parts of a
13       ↳ component without being distracted by the more complex details of queued
14       ↳ components, active components, or components with IDed entities.
15     - text: The component shall send a packet to any connected downstream
16       ↳ component whenever it is scheduled to run.
17     - text: The packet contents shall include a counter in the first data byte
18       ↳ that increments with every schedule tick received.

```

This model defines the *requirement suite* for the *example_component*. It declares 3 different requirements that are used to constrain the design of the component. Comments are provided to explain whether or not each field is optional or required. In this case, you must supply the `text` field for each requirement that usually contains a “shall” statement. Optionally, you may provide a `description` that elaborates or provides rational for the requirement text.

Requirement models in Adamant do not have a direct effect on the component’s code. Currently, they can be used to produce three outputs: CSV output, HTML documentation, and PDF documentation. PDF documentation for requirements is created as part of component documentation and is discussed in the next section.

To build HTML documentation run:

```
> redo build/html/example_component_requirements.html
```

which when opened with your favorite web browser looks something like this:

Example_Component Requirements

Description: The requirements for the Example Component are specified below.

Number	Text	Description
1	The component shall be passive and have less than three connectors.	We want to be able to show a reader of the User Guide how a passive component is put together to show the important parts of a component without being distracted by the more complex details of queued components, active components, or components with IDed entities.
2	The component shall send a packet to any connected downstream component whenever it is scheduled to run.	-
3	The packet contents shall include a counter in the first data byte that increments with every schedule tick received.	-

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/example_component/example_component.requirements.yaml on 2020-05-07 15:11.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

CSV output can be produced by running:

```
> redo build/html/example_component_requirements.csv
```

The CSV feature is currently incomplete and will need to be refined if it is going to be used heavily.

6.16.2 Component Design Document

One of the benefits of the extensive modeling required by Adamant is that rich, informative documentation can be generated with little extra effort. To generate documentation for a component, a component model must first exist. This section demonstrates how to generate (and modify) documentation for the *example_component* that has been discussed in previous sections.

Component documentation must be generated and stored in a subdirectory *doc/* of the component's main directory. In fact, Adamant only produces rules to build component documentation within a *doc/* directory. This convention makes it easy to locate the documentation for any component in the repository.

To create documentation for the *example_component* we first create the doc directory:

```
> mkdir doc # make doc/ directory under example_component/
> cd doc
```

After we are inside the new *doc/* directory we can view the build rules that are available.

```
> redo what
redo what
redo all
redo clean
redo clean_all
redo templates
redo publish
redo targets
redo prove
redo analyze
redo style
redo pretty
redo test_all
redo analyze_all
redo coverage_all
redo build/pdf/example_component_commands.pdf
redo build/pdf/example_component_connectors.pdf
redo build/pdf/example_component_data_products.pdf
redo build/pdf/example_component_description.pdf
redo build/pdf/example_componentEnums.pdf
redo build/pdf/example_component_events.pdf
redo build/pdf/example_component_init.pdf
redo build/pdf/example_component_interrupts.pdf
redo build/pdf/example_component_packets.pdf
redo build/pdf/example_component_preamble.pdf
redo build/pdf/example_component_requirements.pdf
redo build/pdf/example_component_stats.pdf
redo build/pdf/example_component_types.pdf
redo build/pdf/example_component_unit_test.pdf
redo build/template/example_component.tex
redo build/tex/example_component_commands.tex
redo build/tex/example_component_connectors.tex
redo build/tex/example_component_data_products.tex
redo build/tex/example_component_description.tex
redo build/tex/example_componentEnums.tex
redo build/tex/example_component_events.tex
redo build/tex/example_component_init.tex
redo build/tex/example_component_interrupts.tex
```

```

redo build/tex/example_component_packets.tex
redo build/tex/example_component_preamble.tex
redo build/tex/example_component_requirements.tex
redo build/tex/example_component_stats.tex
redo build/tex/example_component_types.tex
redo build/tex/example_component_unit_test.tex

```

As can be seen, there are many different *.tex* files that can be created, each that can be used to produce a small PDF stub of that piece of documentation. Most importantly there is a rule to build the documentation template, which is the master L^AT_EX file for the component documentation. To build and use the template we run:

```

> redo build/template/example_component.tex
> cp build/template/* .  # copy the template file into doc/

```

This template provides an excellent start to our component documentation and often provides sufficient detail to be used without modification. However, like all template files in Adamant, this is meant to be modified, if necessary, to add additional hand-written documentation. Let's take a look at the autogenerated *example_component.tex* file.

example_component.tex:

```

1 \input{common_packages.tex}
2
3 \begin{document}
4
5 \title{\textbf{Example Component} \\
6 \large\textit{Component Design Document}}
7 \date{}
8 \maketitle
9
10 \section{Description}
11 \input{build/tex/example_component_description.tex}
12
13 \section{Requirements}
14 \input{build/tex/example_component_requirements.tex}
15
16 \section{Design}
17
18 \subsection{At a Glance}
19 \input{build/tex/example_component_stats.tex}
20
21 \subsection{Diagram}
22 \begin{figure}[H]
23   \includegraphics[width=1.0\textwidth,center]{../build/eps/example_component.e_}
24   \leftrightarrow ps
25   \caption{Example Component component diagram.}
26 \end{figure}
27
28 \subsection{Connectors}
29 \input{build/tex/example_component_connectors.tex}
30
31 \subsection{Initialization}
32 \input{build/tex/example_component_init.tex}
33
34 \section{Unit Tests}
35 \input{build/tex/example_component_unit_test.tex}
36

```

```

37 \section{Appendix}
38 \subsection{Packed Types}
39
40 \input{build/tex/example_component_types.tex}
41
42 \end{document}

```

As can be seen, this file defines the sections of the component documentation and imports most of the text from external autogenerated *.tex* files. Feel free to modify this file to add extra sections to the component documentation.

To generate a PDF from this file run:

```
> redo build/pdf/example_component.pdf
```

By default, PDF files are generated in the *build/pdf/* directory. This allows the developer to review the output documentation before “publishing” it, ie. saving it in the repository. Once the documentation looks good, you can publish it manually by copying it into the *doc/* directory:

```
> cp build/pdf/example_component.pdf .
```

or alternatively you can run the special Adamant command:

```
> redo publish
```

which will build the documentation and perform the copy for you.

The produced documentation is not shown here for brevity. Below is a list of component documentation that can be found in the Adamant repository, each which demonstrates different implementation:

- [Example Component Design Document](#) - The design document for the *example_component* that was generated in this section.
- [Command Router Design Document](#) - Demonstrates how to add a “context” diagram to the component documentation.
- [CCSDS Router Design Document](#) - Demonstrates how to add many “context” diagrams to the component documentation to demonstrate different use cases for the component.

6.17 Component Metrics

It is often valuable to keep track of certain code metrics such as the number of lines of code, packages, files, etc. for each component and component unit test. Adamant provides the ability to generate metrics for any executable, file, or object in the system. Steps to produce different metrics Adamant are discussed in Section 9.5. The same example there can be applied to component packages and unit tests.

6.18 Advanced Component Topics

This section presents component design topics that are for more experienced Adamant users. Many of these features are not as commonly employed as the features described in previous sections. Examples may not be as in depth and may require more exploratory work by the reader. Components within the Adamant repository that demonstrate the features presented will be referenced to aid in understanding.

6.18.1 Generic Components

Adamant provides a feature that allows for a component to operate on one or more generic parameters. This uses the underlying Ada feature for [generic packages](#). The advantage of defining a component with generic parameters is that these parameters can be used as connector types, allowing the component to operate on data in a generic manner. The actual type that the component operates on is not determined until the generic component is instantiated in an assembly. If components can be generic, they are much more likely to be reusable.

Two good example components that use this feature in Adamant are the the *logger* component, which logs data of a generic type to an in-memory circular buffer, *src/components/loggers/logger*, and the *splitter* component, which can split any *send* connector into an array of outputs, *src/components/splitter*. The splitter component is simple and demonstrates a minimal usage of the generic feature, so it is ideal to use as an example for this section of the document.

Let's begin by taking a look at the *splitter* component model:

splitter.component.yaml:

```
1 ---  
2 description: This is a generic component that can be used to split a single  
→ connector of any type into an arrayed connector of that type. This can be  
→ useful when a component has a single send connector, but you actually need  
→ the data to go to many different places simultaneously. In this case, the  
→ splitter component can be attached to the send connector and then distribute  
→ the data to many downstream components.  
3 execution: passive  
4 generic:  
5   description: The splitter is generic in that it can be instantiated to split a  
→ stream of any type at compile time.  
6   parameters:  
7     - name: T  
8       description: The generic type of data passed in and out of the splitter.  
9   connectors:  
10    - description: The generic invokee connector.  
11      type: T  
12      kind: recv_sync  
13    - description: The generic arrayed invoker connector.  
14      type: T  
15      kind: send  
16      count: 0 # Unconstrained array; size to be determined by assembly.
```

As can be seen, this component uses the YAML tag `generic` to define the component's generic types. An optional `description` is provided followed by a list of generic parameters. Each parameter must include a parameter name and optionally a `description`.

Each parameter can also specify two other tags that are not specified for the *splitter* component, `formal_type` and `optional`. If not specified, the `formal_type` for the generic is assumed to be `Type T is private;`. This formal generic type has specific semantic meaning in Ada. For a list of the different formal types available in Ada see the table found in the [Ada Wiki](#). As can be seen from the link, the default `formal_type` defines a generic parameter that can be any nonlimited definite type. To override this conservative generic formal type, you can specify the `formal_type` field manually. See the *logger* component for an example of manually defining a `formal_type`.

If the `formal_type` field includes a default value, you can inform the Adamant modeling system of this by setting the `optional` field to `True`. This will enable a user to instantiate the component without defining a concrete type for the generic parameter, using the provided default instead.

For the *splitter* the name of the generic type is `T`. As you can see, the type `T` is then used for all of

the *splitter*'s connectors, making them of generic type.

If you build the component base package specification for the splitter component via:

```
> redo build/src/component-splitter.ads
```

you will see that the package is generic and contains the generic parameter T. The package is not shown here for brevity.

The generic type T is then used in the implementation package of the component. Below is the implementation body which shows the behavior of the *splitter* component.

component-splitter-implementation.adb:

```
1 -----  
2 -- Splitter Component Implementation Body  
3 -----  
4  
5 package body Component.Splitter.Implementation is  
6  
7 -----  
8 -- Invokee connector primitives:  
9 -----  
10 -- The generic invokee connector.  
11 overriding procedure T_Recv_Sync (Self : in out Instance; Arg : in T) is  
12 begin  
13     -- Send T to any output connector that we have that is currently  
14     -- connected:  
15     for Index in Self.Connector_T_Send'Range loop  
16         Self.T_Send_If_Connected (Index, Arg);  
17     end loop;  
18 end T_Recv_Sync;  
19  
end Component.Splitter.Implementation;
```

As can be seen, the component takes in a variable of type T and then forwards this out to all of its arrayed connectors.

Instantiation of a generic component is done in the assembly model. Below is an example instantiation of the *splitter* component, using Event.T as the generic type.

```
1 components:  
2     - type: Splitter  
3         description: This component splits the event stream, sending events to 3  
4             different outputs.  
5         generic_types:  
6             - "T => Event.T"  
7         init_base:  
8             - "T_Send_Count => 3"  
9         ...  
10    connections:  
11        ...  
12        etc.
```

The new tag here is generic_types, which contains a list of the generic parameter names and the types to instantiate them with. Similar to the init_base tag, all generic_types must be listed

as id-value pairs of the form “generic_Parameter_Name => Type”. What we have done here is turned our generic *splitter* into a splitter specialized for splitting events. We can easily define a few more splitters in our assembly model for any other type we wish to split. This is the power of generic components.

Note that special care needs to be taken when defining a generic connector type that is of kind *recv_async*. This implies that the generic type will get serialized onto the component’s internal queue. Usually in Adamant, if an *recv_async* connector is called, the type gets serialized onto the component’s internal queue. If the type is a packed record of variable length, then only the bytes used in the type (usually specified by a length field in a header) will be serialized onto the queue. The same behavior is not implemented by default when using a generic type on a *recv_async* connector. By default, the maximum size of the generic type will always get serialized onto the queue. The reason for this is that the component has no way of determining the serialized length of the type, unless you explicitly provide a way.

For example, if a component with a generic *recv_async* connector is instantiated with the type *Command.T*, then every time a command is received by the component, the maximum size of the *Command.T* type is serialized onto the queue. To get around this behavior and allow the intelligent serialization of variable length types you MUST include an additional formal parameter in your generic definition in the component model named *Serialized_Length*, which provides a function that returns the serialized length of the packed record. The definition should look like the example shown below:

```

1 ...
2 generic:
3   description: An example showing how to define a generic type that can be
4     ↳ variable length.
5   parameters:
6     - name: T
7       description: The generic type of data passed to this components recv_async
8         ↳ connector.
9     - name: Serialized_Length
10    description: A method that returns the serialized length of an item of
11      ↳ type T. This is useful for serializing variable length packed types
12      ↳ onto the queue.
13    formal_type: "with function Serialized_Length(src : in T;
14      ↳ num_Bytes_Serialized : out Natural) return
15      ↳ Serializer_Types.Serialization_Status;"
```

When the Adamant system sees a *Serialized_Length* function provided in the generic formal parameter list, it will automatically include the autocoded logic to intelligently serialize variable length types onto the component’s internal queue. Note that all packed records defined in Adamant, whether they are variable length or not, include a *Serialized_Length* function which fits the definition shown above. Thus this generic formal definition can be used to instantiate the component with static sized or variable length packed records alike. The underlying autocoded base package will handle both cases correctly.

For an example of this pattern see the *Limiter* component found in *src/components/generics/limiter*. The component contains unit tests that exercise the component’s asynchronous connector instantiated with both a statically sized and variable length packed record.

6.18.2 Component Interrupt Handling

Adamant provides modeling support for any component that directly handles interrupts. Interrupt specific parameters such as the interrupt priority and ID are often set via the *discriminant* in Ada. Indeed, you could simply use the *discriminiant* tag in your component model to pass in this interrupt data, see Section 6.7.1 but then the Adamant modeling system would not know about the interrupt. By providing a way to specify interrupts directly within a component model we are able to more readily produce documentation and other useful autocode associated with all the interrupts in the system. So if you need to handle and interrupt in your component, using the feature presented in this section is highly recommended.

Most interrupt handling in Adamant can be achieved by using the components found in *src/components/interrupt_handlers*. Sometimes you need to handle interrupts in a particular way that these components cannot accommodate, and thus you need to define an interrupt handling component from scratch. In this case, using the Adamant interrupt handling components will serve as good examples. In fact, we will use the Adamant *interrupt_servicer* component to demonstrate the interrupt feature in this section.

Let's take a look at the interrupt servicer component model which can be found in *src/components/interrupt_handlers/interrupt_servicer/*:

interrupt_servicer.component.yaml:

```
1  ---
2  description: The Interrupt Servicer provides a task that an attached component
3    ↳ executes on when an interrupt is received. The component is attached to an
4    ↳ interrupt. When it receives the interrupt, the component's internal task is
5    ↳ released, and the component calls the Interrupt_Data_Type_Send connector
6    ↳ causing an attached component to execute on its internal task. A
7    ↳ user-defined generic data type is passed between the Interrupt Servicer and
8    ↳ the connected component. This data type will usually be filled in within a
9    ↳ user defined custom interrupt handler, provided to the Interrupt Servicer at
10   ↳ instantiation, and called by the Interrupt Servicer every time the specified
11   ↳ interrupt is received.
12
13 execution: active
14 with:
15   - Interrupt_Handlers
16 generic:
17   description: The Interrupt Servicer is parameterized by the
18     ↳ Interrupt_Data_Type. The instantiation of this type is often determined by
19     ↳ what data needs to be collected during an interrupt and how a connected
20     ↳ component intends to process that data. This type a user defined type that
21     ↳ is passed into the user's custom interrupt handler as an in-out parameter
22     ↳ and then passed to a downstream components out of the
23     ↳ Interrupt_Data_Type_Send connector. Usually, this type will be used to
24     ↳ capture data related to an interrupt in the interrupt handler. Often, a
25     ↳ timestamp, relating when the interrupt occurred, will be included in this
26     ↳ custom type. The Interrupt Servicer will automatically insert the
27     ↳ timestamp into the custom type if the second generic parameter, the
28     ↳ Set_Interrupt_Data_Time procedure, is provided.
29
30 parameters:
31   - name: Interrupt_Data_Type
32     description: The user's custom datatype that is is set in the custom
33       ↳ interrupt handler and then passed to downstream components. If you do
34       ↳ not foresee needing to collect specific data in the interrupt handler
35       ↳ for the downstream component, consider using the Tick.T type, which
36       ↳ includes a timestamp and a count.
37   - name: Set_Interrupt_Data_Time
38     formal_type: "with procedure Set_Interrupt_Data_Time (Interrupt_Data : in
39       ↳ out Interrupt_Data_Type; Time : in Sys_Time.T) is null;"
```

```

13     optional: True
14     description: This is an optional generic parameter that is used if the
15         ↪ Interrupt_Data_Type includes a timestamp which should be filled in
16         ↪ when an interrupt occurs. This parameter is a user defined procedure
17         ↪ that, given a time, copies that time to the Interrupt_Data_Type. If
18         ↪ this parameter is not provided, then time will not be set in the
19         ↪ Interrupt_Data_Type. In this case, the user should not connect the
20         ↪ Sys_Time_T_Get connector to avoid the CPU overhead of fetching a time
21         ↪ that is never used. Note, if you are using Tick.T for the
22         ↪ Interrupt_Data_Type, consider using the Handler procedure in the
23         ↪ Tick_Interrupt_Handler package to instantiate this parameter.
24
25 preamble: |
26     -- Define the custom interrupt handling package type
27     package Custom_Interrupt_Handler_Package is new Interrupt_Handlers
28         (Interrupt_Data_Type);
29
30 discriminant:
31
32 parameters:
33     - name: custom_Interrupt_Procedure
34         type: Custom_Interrupt_Handler_Package.Interrupt_Procedure_Type
35     description: A custom procedure to be called within the interrupt handler.
36         ↪ The null procedure can be used here if no specific behavior is
37         ↪ desired.
38
39 interrupts:
40     - name: interrupt
41         description: This component counts the number of times this interrupt
42             ↪ occurs.
43
44 connectors:
45     - description: The data send connection.
46         type: Interrupt_Data_Type
47         kind: send
48     - description: The system time is retrieved via this connector.
49         return_type: Sys_Time.T
50         kind: get

```

This is a complex model that demonstrates many of the advanced features of component modeling. We can see there is a generic instantiation, the details of which are presented in the previous section. There is also a discriminant defined, details found in Section 6.7.1. The new tag we can see in this model is the `interrupts` tag.

The `interrupts` tag takes a list of interrupts that the component handles. Each interrupt is provided a name and an optional description. That is it.

The purpose of this tag is twofold 1) to provide the Adamant modeling system with information about all interrupts in the system for autocoding and documentation generation purposes, and 2) to allow the component's code generator to automatically insert interrupt related parameters into the component's *discriminant*. We will demonstrate the second purpose by taking a look at the `interrupt_servicer`'s implementation specification:

component-interrupt_servicer-implementation.ads:

```

1 -----
2 -- Interrupt_Servicer Component Implementation Spec
3 -----
4
5 -- Discriminant Type Includes:
6 with Ada.Interrupts;
7 with System;
8
9 -- This is the Interrupt Servicer component. It is attached to an interrupt and
10    ↪ sends out a Tick.T every time the interrupt is triggered. This component
11    ↪ MUST be made active in order to function properly.

```

```

10 generic
11 package Component.Interrupt_Servicer.Implementation is
12
13   -- The component class instance record:
14   --
15   -- Discriminant Parameters:
16   -- custom_Interrupt_Procedure :
17   --   Custom_Interrupt_Handler_Package.Interrupt_Procedure_Type - A custom
18   --   procedure to be called within the interrupt handler. The null procedure
19   --   can be used here if no specific behavior is desired.
20   -- interrupt_Id : Ada.Interrupts.Interrupt_ID - Interrupt identifier number
21   --   for interrupt
22   -- interrupt_Priority : System Interrupt_Priority - Interrupt priority for
23   --   interrupt
24   --
25
26   type Instance (Custom_Interrupt_Procedure :
27     Custom_Interrupt_Handler_Package.Interrupt_Procedure_Type; Interrupt_Id :
28     Ada.Interrupts.Interrupt_ID; Interrupt_Priority :
29     System Interrupt_Priority) is new Interrupt_Servicer.Base_Instance with
30   private;
31
32 private
33
34   -- The component class instance record:
35   --
36   -- Discriminant Parameters:
37   -- custom_Interrupt_Procedure :
38   --   Custom_Interrupt_Handler_Package.Interrupt_Procedure_Type - A custom
39   --   procedure to be called within the interrupt handler. The null procedure
40   --   can be used here if no specific behavior is desired.
41   -- interrupt_Id : Ada.Interrupts.Interrupt_ID - Interrupt identifier number
42   --   for interrupt
43   -- interrupt_Priority : System Interrupt_Priority - Interrupt priority for
44   --   interrupt
45   --
46
47   type Instance (Custom_Interrupt_Procedure :
48     Custom_Interrupt_Handler_Package.Interrupt_Procedure_Type; Interrupt_Id :
49     Ada.Interrupts.Interrupt_ID; Interrupt_Priority :
      System Interrupt_Priority) is new Interrupt_Servicer.Base_Instance with
      record
        -- The internal task signal protected object.
        The_Signal : Custom_Interrupt_Handler_Package.Task_Signal
          -- (Interrupt_Priority, Interrupt_Id, Custom_Interrupt_Procedure);
      end record;
50
51 -----
52   -- Invoker connector primitives:
53 -----
54
55   -- This procedure is called when a Interrupt_Data_Type_Send message is
56   -- dropped due to a full queue.
57   overriding procedure Interrupt_Data_Type_Send_Dropped (Self : in out
58     Instance; Arg : in Interrupt_Data_Type) is null;
59
60 -----
61   -- Definition of cycle function for task execution:
62 -----
63
64   -- This is an active component with no queue, so the
65   -- cycle function for the component's task must be
66   -- implemented here in the implementation class as
67   -- a user defined custom function.
68   overriding procedure Cycle (Self : in out Instance);

```

```

50
51 end Component.Interrupt_Servicer.Implementation;

```

As can be seen in the component `Instance` definition, the discriminant not only consists of the discriminant parameter defined in the model, `custom_Interrupt_Procedure`, it also has two additional parameters, `interrupt_Id` and `interrupt_Priority`. The second two parameters are inserted automatically due to the `interrupt` definition in the component model. We can see that these two parameters are used to instantiate a `Task_Signal` object in the component record, whose definition can be found in the Adamant core interrupt handlers package, `src/core/interrupt_handlers`. You should consider using, or adding to, this package if you are defining a custom interrupt handling component in your system.

Instantiating a component with interrupts in the assembly model is no different than instantiating any other component with discriminant parameters defined. An example is shown below.

```

1 components:
2   - type: Interrupt_Servicer
3     description: This component handles and interrupt and passes it along to the
4       ↳ assembly.
5     priority: 1
6     stack_size: 2000
7     secondary_stack_size: 10000
8     generic_types:
9       - "Interrupt_Data_Type => Tick.T"
10      - "Set_Interrupt_Data_Time => Tick_Interrupt_Handler.Set_Tick_Time"
11      discriminant:
12        - "interrupt_Priority => System.Interrupt_Priority'Last"
13        - "interrupt_Id => Ada.Interrupts.Names.SIGUSR1"
14        - "custom_Interrupt_Procedure => Tick_Interrupt_Handler.Handler'Access"
15      ...
16      connections:
17        ...
etc.

```

As can be seen, three parameters are passed into the discriminant, two of which are the interrupt specific parameters.

6.18.3 Component Subtasks

An *active* component only has a single task associated with it. There are times when it makes sense to have two tasks contained inside a single component. Usually this is necessary when you need two threads of execution to handle a single piece of shared data or hardware, such as a socket. In many cases, the desired behavior can be achieved using two *active* components that work in concert through protected objects. However, there are times when the cleaner design is simply to have a single component with 2 or more tasks.

To accomplish this, Adamant includes a component model feature to instantiate one or more *subtasks* within the component. A subtask is the same as the component's *active* task (if it has one), except that the behavior of the subtask must always be completely specified by the developer.

A good example of defining a subtask is the `ccsds_socket_interface` component, found in `src/component/ccsds_socket_interface/`. This component uses two tasks to manage a single socket interface. The main *active* component task is responsible for waiting on the component's queue for data to send out of the socket. When data is received, it is forwarded out the socket using the *active* task. A subtask is instantiated to listen on the socket for incoming data. When incoming data is received on the

listener task, it is then forwarded out to the rest of the system through a connector. By instantiating a subtask, we are able to accomplish reading and writing on a shared socket within a single component.

Defining a subtask must be done in the component model. Below is the component model for the *ccsds_socket_interface*.

ccsds_socket_interface.component.yaml:

```

1  ---
2  description: |
3      The Socket Interface Component is an interface component which connects the
4      rest of the assembly to an outside entity (usually the ground system) via
5      a TCP/IP socket. It spawns an internal task to listen to the socket for
6      incoming data. It also provides an asynchronous receive connector which it
7      services on its task, sending any data it receives out of the socket. The
8      data send and receive connectors are CCSDS.
9  execution: active
10 init:
11     description: This initialization subprogram connects the component to a TCP
12         ↵ socket on the given address and port.
13 parameters:
14     - name: Addr
15         type: String
16         default: "\"127.0.0.1\""
17         description: The IP address or hostname that the component should connect
18             ↵ to. This could be something like 127.0.0.1 or www.google.com.
19     - name: Port
20         type: Natural
21         default: "2001"
22         description: The port that the component should connect to.
23 subtasks:
24     - name: Listener
25         description: This internal task is used to listen on the socket for incoming
26             ↵ packets.
27 connectors:
28     - description: On this connector the Socket Interface Component receives data
29         ↵ and sends it out of the socket.
30         type: Ccsds_Space_Packet.T
31         kind: recv_async
32     - description: On this connector the Socket Interface Component sends any data
33         ↵ it received from the socket.
34         type: Ccsds_Space_Packet.T
35         kind: send
36     - description: Events are sent out of this connector.
37         type: Event.T
38         kind: send
39     - description: The system time is retrieved via this connector.
40         return_type: Sys_Time.T
41         kind: get

```

As can be seen there is a new **subtasks** tag under which is listed the required subtask name and optional description. This is all the data required to define a subtask. If a component needs more than one subtask, the list can be appended to.

To see how a subtask affects the component we can look at the component implementation specification for the *ccsds_socket_interface*.

component-ccsds_socket_interface-implementation.ads:

```

1 -----
2 -- Ccsds_Socket_Interface Component Implementation Spec
3 -----
4
5 -- Standard Includes:
6 -- Invokee Connector Includes:
7 with Socket;
8 with Socket_Address;
9
10 -- The Socket Component is an interface component which connects the rest of the
--> assembly to an outside entity (usually the ground system) via a TCP/IP
--> socket. It spawns an internal task to listen to the socket for incoming
--> data. It also provides an asynchronous receive connector which it services
--> on its task, sending any data it receives out of the socket. The data send
--> and receive connectors are of a generic buffer type, Com_Packet, so that
--> data of an arbitrary format can be sent via this component.
11 --
12 package Component.Ccsds_Socket_Interface.Implementation is
13
14     type Instance is new Ccsds_Socket_Interface.Base_Instance with private;
15
16 -----
17 -- Subprogram for implementation init method:
18 -----
19 -- This initialization subprogram connects the component to a TCP socket on
--> the given address and port.
20 overriding procedure Init (Self : in out Instance; Addr : in String :=
--> "127.0.0.1"; Port : in Natural := 2_001);
21 -- Public function which closes the TCP socket.
22 not overriding procedure Final (Self : in out Instance);
23
24 private
25     -- The component class instance record:
26     type Instance is new Ccsds_Socket_Interface.Base_Instance with record
27         Sock : Socket.Instance;
28         Addr : String (1 .. 512);
29         Port : Natural;
30     end record;
31
32 -----
33 -- Invokee connector primitives:
34 -----
35 -- On this connector the Socket Interface Component receives data and sends
--> it out of the socket.
36 overriding procedure Ccsds_Space_Packet_T_Recv_Async (Self : in out Instance;
--> Arg : in Ccsds_Space_Packet.T);
37 -- This procedure is called when a Ccsds_Space_Packet_T_Recv_Async message is
--> dropped due to a full queue.
38 overriding procedure Ccsds_Space_Packet_T_Recv_Async_Dropped (Self : in out
--> Instance; Arg : in Ccsds_Space_Packet.T) is null;
39
40 -----
41 -- Invoker connector primitives:
42 -----
43 -- This procedure is called when a Ccsds_Space_Packet_T_Send message is
--> dropped due to a full queue.
44 overriding procedure Ccsds_Space_Packet_T_Send_Dropped (Self : in out
--> Instance; Arg : in Ccsds_Space_Packet.T) is null;
45 -- This procedure is called when a Event_T_Send message is dropped due to a
--> full queue.
46 overriding procedure Event_T_Send_Dropped (Self : in out Instance; Arg : in
--> Event.T) is null;

```

```

47 -----
48 -- Private helper functions:
49 -----
50
51 -- Helper function to convert GNAT socket address to Adamant Socket_Address
52 function Convert_Socket_Address (Self : in Instance) return Socket_Address.T;
53
54 -----
55 -- Definition of subtasks functions for task execution:
56 -----
57 -- This internal task is used to listen on the socket for incoming packets.
58 -- IMPORTANT: This component needs an inner task to run the
59 -- following function as if it were the "cycle" method of
60 -- a normal component. The "cycle" method of this component
61 -- acts in the normal way, blocking on the component
62 -- queue, waiting for messages to send through the socket.
63 -- The special cycle method below should be run on another
64 -- task to block on the socket, waiting for incoming messages
65 -- which it will then send out to the rest of the assembly.
66 overriding procedure Listener (Self : in out Instance);
67
68 end Component.Ccsds_Socket_Interface.Implementation;

```

We can see a special overriding procedure that is generated called `Listener`. This is the main procedure for the *listener* task, and serves the same purpose as the `Cycle` procedure for the *active* tasks, see Section 6.4. Specifically, the `Listener` procedure will be called endlessly in a loop whenever the *listener* task is given time to execute by the Ada scheduler. In the case of the `ccsds_socket_interface` component, the `Listener` function blocks while waiting on incoming data from the socket, and then forwards any data it receives out of a send connector. This can be verified by looking at the implementation body for the `ccsds_socket_interface` component, not shown here for brevity.

To instantiate a component with a subtask within an assembly, we use an additional tag `subtasks`. An example of instantiating the `ccsds_socket_interface` component is shown below.

```

1 components:
2   - type: Ccsds_Socket_Interface
3     description: This component receives commands from and sends events through
4       ↪ a TCP/IP Socket.
5     priority: 6
6     stack_size: 50000
7     secondary_stack_size: 10000
8     init_base:
9       - "queue_Size => 8192"
10    init:
11      - "addr => \"127.0.0.1\""
12      - "port => 2001"
13    subtasks:
14      - name: Listener
15        priority: 0
16        stack_size: 10000
17        secondary_stack_size: 5000
18        # Uncomment the following line if you want to disable the
19        # subtask from running in the assembly.
20        # disabled: True
21    ...
22  ...
23  etc.

```

Note that the single subtask `Listener` is instantiated with `priority`, `stack_size`, and `secondary_stack_size` parameters in the same way that any *active* component's task is normally instantiated. Sometimes you may not want the subtask to run in your assembly. This can be done by setting `disabled` to `True`. In this case `priority`, `stack_size`, and `secondary_stack_size` still need to be specified, but will not be used in the generated code.

6.18.4 Assembly-parameterized Components

One of the most powerful features of Adamant is the rich modeling language and extensive autocoding system. At the assembly level, quite a lot of information is known about the system as a whole including the commands it responds to, the packets it creates, the data products it produces, etc. Adamant supports and encourages a pattern of creating "assembly-parameterized components". These are reusable components whose behavior depends on the assembly that they are part of.

Simple examples of this concept are the Adamant components found in `src/components/monitors`. There is a `cpu_monitor` which produces a packet that contains performance information for every task in the assembly. There is a `queue_monitor` which produces a packet that contains usage information for every queue in the assembly. Finally, there is a `stack_monitor` component, which produces a packet that contains usage data for every stack used in the assembly. All of these components are *parameterized* by the assembly that they are part of in that they produce a packet whose contents are determined by modeling data only available at the assembly level.

This parameterization is usually achieved through component-specific generators that autocode assembly-derived data structures that the component can then operate on. This concept is complex and cannot be fully discussed here. The best way to understand this pattern is to look at components in Adamant that employ it. One of the best examples is the `product_packetizer` component, which will be used as an example for this section.

The `product_packetizer` can be found in `src/components/packetizers/product_packetizer/`. The component's main job is to grab data products from a central database onboard the embedded system and then packetize the data products into predefined packets that are then sent out. The layout of the different packets that the `product_packetizer` creates is defined by a specific model type for the component which is stored in a file that ends in `*.product_packets.yaml`. The YAML model defines the packet layout in terms of the data products that are included in each packet. Note that this is assembly-level information. The `product_packetizer` component can be used in a variety of assemblies, each with different components that produce different data products with different IDs. The only way to get the specific information about the data products in the particular assembly that the `product_packetizer` is being used in is to use autocoding.

To achieve this the `product_packetizer` defines its own specific generator, which can be found `src/components/packetizers/product_packetizer/gen`. If you look at the contents of that directory you will find that it mirrors the Adamant `gen/` directory exactly. Following this pattern is the proper way to define a component-specific generator. For details on building a component-specific generator or an Adamant-wide generator you can follow the instructions in Section 9.11.2. Documentation specific to the `product_packetizer` generator can be found in `src/components/packetizers/product_packetizer/gen/doc`.

The `product_packetizer` generator takes the packet definitions, makes sure the defined data products exist in the assembly, determines the IDs for those data products, and then finally outputs a record that contains all the data necessary for the component to make the specific packets. This record is autocoded based on the assembly and is then provided to the component at instantiation. In this way, we have parameterized the behavior of the component based on the assembly it is part of.

Again, this pattern can only really be understood by studying the existing components that use it. Besides the `product_packetizer` and the monitor components mentioned above, the `ccsds_router`

is another good example, found in *src/components/ccsds/ccsds_router*. The *ccsds_router* routes CCSDS packets to destination components using their component instance name, again data that is only available at the assembly level.

7 Assemblies

In Adamant, an *assembly* is a collection of components that have been connected together to form an executable deployment of software. Assemblies are lightweight, modeled constructs that can be used for a variety of purposes. Often when developing software with Adamant you will develop a multitude of assemblies. You may have an assembly that contains a configuration of components that perform an early demonstration test of your system. You may have an assembly specially tuned to test a single piece of hardware. You may have an assembly that only executes in a simulation environment on a desktop PC. Finally, you will almost certainly have a production assembly that is used to construct the target software for your project.

The component-based principles of Adamant make constructing different assemblies of components for different purposes straightforward. Because components are loosely coupled, they can easily be swapped and rearranged as necessary to create differently functioning software while reusing the code found in common core components.

This section demonstrates how to construct an assembly and use it to produce integrated software made up of a set of loosely coupled components. This section also presents diagram and documentation generation, integration with a “ground” system, integration-level testing, and more.

7.1 Creating an Assembly

An assembly is simply a collection of components and their shared connections. This section provides a set of *example components* from which we build up the assembly. The example components are shown in the next section and demonstrate many of the features that were described in Section 6. Note that these example components are the same components shown in the [Architectural Description Document](#). After constructing the assembly we create a main file that allows us to run all the components together as a compiled executable program.

7.1.1 The Example Components

The example component models are presented below along with a brief description of the component’s function and its component diagram.

Example_Time

This Example_Time component has a single *return* connector, which returns the current time upon request. Note that because of the many-to-one relationship for connectors, many different components can connect to the Time_Return connector to obtain time. The time component does all its work synchronously. The component can be configured at instantiation (via its *discriminant*) to either get the system time from a Software or Hardware source. See Section 6.7.1 for more details on the component *discriminant*.

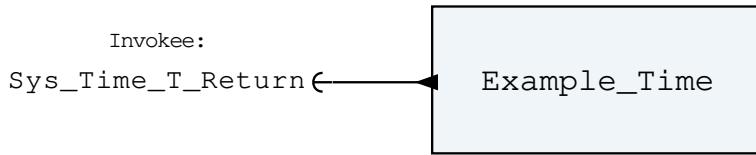
example_time.component.yaml:

```
1  ---
2  description: This component provides system time to components in the assembly.
3  execution: passive
4  preamble: |
5    type Time_Source_Type is (Software, Hardware);
6  discriminant:
7    description: The discriminant is used to specify the time source.
8  parameters:
9    - name: time_Source
10   type: Time_Source_Type
11   description: Specify whether the source for system time should be driven
12     by the hardware or software.
13 connectors:
14   - return_type: Sys_Time.T
```

```

14     kind: return
15     description: The current system time is returned on this connector.

```



Example_Logger

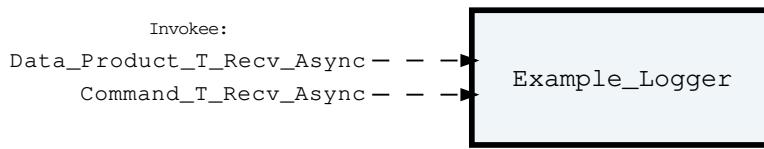
This Example_Logger component has two asynchronous *invokee* connectors. For this reason, it is created with an internal queue. The purpose of this component is to log data products to disk. Data products come in asynchronously via the Data_Product_T_Recv_Async connector, and are logged on demand. This component also includes an asynchronous command *invokee* connector for receiving configuration commands. A component like this would most likely be associated with a low priority task that runs in the background. The component can either be enabled or disabled on startup. This is configured in the component's custom Init procedure. For more information on component Init procedures see Section 6.7.5.

example_logger.component.yaml:

```

1  ---
2  description: This component logs data products to a file.
3  execution: active
4  init:
5    description: An initialization function for the logger.
6    parameters:
7      - name: is_Enabled
8        type: Boolean
9        default: "True"
10       description: Should the logger log to a file by default on startup?
11 connectors:
12   - type: Data_Product.T
13     kind: recv_async
14     description: Data products are received on this connector
15   - type: Command.T
16     kind: recv_async
17     description: Commands to enable/disable the logger are received on this
      ↳ connector.

```



Example_Parameters

The Example_Parameters component's primary task is to push the current value of parameters to components throughout the system. To accomplish this, it has an arrayed *provide* connector through which it sends parameter updates to destination components. The component also includes a connector which allows parameter values to be changed by command. For more detail on component parameters see Section 6.14.

example_parameters.component.yaml:

```
1  ---
2  description: This component stores parameters for the assembly and pushes new
   ↳ values out to parameterized components.
3  execution: passive
4  connectors:
5  - description: The arrayed parameter request connector. Parameters stages,
   ↳ updates, and fetches are sent out this connector and a status is returned.
6  type: Parameter_Update.T
7  kind: provide
8  count: 0
9  - type: Command.T
10 kind: recv_sync
11 description: Commands to upload and change parameters are received on this
   ↳ connector.
```



Example_Rate_Group

The Example_Rate_Group is useful for scheduling the activity of other components in a system. It takes a single asynchronous connector of type Tick.T in at one rate and schedules the execution of components connected to its arrayed Tick_T_Send connectors using its internal task. This component could be used to take a 1 Hz hardware driven interrupt and schedule the execution of components throughout the system that need to execute at a 1 Hz cadence.

example_rate_group.component.yaml:

```
1  ---
2  description: This component provides a task for passive components to execute on
   ↳ cyclically.
3  execution: active
4  connectors:
5  - type: Tick.T
6  kind: recv_async
7  description: A periodic tick from the hardware or software timer is received
   ↳ on this connector.
8  - type: Tick.T
9  kind: send
10 count: 5
11 description: Ticks at different frequencies are sent out of this connector.
```



Example_Command_Router

This Example_Command_Router component's job is to take commands asynchronously from an *invokee* connector, determine where that command should be routed to for execution, and then send that command out the correct *invoker* connector. Note that the Command_T_Send

connector has an array length of `<>`, specified via a `count` of 0. This denotes that the size of the array will be determined during the construction of the assembly. The component itself is made more generic by not having to hard code this connector array length into the component model.

example_command_router.component.yaml:

```

1  ---
2  description: This component routes commands to components in the assembly.
3  execution: active
4  connectors:
5    - type: Command.T
6      kind: recv_async
7      description: Commands to route are received on this connector.
8    - type: Command.T
9      kind: send
10     count: 0
11     description: Commands are routed out of this arrayed connector to their
→   destination.

```



Example_Data_Collector

The `Example_Data_Collector` component's execution is scheduled via a `Tick_T_Recv_Sync` connector. When called, the component grabs data from onboard sensors and sends the values out as data products. To time stamp the data products, this component has a connection to get time.

example_data_collector.component.yaml:

```

1  ---
2  description: This component collects sensor measurements and outputs them as
→   data products. It executes on a periodic tick.
3  execution: passive
4  connectors:
5    - type: Tick.T
6      kind: recv_sync
7      name: Tick_T_Recv_Sync
8      description: This connector provides the schedule tick for the component
9    - type: Data_Product.T
10     kind: send
11     description: Data products are sent out of this connector.
12    - return_type: Sys_Time.T
13     kind: get
14     description: Time is fetched on this connector.

```

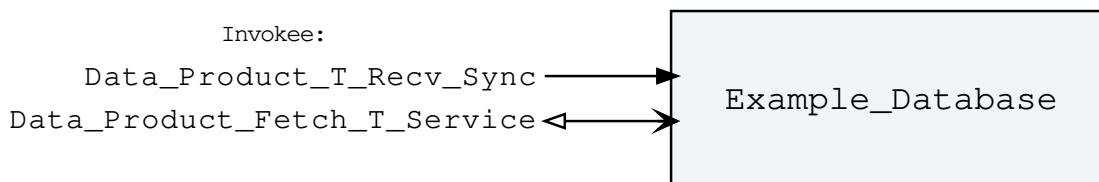


Example_Database

The Example_Database is a passive component that acts as a central store for the data products produced by components throughout the assembly. The database stores a single, most-up-to-date value for each data product in the system. It provides connectors to set and get data product values.

example database.component.yaml:

```
1 ---  
2 description: This component stores system state variables and collected sensor  
3   ↳ measurements as data products within a database.  
4 execution: passive  
5 connectors:  
6   - type: Data_Product.T  
7     kind: recv_sync  
8     description: Store a data product item in the database.  
9   - type: Data_Product_Fetch.T  
10    return_type: Data_Product_Return.T  
11    kind: service  
12    description: Fetch a data product item from the database.
```



Example Science

The Example_Science component is more complicated than the rest, since this component performs the function of the example mission. In this case, the component's execution is scheduled via a Tick_T_Recv_Sync connector. The component can be configured asynchronously via command. The component produces processed science data products based on data dependencies it fetches from the database. It receives updated parameters through its Parameter_Update_T_Modify connector. To time stamp the data products, this component also has a connection to get time.

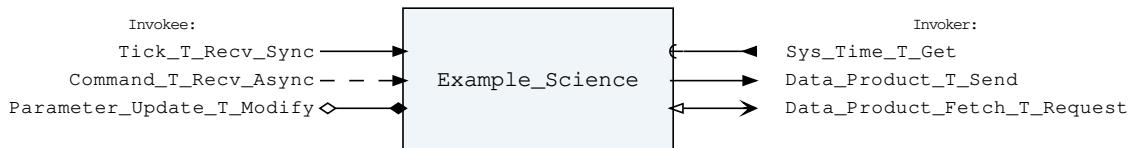
example science.component.yaml:

```
1 ---  
2 description: This is the science component that performs the mission behavior.  
3 execution: passive  
4 connectors:  
5   - type: Tick.T  
6     kind: recv_sync  
7     name: Tick_T_Recv_Sync  
8     description: This connector provides the schedule tick for the component  
9   - type: Command.T  
10    kind: recv_async  
11    description: Commands are received on this connector.  
12   - return_type: Sys_Time.T  
13    kind: get  
14    description: Time is fetched on this connector.  
15   - type: Parameter_Update.T  
16    kind: modify  
17    description: The parameter update connector.  
18   - type: Data_Product.T  
19    kind: send  
20    description: Data products are sent out of this connector.  
21   - type: Data_Product_Fetch.T
```

```

22     return_type: Data_Product_Return.T
23     kind: request
24     description: Fetch a data product item (data dependency) from the database.

```



In the next section we will show how all these components can be connected together into an executable assembly.

7.1.2 Modeling an Assembly

In this section we will create an assembly model to connect all the example components together. First we need to create a new directory to hold our assembly model.

```

> mkdir science_assembly      # make assembly directory
> cd science_assembly
> touch .all_path            # add directory to build path

```

To define an assembly in this directory, a developer must create a YAML model file in the form *name.assembly.yaml* where *name* is the desired name of the *assembly*. Below is the component definition for the *science_assembly*. This definition is stored in a file called *science_assembly.yaml* within this new directory.

science_assembly.yaml:

```

1  ---
2  # Optional - A description for the assembly
3  description: This is the example assembly.
4  # Required - A list of components included in the assembly
5  components:
6      # Required - The component type, this is the name of the
7      # component model.
8      - type: example_rate_group
9          # Optional - The component name. If not specified the
10         # component will be the component type name followed by
11         # "_Instance"
12         name: rate_group_instance
13         # Optional - This is only required when the component
14         # model specifies the execution as "either". If specified,
15         # this must not conflict with the component model's
16         # definition of "execution".
17         execution: active
18         # Required (only for "active" components) - specify the
19         # Ada scheduling priority for this component's task.
20         priority: 3
21         # Required (only for "active" components) - specify the
22         # task primary stack size in bytes.
23         stack_size: 3000
24         # Required (only for "active" components) - specify the
25         # task secondary stack size in bytes.
26         secondary_stack_size: 1000
27         # Required (only for components that have an "init_base"
28         # procedure) - specify the init_base parameter values as
29         # key value pairs.

```

```

30     init_base:
31         - "queue_Size => 1000" # in bytes
32     - type: example_command_router
33         name: command_router_instance
34         execution: active
35         priority: 2
36         stack_size: 3000
37         secondary_stack_size: 1000
38         # Required (only for components that have an "init_base"
39         # procedure) - specify the init_base parameter values as
40         # key value pairs.
41     init_base:
42         - "queue_Size => 1000" # in bytes
43         - "command_T_Send_Count => 3"
44     - type: example_science
45         name: science_instance
46     init_base:
47         - "queue_Size => 1000" # in bytes
48         # Optional - declare the base id for an ided entity
49         # suite. If not declared, or a parameter is omitted,
50         # then Adamant will automatically figure out the base
51         # id to use, selecting the lowest remaining ID
52         # available.
53     set_id_bases:
54         - "data_Product_Id_Base => 100"
55         # Required (only for components that have a data_dependencies
56         # model) - specify the data products that map to the data
57         # dependencies
58     map_data_dependencies:
59         - data_dependency: sensor_1_Reading
60             data_product: data_Collector_Instance.Sensor_1_Data
61             stale_limit_us: 1000000
62         - data_dependency: sensor_2_Reading
63             data_product: data_Collector_Instance.Sensor_2_Data
64             stale_limit_us: 1000000
65     - type: example_time
66         name: time_instance
67         discriminant:
68             - "time_Source => Example_Time.Software"
69     - type: example_parameters
70         name: parameters_instance
71     init_base:
72         - "parameter_Update_T_Provide_Count => 1"
73     - type: example_logger
74         name: logger_instance
75         priority: 1
76         stack_size: 3000
77         secondary_stack_size: 1000
78     init_base:
79         - "queue_Size => 1000" # in bytes
80         # Required (only for components that have an "init"
81         # procedure) - specify the init parameter values as
82         # key value pairs.
83     init:
84         - "is_Enabled => True"
85     - type: example_data_collector
86         name: data_collector_instance
87     - type: example_database
88         name: database_instance
89 connections:
90     # Required - The invoker component the connection is called

```

```

91      # from. This must be the component instance name, not the
92      # component type name.
93      - from_component: Rate_Group_Instance
94          # Required - The invoker connector the connection originates
95          # from.
96          from_connector: Tick_T_Send
97              # Required - The invokee component the connection connects to.
98              # This must be the component instance name, not the component
99              # type name.
100         to_component: Data_Collector_Instance
101             # Required (only for arrayed connectors) - The array index the
102             # connection connects from. This is a number between 1 and the
103             # size of the arrayed connector.
104         from_index: 1
105             # Required - The invokee connector the connection terminates
106             # at.
107         to_connector: Tick_T_Recv_Sync
108             # Optional - a description explaining this connection
109             description: This is the first connection in the model
110
111     # Science connections:
112     - from_component: Rate_Group_Instance
113         from_connector: Tick_T_Send
114         from_index: 2
115         to_component: Science_Instance
116         to_connector: Tick_T_Recv_Sync
117         description: This is the first connection in the model
118     - from_component: Command_Router_Instance
119         from_connector: Command_T_Send
120         from_index: 1
121         to_component: Science_Instance
122         to_connector: Command_T_Recv_Async
123     - from_component: Science_Instance
124         from_connector: Sys_Time_T_Get
125         to_component: Time_Instance
126         to_connector: Sys_Time_T_Return
127     - from_component: Parameters_Instance
128         from_connector: Parameter_Update_T_Provide
129         to_component: Science_Instance
130         to_connector: Parameter_Update_T_Modify
131     - from_component: Science_Instance
132         from_connector: Data_Product_T_Send
133         to_component: Logger_Instance
134         to_connector: Data_Product_T_Recv_Async
135     - from_component: Science_Instance
136         from_connector: Data_Product_Fetch_T_Request
137         to_component: Database_Instance
138         to_connector: Data_Product_Fetch_T_Service
139     - from_component: Command_Router_Instance
140         from_connector: Command_T_Send
141         from_index: 2
142         to_component: Logger_Instance
143         to_connector: Command_T_Recv_Async
144     - from_component: Command_Router_Instance
145         from_connector: Command_T_Send
146         from_index: 3
147         to_component: Parameters_Instance
148         to_connector: Command_T_Recv_Sync
149
150     # Data collector connections:
151     - from_component: Data_Collector_Instance
152         from_connector: Sys_Time_T_Get
153         to_component: Time_Instance

```

```

152     to_connector: Sys_Time_T_Return
153 - from_component: Data_Collector_Instance
154     from_connector: Data_Product_T_Send
155     to_component: Database_Instance
156     to_connector: Data_Product_T_Recv_Sync
157 # This pattern is used to turn off the Adamant warning for an unconnected
158 # connector:
159 - from_component: ignore
160     from_connector: ignore
161     to_component: command_Router_Instance
162     to_connector: Command_T_Recv_Async

```

An assembly model consists of two primary sections 1) a list of the components to include in the assembly and 2) the connections between those component. This YAML model file above contains comments explaining whether or not each field is optional or required.

First let's take a look at the components included in this assembly under the `components` tag. Each component is defined with its component `type` and, optionally, a component instance name. A component name does not need to be provided. If not provided, Adamant will generate a component name by appending `_Instance` to the component type.

The first component defined, `rate_Group_Instance`, is an active component. For active components you must provide the parameters `priority`, the execution priority of the component (higher numbers execute with higher priority), `stack_size`, the primary stack size in bytes, and `secondary_stack_size`, the secondary stack size in bytes. For more information on sizing the different stacks in Ada see [this link](#).

The second component defined is the `command_Router_Instance`. This component contains an `init_base` tag which is used to initialize variables in the component base package. This is required because the component has an arrayed connector and an internal queue which must be allocated memory on the heap during initialization. For more information about the `Init_Base` procedure within components see Section 6.7.2. All parameters for any initialization tag within an assembly model must listed as id-value pairs of the form “`parameter_Name => Value`”, as can be seen in the example above. You may NOT just list the values. This is to maintain readability and also allow parameters to be listed in any order.

The `time_Instance` component has a special tag `discriminant` which provides the discriminant initialization parameters for the component. These will be passed to the component during instantiation in the autocode. For more information about the `discriminant` within components, see Section 6.7.1. Similar to the `init_base` tag, all discriminant parameters must be listed as id-value pairs of the form “`parameter_Name => Value`”.

The `logger_Instance` component has a special tag `init` which provides the implementation package initialization parameters for the component. These will be passed to the component during a call to its `Init` procedure. For more information about the `Init` procedure within components, see Section 6.7.5. Similar to the `init_base` tag, all discriminant parameters must be listed as id-value pairs of the form “`parameter_Name => Value`”.

The `science_Instance` component has a special tag `set_id_bases` which provides a base id for the component's data product suite. The component also specifies `set_data_dependencies`, which maps its two data dependencies to two data products of the same type in the assembly. Note that the component also has command, parameters, and events, however we have chosen to omit specifying those base ids. In this case, Adamant will select the IDs for us. For more details on the component `Set_Id_Bases` and `Set_Data_Dependency_Ids` function, see Sections 6.7.3 and 6.7.4. The automatic selection of IDs by Adamant is discussed further in Section 7.3.

Below the component declarations is the list of connections. These simply list the component

instance name and connector name the connection originates at and the component instance name and connector name that the connection terminates at. Connections must always be listed with the `from` component as the *invoker* and the `to` component as the *invokee*. Note that if the connector is an arrayed connector, then an index must be provided with the `from_Index` tag.

There are a lot of details to remember when defining a component model, however Adamant provides a lot of error handling to make it easier. Whenever you try to autocode something that uses the assembly model as source, see the following sections, many checks are run on the model. For example, if you forget to define a `init` tag for a component that has an `Init` method, forget a parameter in the `init_base` declaration, misname a component or connection, forget to declare the `stack_size` for an active component, connect two connectors that have conflicting types, etc. then the Adamant build system will throw an informative error alerting you to what needs to be corrected. No output autocode will be produced from the assembly model unless it passes these checks first.

In addition, Adamant will always produce a list of warning for any unconnected connectors. This can greatly aid in the connection writing process, helping you track which connections have been made, and which still need to be made. The warnings will look something like:

```
Warning science_assembly.assembly.yaml: component 'rate_Group_Instance' has
    unattached connector 'Tick_T_Recv_Sync'.
Warning science_assembly.assembly.yaml: component 'rate_Group_Instance' has
    unattached connector 'Tick_T_Send[2]'.
Warning science_assembly.assembly.yaml: component 'rate_Group_Instance' has
    unattached connector 'Tick_T_Send[3]'.
Warning science_assembly.assembly.yaml: component 'rate_Group_Instance' has
    unattached connector 'Tick_T_Send[4]'.
Warning science_assembly.assembly.yaml: component 'rate_Group_Instance' has
    unattached connector 'Tick_T_Send[5]'.
Warning science_assembly.assembly.yaml: component 'command_Router_Instance' has
    unattached connector 'Command_T_Recv_Async'.
```

Sometimes it is necessary to purposely leave a connector disconnected, but you don't want to keep seeing the Adamant warning about the connector being unconnected. To disable this warning you simply need to connect the connector to a component and connector with both fields set to `ignore`, as is shown in the example for the `command_Router_Instance.Command_T_Recv_Async` connector. This turns off the warning.

Now that the assembly model is created, many new build rules are available (some output removed for brevity):

```
> redo what # show "redo" commands that can be run in this directory
redo what
redo all
redo clean
redo clean_all
redo templates
redo publish
redo targets
redo prove
redo analyze
redo style
redo pretty
redo test_all
redo analyze_all
redo coverage_all
```

```

redo build/dot/science_assembly.dot
redo build/eps/science_assembly.eps
redo build/html/science_assembly_commands.html
redo build/html/science_assembly_components.html
redo build/html/science_assembly_connections.html
redo build/html/science_assembly_cpu_monitor_packet_type.html
redo build/html/science_assembly_data_products.html
redo build/html/science_assembly_events.html
redo build/html/science_assembly_interrupts.html
redo build/html/science_assembly_packets.html
redo build/html/science_assembly_priorities.html
redo build/html/science_assembly_queue_monitor_packet_type.html
redo build/html/science_assembly_queues.html
redo build/html/science_assembly_stack_monitor_packet_type.html
redo build/hydra/Config/science_assembly.xml
redo build/hydra/Scripts/science_assembly_packet_pages.prc
redo build/obj/Linux/science_assembly.o
redo build/obj/Linux/science_assembly_data_products.o
redo build/obj/Linux/science_assembly_events.o
redo build/obj/Linux/science_assembly_packets.o
redo build/obj/Linux/science_assembly_parameters.o
redo build/png/science_assembly.png
redo build/src/science_assembly.adb
redo build/src/science_assembly.ads
redo build/src/science_assembly_commands.ads
redo build/src/science_assembly_data_products.ads
redo build/src/science_assembly_events.ads
redo build/src/science_assembly_packets.ads
redo build/src/science_assembly_parameters.ads
redo build/svg/science_assembly.svg

```

In this section we will explore the primary Ada package that can be autogenerated `Science_Assembly`. To build this package we run:

```
> redo build/obj/Linux/science_assembly.o
```

which autocodes the `build/src/science_assembly.ads` and `build/src/science_assembly.adb` files. The purpose of these files is to create, initialize, and connect the components together so that they can run in unison. Note, these autocoded files never need to be modified by hand, but it is good to know what they contain. Let's take a look at the specification first.

science_assembly.ads:

```

1 -----
2 -- Science_Assembly Assembly Spec
3 --
4 -- Generated from science_assembly.assembly.yaml on 2025-07-15 21:59.
5 -----
6 
7 -- This is the example assembly.
8 package Science_Assembly is
9 
10  -----
11  -- Public Subprograms:
12  -----
13  -- The subprograms below are listed in the recommended order that they
14  -- should be called at program start up.

```

```

15   --
16   -- Initialize the base classes of each component. This procedure
17   -- allocates necessary heap memory for things like queues and arrayed
18   -- connectors.
19   procedure Init_Base;
20   -- Set the appropriate command, event, parameter, packet and data product id
21   -- bases in all the
22   -- components. This also resolves the data dependency ids for components that
23   -- have data
24   -- dependencies.
25   procedure Set_Id_Bases;
26   -- Attach the connectors for all the components in the assembly
27   procedure Connect_Components;
28   -- Call the implementation initialization procedure of each component
29   procedure Init_Components;
30   -- Procedure which releases all the active components in the system, allowing
31   -- their tasks to execute:
32   procedure Start_Components;
33   -- Procedure which ends all of the active components in the system, exiting
34   -- their
35   -- task loops. This procedure should not be called on an embedded system.
36   procedure Stop_Components;
37   -- Call the component set up procedures. This is generally called after all
38   -- component initialization has been completed and tasks have been started.
procedure Set_Up_Components;

end Science_Assembly;

```

First we can see that the package includes all the component packages via with statements.

In the package declaration there are a series of public subprograms. These subprograms represent the initialization phases of an Adamant assembly. They will be called in order in our main file in Section 7.1.4.

The phases of initialization are described in Section 6.7 from the context of a single component. The same phases are listed below, with some assembly specific phases added, and are described from the perspective of the assembly as a whole:

1. **Init _ Base** - This procedure calls `Init_Base` for every component that contains an `Init_Base` procedure. This allocates all heap space for arrayed connectors and internal queues. See Section 6.7.2 for more details.
2. **Set _ Id _ Bases** - This procedure sets the base ids for component IDed entities such as commands, parameters, events, etc. For more details on defining a component with a `Set_Id_Bases` procedure see Section 6.7.3. Adamant automatic ID assignment is discussed further in Section 7.3. Component data dependency mapping to data data product IDs is also done via the call to this function, discussed further in Section 6.7.4.
3. **Connect _ Components** - This procedure connects all the component connectors together that are defined in the assembly model.
4. **Init _ Components** - This procedure calls `Init` for every component that contains an `Init` procedure. This initializes the component implementation package for all components that require it. See Section 6.7.5 for more details.
5. **Start _ Components** - This procedure starts the internal tasks for any active component.
6. **Stop _ Components** - This procedure stops the internal tasks for any active component. This procedure is usually not used, since embedded systems rarely stop their tasks once started. In fact, for Ravenscar based systems, tasks are not allowed to terminate, so this procedure is usually only called on Linux development systems where you want a clean exit to a simulation

of the system.

7. **Set_Up_Components** - This procedure calls the `Set_Up` procedure for every component in the assembly. For most components, the implementation of this procedure will be `null`. However, it is provided to allow components to call connectors during initialization if necessary. See Section 6.7.6 for more details.

Following the initialization procedures are the actual component instantiations themselves. Notice the `Time_Instance` instantiation which passes a parameter to the discriminant.

Below the components are the task declarations for any active components in the system. The declaration consists of creating a `Task_Types.Task_Info` record that allows Adamant to track some basic task information, an `Ada.Synchronous_Task_Control.Suspension_Object` used to make sure the task threads are started at the same time, and a `Component.Active_Task` which is the task itself. Notice how the stack sizes and task priority are passed into the tasks as parameters.

Finally, there are a few autocoded lists of the tasks, interrupts, queued components, etc. found in the assembly. These lists are used by various monitoring components provided with Adamant in `src/components/monitors`.

Now let's take a look at the implementation.

`science_assembly.adb`:

```
1  -----
2  -- Science_Assembly Assembly Body
3  --
4  -- Generated from science_assembly.assembly.yaml on 2025-07-15 21:59.
5  -----
6
7  -- Includes:
8  with Ada.Real_Time;
9  with Science_Assembly_Components;
10 with Ada.Synchronous_Task_Control;
11
12 package body Science_Assembly is
13
14    use Science_Assembly_Components;
15
16    procedure Init_Base is
17    begin
18        -----
19        -- Heap Initialization:
20        -----
21        Rate_Group_Instance.Init_Base (Queue_Size => 1000);
22        Command_Router_Instance.Init_Base (Queue_Size => 1000,
23            ↳ Command_T_Send_Count => 3);
24        Science_Instance.Init_Base (Queue_Size => 1000);
25        Parameters_Instance.Init_Base (Parameter_Update_T_Provide_Count => 1);
26        Logger_Instance.Init_Base (Queue_Size => 1000);
27    end Init_Base;
28
29    procedure Set_Id_Bases is
30    begin
31        -----
32        -- ID Base Initialization:
33        -----
34        Rate_Group_Instance.Set_Id_Bases (Event_Id_Base => 1);
35        Command_Router_Instance.Set_Id_Bases (Command_Id_Base => 1, Event_Id_Base
36            => 3);
```

```

35     Science_Instance.Set_Id_Bases (Data_Product_Id_Base => 100, Event_Id_Base
36         => 6, Parameter_Id_Base => 1, Command_Id_Base => 2);
37     Data_Collector_Instance.Set_Id_Bases (Data_Product_Id_Base => 1);
38
39     -----
40     -- Data dependency ID Mapping:
41     -----
42     Science_Instance.Map_Data_Dependencies (Sensor_1_Reading_Id => 1,
43         => Sensor_1_Reading_Stale_Limit => Ada.Real_Time.Microseconds (1000000),
44         => Sensor_2_Reading_Id => 2, Sensor_2_Reading_Stale_Limit =>
45             => Ada.Real_Time.Microseconds (1000000));
46 end Set_Id_Bases;
47
48 procedure Connect_Components is
49 begin
50     -----
51     -- Component Connections:
52     -----
53     -- This is the first connection in the model
54     Rate_Group_Instance.Attach_Tick_T_Send (From_Index => 1, To_Component =>
55         => Data_Collector_Instance'Access, Hook =>
56             => Data_Collector_Instance.Tick_T_Recv_Sync_Access);
57     -- This is the first connection in the model
58     Rate_Group_Instance.Attach_Tick_T_Send (From_Index => 2, To_Component =>
59         => Science_Instance'Access, Hook =>
60             => Science_Instance.Tick_T_Recv_Sync_Access);
61     Command_Router_Instance.Attach_Command_T_Send (From_Index => 1,
62         => To_Component => Science_Instance'Access, Hook =>
63             => Science_Instance.Command_T_Recv_Async_Access);
64     Science_Instance.Attach_Sys_Time_T_Get (To_Component =>
65         => Time_Instance'Access, Hook => Time_Instance.Sys_Time_T_Return_Access);
66     Parameters_Instance.Attach_Parameter_Update_T_Provide (From_Index => 1,
67         => To_Component => Science_Instance'Access, Hook =>
68             => Science_Instance.Parameter_Update_T_Modify_Access);
69     Science_Instance.Attach_Data_Product_T_Send (To_Component =>
70         => Logger_Instance'Access, Hook =>
71             => Logger_Instance.Data_Product_T_Recv_Async_Access);
72     Science_Instance.Attach_Data_Product_Fetch_T_Request (To_Component =>
73         => Database_Instance'Access, Hook =>
74             => Database_Instance.Data_Product_Fetch_T_Service_Access);
75     Command_Router_Instance.Attach_Command_T_Send (From_Index => 2,
76         => To_Component => Logger_Instance'Access, Hook =>
77             => Logger_Instance.Command_T_Recv_Async_Access);
78     Command_Router_Instance.Attach_Command_T_Send (From_Index => 3,
79         => To_Component => Parameters_Instance'Access, Hook =>
80             => Parameters_Instance.Command_T_Recv_Sync_Access);
81     Data_Collector_Instance.Attach_Sys_Time_T_Get (To_Component =>
82         => Time_Instance'Access, Hook => Time_Instance.Sys_Time_T_Return_Access);
83     Data_Collector_Instance.Attach_Data_Product_T_Send (To_Component =>
84         => Database_Instance'Access, Hook =>
85             => Database_Instance.Data_Product_T_Recv_Sync_Access);
86 end Connect_Components;
87
88 procedure Init_Components is
89 begin
90     -----
91     -- Component Initialization:
92     -----
93     Logger_Instance.Init (Is_Enabled => True);
94 end Init_Components;
95
96

```

```

72  procedure Start_Components is
73  begin
74      -----
75      -- Component Task Start:
76      -----
77      Ada.Synchronous_Task_Control.Set_True
78          ↳ (Rate_Group_Instance_Active_Task_Signal);
79      Ada.Synchronous_Task_Control.Set_True
80          ↳ (Command_Router_Instance_Active_Task_Signal);
81      Ada.Synchronous_Task_Control.Set_True
82          ↳ (Logger_Instance_Active_Task_Signal);
83  end Start_Components;
84
85  procedure Stop_Components is
86  begin
87      -----
88      -- Component Task Stop:
89      -----
90      Ada.Synchronous_Task_Control.Set_True
91          ↳ (Rate_Group_Instance_Active_Task_Signal);
92      Rate_Group_Instance.Stop_Task;
93      Ada.Synchronous_Task_Control.Set_True
94          ↳ (Command_Router_Instance_Active_Task_Signal);
95      Command_Router_Instance.Stop_Task;
96      Ada.Synchronous_Task_Control.Set_True
97          ↳ (Logger_Instance_Active_Task_Signal);
98      Logger_Instance.Stop_Task;
99  end Stop_Components;
100
101  -- Call the component set up procedures. This is generally called after all
102  -- component initialization has been completed and tasks have been started.
103  procedure Set_Up_Components is
104  begin
105      -----
106      -- Component Set Up:
107      -----
108      Rate_Group_Instance.Set_Up;
109      Command_Router_Instance.Set_Up;
110      Science_Instance.Set_Up;
111      Time_Instance.Set_Up;
112      Parameters_Instance.Set_Up;
113      Logger_Instance.Set_Up;
114      Data_Collector_Instance.Set_Up;
115      Database_Instance.Set_Up;
116  end Set_Up_Components;
117
118  end Science_Assembly;

```

In the `Init_Base` procedure we can see that any component that has an internal queue or arrayed connector is initialized. In the `Set_Id_Bases` procedure we can see that IDs are assigned to all component IDed entities within the assembly. Some of these IDs were specified directly in the model, others were calculated by Adamant. At the bottom of the function the IDs for the science component's two data dependencies are also assigned. For more information on ID assignment see Section 7.3. In the `Connect_Components` procedure we can see that all the connections are made from invoker to invokee. Note that arrayed connectors contain a numerical index which specifies which index in the array needs to be connected where. In the `Init_Components` procedure we can see that the `logger_Instance`'s special `Init` method is called with the parameters defined in the model. The `Start_Components` and `Stop_Components` procedures manipulate the active tasks' `Ada.Synchronous_Task_Control.Suspension_Objects` to make sure all tasks are started or stopped at the same time. Finally, the `Set_Up_Components` procedure simply calls `Set_Up` for

every component instance defined in the model.

In section 7.1.4 we will see how we can string calls to these procedures together to initialize and run an assembly.

7.1.3 Creating an Assembly Diagram

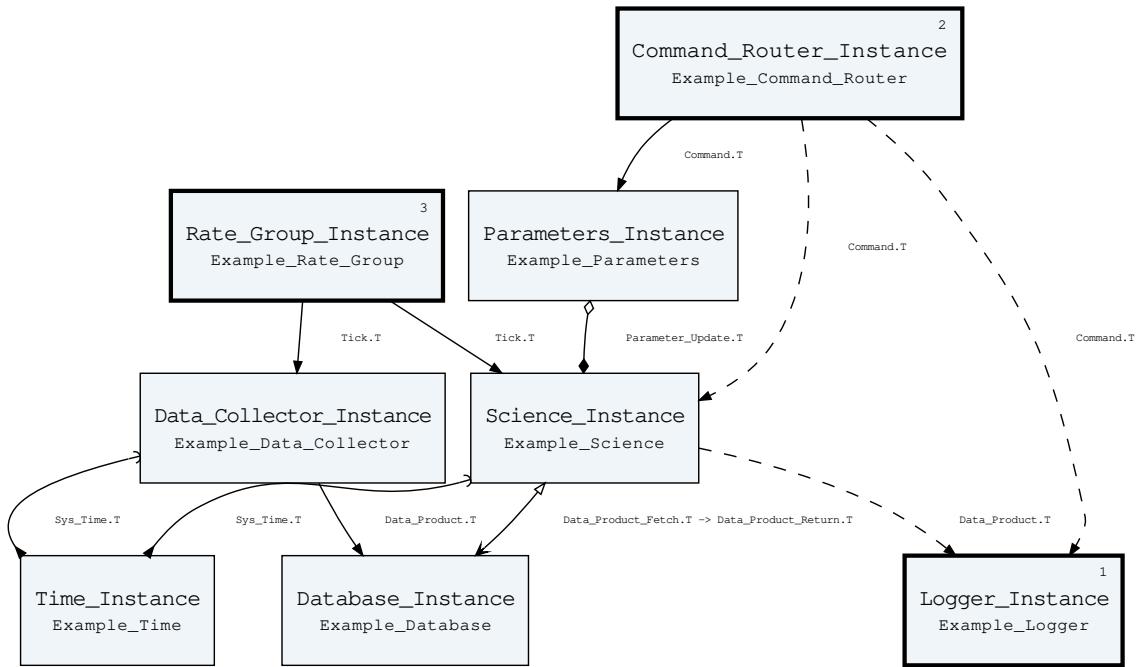
Before we compile and run the assembly it is usually good to view it in diagram form to make sure things look right. An assembly diagram can be generated from the *science_assembly* by running the following from the assembly directory.

```
> redo build/svg/science_assembly.svg
```

which produces an SVG diagram of the component that can be viewed in any web browser. An *eps* version, which is easy to import into PDF documentation can also be constructed:

```
> redo build/eps/science_assembly.eps
```

and the output is shown below:



As can be seen, all the components in the system attach to the *science_Instance* allowing the mission to execute.

This diagram is fairly readable since there are so few components in the assembly. Preferably, we could rearrange the components to make things more visually pleasing. Also note, that as more components are added, this assembly diagram will quickly become overwhelming to look at. These limitations are overcome in Adamant through assembly views which are presented in Section 7.4.

7.1.4 Running an Assembly

In this section we will create a main program that can run the assembly modeled in the last sections. First we need to create a directory to hold our main file. Usually we create this directory as a subdirectory of our assembly directory:

```
> mkdir main # make main/ directory under science_assembly/
> cd main
```

Note that we did NOT add this directory to the *build path* by adding a path file. Main files are never included by any other packages, and there may be more than a single main file in a repository, so it is best to not put this directory in the *build path*.

In this directory we create a main file called *main.adb* with the following contents:

main.adb:

```

1  with Ada.Real_Time; use Ada.Real_Time;
2  with Science_Assembly;
3  with Last_Chance_Handler;
4  with System;
5  pragma Unreferenced (Last_Chance_Handler);
6
7  procedure Main is
8      -- Set the priority of this main procedure to the highest so that it
9      -- takes precedence over any task during initialization.
10     pragma Priority (System.Priority'Last);
11     -- Define some variables helpful for sleeping.
12     Wait_Time : constant Ada.Real_Time.Time_Span := Ada.Real_Time.Microseconds
13         => (1_000_000);
14     Next_Time : Ada.Real_Time.Time;
15 begin
16     -- Set up the assembly by calling the initialization
17     -- phases in the correct order:
18     Science_Assembly.Init_Base;
19     Science_Assembly.Set_Id_Bases;
20     Science_Assembly.Connect_Components;
21     Science_Assembly.Init_Components;
22
23     -- Wait for all the tasks to have reached their "wait" on the
24     -- suspecnsion object.
25 declare
26     Start_Time : constant Ada.Real_Time.Time := Ada.Real_Time.Clock +
27         => Wait_Time;
28 begin
29     delay until Start_Time;
30 end;
31
32     -- All tasks are waiting, now start them all simultaneously.
33     Science_Assembly.Start_Components;
34
35     -- Give time for all the tasks to be up and running.
36 declare
37     Start_Time : constant Ada.Real_Time.Time := Ada.Real_Time.Clock +
38         => Wait_Time;
39 begin
40     delay until Start_Time;
41 end;
42
43     -- Now run the set up procedure.
44     Science_Assembly.Set_Up_Components;
45
46     -- Loop indefinitely, doing nothing important...
47     -- All the real work is done by the component tasks.
48     Next_Time := Ada.Real_Time.Clock + Wait_Time;
49 loop
50     -- Wait for time:
51
```

```

48     delay until Next_Time;
49     Next_Time := @ + Wait_Time;
50   end loop;
51 end Main;

```

First, we include the `Science_Assembly` autogenerated package via a `with` statement. There is also a `with` statement that includes a `Last_Chance_Handler` package. This is an Ada construct for dealing with unhandled exceptions in a Ravenscar system. It is discussed further in Section 9.7.

The main procedure begins by calling the `Science_Assembly` initialization procedures in the correct order. Before starting the component tasks, the program sleeps for a second to ensure that all component tasks have reached their suspension object and are blocked. This makes sure that all tasks are started simultaneously with the call to `start_Components`. Next, the main thread will sleep for one more second to ensure that all tasks are running before calling `Set_Up`. Lastly, the main program loops and sleeps indefinitely. After initialization, all the execution in an Adamant system is provided by the component tasks.

Now let's compile and run this file. The presence of a `main.adb` file prompts Adamant to create additional build rules (some output removed for brevity):

```

> redo what # show "redo" commands that can be run in this directory
redo what
redo all
redo clean
redo clean_all
redo templates
redo publish
redo targets
redo prove
redo analyze
redo style
redo pretty
redo test_all
redo analyze_all
redo coverage_all
redo build/bin/Linux/main.elf
redo build/obj/Linux/main.o
redo run

```

We can see that we can now compile the main object file, `build/obj/Linux/main.o`, and executable, `build/bin/Linux/main.elf`. We can also run the special command `redo run` which will build the executable and then run it.

```
> redo run
```

The `science_assembly` presented here is for demonstration purposes, does not contain any actual code, and so cannot be compiled. However, the reader is encouraged to check out the [example repository](#) which contains a compilable and runnable assembly.

7.2 Subassemblies

With complex systems consisting of many components, the resulting assembly model can become quite large. Adamant provides a way to break assemblies up into smaller constituents called *subassemblies*. Subassemblies provide a way to modularize different parts of an assembly and then reuse those parts in many different, larger, assemblies.

A subassembly is an assembly. To create a subassembly, follow all of the instructions presented in Section 7.1.2. An assembly becomes a subassembly when it is included in a larger assembly. To include a subassembly inside of another assembly use the following pattern in your assembly model:

large_assembly.assembly.yaml:

```
1  ---
2  # Include the following subassemblies inside this
3  # larger assembly
4  subassemblies:
5  - smaller_assembly_1
6  - smaller_assembly_2
7  - smaller_assembly_3
8
9  # Below is the rest of the assembly model.
10 ...
11 components:
12 ...
13 connections:
14 ...
15 etc.
```

Where the model files *smaller_assembly_1.assembly.yaml*, *smaller_assembly_2.assembly.yaml*, and *smaller_assembly_3.assembly.yaml* exist somewhere in the Adamant *build path*. Any number of subassemblies may be listed.

Note that each subassembly must also be able to act as a standalone assembly. Specifically, any connections defined in an assembly must be between components defined in that assembly or one of that assembly's subassemblies (or subassembly's subassemblies, etc.).

Also note that subassemblies are purely a modeling concept. A subassembly is not reflected in any of the output products including the autocode. In the autocode, the assembly will still appear as a flat collection of components and connections. The subassembly feature is a modeling construct to enable the modularization and reuse of assembly model definitions, and it is not an architectural concept of Adamant.

7.3 Assembly IDed Entities

The following sections discusses how IDed entities are assigned IDs within Adamant. IDed entities such as commands, parameters, events, etc. can be modeled inside of a component as described in Section 6.8. To aid in the following discussion, some IDed entity models have been included with the example components presented in Section 7.1.1. See the next section.

7.3.1 The Example Component IDed Entity Models

This section shows the IDed entity models for the example components presented in Section 7.1.1. If you are unfamiliar with these model types see Section 6.8.

Example_Rate_Group IDed Entity Models

The Example_Rate_Group component contains IDed entity models for events.

example_rate_group.events.yaml:

```

1  ---
2  events:
3    - name: Cycle_Slip
4      description: Execution ran long on this cycle.
5      param_type: Cycle_Slip_Param.T
6    - name: Incoming_Tick_Dropped
7      description: The rate group component's queue is full, so it cannot store
8        ↳ the tick coming in. This usually means the rate group is cycle slipping
9        ↳ and not running as fast as it needs to.
10     param_type: Tick.T

```

Example_Command_Router IDed Entity Models

The Example_Command_Router component contains IDed entity models for commands and events.

example_command_router.commands.yaml:

```

1  ---
2  description: These are the commands for the Example Command Router component.
3  commands:
4    - name: Noop
5      description: Simple NOOP command which produces an event saying that it was
6        ↳ triggered.

```

example_command_router.events.yaml:

```

1  ---
2  events:
3    - name: Command_Received
4      description: A command was received by the command router to be routed.
5      param_type: Command_Header.T
6    - name: Noop_Received
7      description: A Noop command was received.
8    - name: Invalid_Command_Received
9      description: A command was received with invalid parameters.
10     param_type: Invalid_Command_Info.T

```

The Example_Data_Collector component contains an IDed entity model for data products.

example_data_collector.data_products.yaml:

```

1  ---
2  description: Data products for the Example Data Collector Component.
3  data_products:
4    - name: Sensor_1_Data
5      description: Sensor data value 1.
6      type: Packed_U32.T
7    - name: Sensor_2_Data
8      description: Sensor data value 2.
9      type: Packed_U32.T

```

Example_Science IDed Entity Models

The Example_Science component contains IDed entity models for commands, events, parameters, and data products.

example_science.commands.yaml:

```

1  ---
2  description: These are the commands Example Science Component
3  commands:
4    - name: Enable_Science
5      description: Start collecting science.
6    - name: Disable_Science
7      description: Stop collecting science.

```

example_science.events.yaml:

```

1  ---
2  events:
3    - name: Science_Started
4      description: Science collection has started.
5    - name: Science_Stopped
6      description: Science collection has stopped.

```

example_science.parameters.yaml:

```

1  ---
2  description: Parameters for the Example Science Component.
3  parameters:
4    - name: Calibration_Coefficient_1
5      description: Science calibration coefficient for sensor 1.
6      type: Packed_F32.T
7      default: "(Value => 1.0)"
8    - name: Calibration_Coefficient_2
9      description: Science calibration coefficient for sensor 2.
10     type: Packed_F32.T
11     default: "(Value => 1.0)"

```

example_science.data_products.yaml:

```

1  ---
2  description: Data products for the Example Science Component.
3  data_products:
4    - name: Science_1_Data
5      description: Science data value 1.
6      type: Packed_F32.T
7    - name: Science_2_Data
8      description: Science data value 2.
9      type: Packed_F32.T

```

example_science.data_dependencies.yaml:

```

1  ---
2  description: Data dependencies for the Example Science Component.
3  data_dependencies:
4    - name: Sensor_1_Reading
5      description: Sensor data value 1, as input to science processing.
6      type: Packed_U32.T
7    - name: Sensor_2_Reading
8      description: Sensor data value 2, as input to science processing.
9      type: Packed_U32.T

```

7.3.2 IDed Entity ID Assignment

The IDs for the different entities described in the last section can be set in one of two ways 1) manually by defining the *ID base* for that IDed suite within the assembly model, or 2) by allowing Adamant to automatically assign the IDs. Note that data dependency ID assignment is done slightly

differently, and requires the developer to specify the mapping from data dependencies to data products within the system. Data dependency ID resolution is discussed at the end of this section.

Note that the ID spaces for each of the IDed entity types should be treated as completely independent, even though some IDs may be shared. Specifically, the ID space for commands is a separate and distinct ID space from the one used for events. In Ada, this is enforced by using a different type for the two ID spaces so that they may not be intermixed. So even though there may be a command with ID 4 and an event with ID 4, they are not the same thing.

IDs within an IDed entity model are always contiguous. Once exception to this is a packet model where the global IDs are assigned manually, see Section 6.12. Another exception is a data dependency model where the IDs are mapped to corresponding data products manually, see Section 6.11. This means that if a entity model contains 3 entities, the IDs for those entities will be incrementing integers, one after the other, starting at the ID base. For example, the *example_command_router.events.yaml* model defined above contains 3 events. If the ID base for this model is set to 7, then the three events will have IDs 7, 8, and 9.

As described earlier, the ID base may be set manually by the user for an entity model within the assembly model via the `set_id_bases` tag. For example, within the *science_assembly.assembly.yaml* model, Section 7.1.2, we set the ID base of the `science_Instance`'s data products to 100. This means that the IDs of the data products `Sensor_1_Data` and `Sensor_2_Data` will be 101 and 102, respectively.

If the ID base is not set manually by the user in the assembly model, then Adamant assigns the ID for that entity model. The assigned IDs follow a simple algorithm. After all the user IDs have been assigned, Adamant looks for the lowest contiguous set of IDs that can fit the ID space for that IDed entity model. For instance, for the *example_command_router.events.yaml* model, the lowest set of 3 contiguous IDs unused by other components would be assigned. In this way, Adamant chooses the *ID base* for that entity model. The computed ID bases can be seen reflected in the `Set_Id_Bases` procedure within the assembly package's autocoded body.

If no ID bases are specified by the user within the assembly model for an ID entity type, the resulting generated ID space for that entity type would start at 0 (or 1 depending on the IDed entity type) and end at the number of entities of that type. Adamant always strives to produce the most compact ID space with the lowest possible integer ID values. The benefit of this kind of ID space is that data structures using the ID as index can efficiently be implemented as arrayed look up tables.

While generating any outputs of an assembly model, including the autocode, the modeling system will check the user defined IDs to ensure that they do not conflict or overlap. This ensures that all entities are assigned a unique ID. An error is thrown if a conflict is found.

Using this system, the developer can choose the ID space they want for each type of entity. Often the IDs for commands will be defined manually to create a sparse ID space. Working this way makes sure that command IDs remain relatively stable over the course of a project, which might be beneficial for debugging and trending. Conversely, data product IDs are rarely specified manually, since these IDs are not configuration controlled like commands. The resulting compact ID space makes a database of data products easy to implement in an efficient manner, using Adamant's ID assigning behavior to its advantage. See *src/components/product_database* for an example of a data product database component.

Sometimes you want Adamant to define a compact ID space for the IDed entity, but you want to define where the ID space starts, ie. the minimum ID. Adamant allows a developer to specify minimum on a per ID type basis using the following pattern within an assembly model:

id_base_demo_assembly.assembly.yaml:

```

1  ---
2  # We set events to start at 0x1000 and data products to
3  # start at 0x2000
4  id_bases:
5    - "event_Id_Base => 4096"
6    - "data_Product_Id_Base => 8192"
7
8  # Below is the rest of the assembly model.
9    ...
10 components:
11   ...
12 connections:
13   ...
14 etc.

```

The above model sets the event and data product ID spaces separately. Other IDed entity types, those not specified, will start at their default minimum of 0 or 1. In this way we can make the autogenerated ID space start wherever we like.

Data dependency ID specification is done in a slightly different manner, via the `set_data_dependencies` tag within the assembly. For example, within the `science_assembly.assembly.yaml` model, Section 7.1.2, we map the `science_Instance`'s two data dependencies to two external data products in the system, ie. `"sensor_1_Reading => data_Collector_Instance.Sensor_1_Data"` which maps the `science_Instance`'s `sensor_1_Reading` data dependency to the `data_Collector_Instance`'s `Sensor_1_Data` data product. Adamant will ensure that both the data product and data dependency exist in the assembly and are of the same type. If these checks pass, then the IDs of the data dependencies will be resolved to the appropriate data product IDs in the `Set_Id_Bases` procedure within the assembly package's autocoded body. In this way, components can safely share data among one another while retaining their compile-time independent property.

7.3.3 IDed Entity Ada Specification

For every IDed entity type in an assembly, Adamant provides an autocoded Ada specification file that contains useful constants for configuring the system. These constants are often used while configuring the initialization parameters of component in the assembly model.

As an example, for the `science_assembly` we can build the specification for events by running:

```
> redo build/src/science_assembly_events.ads
```

Note that the analogous build rules exist for commands, parameters, data products, packets, and faults as well. Let's look at the autocoded contents.

`science_assembly_events.ads:`

```

1 -----
2 -- Science_Assembly Assembly Spec
3 --
4 -- Generated from science_assembly.assembly.yaml on 2025-07-15 21:59.
5 -----
6
7 with Event_Types; use Event_Types;
8
9 -- Event related constants for the Science_Assembly assembly.

```

```

10 package Science_Assembly_Events is
11
12     -- Assembly-wide constants:
13     Minimum_Event_Id : constant Event_Id := 1;
14     Maximum_Event_Id : constant Event_Id := 7;
15     Number_Of_Events : constant Natural := 7;
16     Number_Of_Components_With_Events : constant Natural := 3;
17
18     -- List of event ids:
19     Rate_Group_Instance_Cycle_Slip : constant Event_Id := 1; -- 0x0001
20     Rate_Group_Instance_Incoming_Tick_Dropped : constant Event_Id := 2; -- 0x0002
21     Command_Router_Instance_Command_Received : constant Event_Id := 3; -- 0x0003
22     Command_Router_Instance_Noop_Received : constant Event_Id := 4; -- 0x0004
23     Command_Router_Instance_Invalid_Command_Received : constant Event_Id := 5; --
24         --> 0x0005
25     Science_Instance_Science_Started : constant Event_Id := 6; -- 0x0006
26     Science_Instance_Science_Stopped : constant Event_Id := 7; -- 0x0007
27
28 end Science_Assembly_Events;

```

As can be seen there are constants relating the minimum ID, maximum ID, and total number of events in the system. There is also a constant that supplies the ID for every event in the system.

7.3.4 IDed Entity Documentation

Adamant provides automatic generation of documentation for all IDed entities in an assembly. Currently two versions of documentation can be created: HTML, which is useful for presentation in meetings or for quick reference, and PDF (via L^AT_EX), which is useful for formal documentation. PDF documentation is produced as part of an Assembly Design Document and is not discussed here. See Section 7.5 for details.

HTML documentation is available for all IDed entities. As an example we will build the events HTML. From the assembly directory run:

```
> redo build/html/science_assembly_events.html
```

which when opened with your favorite web browser looks something like this:

Science_Assembly Events

Description: This is the example assembly.

Summary:

- Number of Components with Events: 3
- Number of Events: 7

Event Id	Event Name	Parameter Type	Description
0x0001 (1)	rate_Group_Instance.Cycle_Slip	Cycle_Slip_Param.T	Execution ran long on this cycle.
0x0002 (2)	rate_Group_Instance.Incoming_Tick_Dropped	Tick.T	The rate group component's queue is full, so it cannot store the tick coming in. This usually means the rate group is cycle slipping and not running as fast as it needs to.
0x0003 (3)	command_Router_Instance.Command_Received	Command_Header.T	A command was received by the command router to be routed.
0x0004 (4)	command_Router_Instance.Noop_Received	-	A Noop command was received.
0x0005 (5)	command_Router_Instance.Invalid_Command_Received	Invalid_Command_Info.T	A command was received with invalid parameters.
0x0006 (6)	science_Instance.Science_Started	-	Science collection has started.
0x0007 (7)	science_Instance.Science_Stopped	-	Science collection has stopped.

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/science_assembly.yaml on 2020-05-09 02:22.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

Note that the analogous build rules exist for commands, parameters, data products, packets, and faults as well.

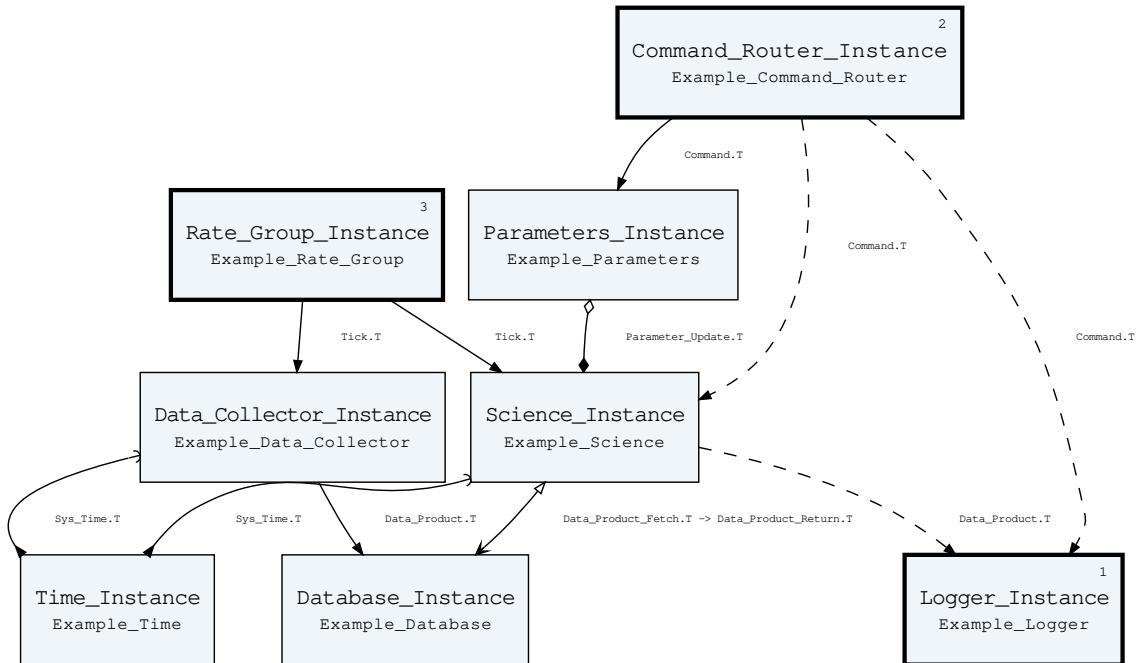
7.4 Assembly Views

Assembly diagrams, Section 7.1.3, can get complex and hard to understand as more components and connections are added. In order to provide a more digestible way of consuming the information in an assembly model, the concept of assembly *views* is introduced. A view is a diagram that only shows a subset of the entire assembly model in order to demonstrate a particular point to a set of stakeholders. Because a view is focused, and not cluttered by information that may distract from its main message, it is usually more adept at presenting certain aspects of the architecture. A collection of views can generally communicate architectural ideas much more effectively than an entire assembly diagram, making for more relevant discussions and more valuable design reviews.

Assembly diagrams in Adamant are directed graphs. For visualization Adamant uses [Graphviz](#), in particular the [DOT language](#) language which excels at presenting directed graphs.

7.4.1 Creating a View Model

A view is specific to the assembly it is associated with. A view model can be thought of a “graph filter” for the assembly. Imagine the entire, complex assembly graph. We will be working with the *science_assembly* diagram that was generated in Section 7.1.3. The diagram is shown again below for reference.



Using a view model, we can apply filters to this diagram to limit what is shown. When the view filter’s are applied, many components and connections are removed, leaving a subgraph that displays the desired “view” of the assembly. The view model provides many mechanisms that allow filtering of the assembly graph, which will be presented in this section.

Creating views in Adamant is achieved using a *view* model. To create a view for *science_assembly*, we simply need to name the view model appropriately. View models will always be of the form

`view_name.assembly_name.view.yaml` where `view_name` is the name of the specific view and `assembly_name` is the name of the assembly that the view applies to. Note that there is no limit to how many view models can be created for an assembly.

Usually it is best to create view models in their own subdirectory under the assembly directory. We begin by creating a `views` directly:

```
> mkdir views # make views/ directory under science_assembly/
> cd views
```

In this directory we create our first view model named `command_view.science_assembly.view.yaml`. The contents are shown below:

`command_view.science_assembly.view.yaml`:

```

1  ---
2  # Optional - a description of the view
3  description: This is the command view.
4  # Optional - layout direction, default is ;left-to-right'.
5  # Other options include: 'top-to-bottom', 'right-to-left',
6  # and 'bottom-to-top'.
7  layout: left-to-right
8  #
9  # Below is a set of switches that turn items on and off in the
10 # view. Providing these is optional. By default all of these
11 # are set to "True".
12 #
13 # Show the component type name inside a component box
14 show_component_type: True
15 # Shows components outlined in bold for active components and
16 # not outlined in bold for passive components
17 show_component_execution: True
18 # Shows the component's priority number
19 show_component_priority: False
20 # Shows the component instance name inside a component box
21 show_component_name: True
22 # Shows the type of the connector next to the arrow
23 show_connector_type: True
24 # If set to false shows a dotted line around "groups" in the view
25 hide_group_outline: True
26 # If set to true shows the data dependency pseudo connectors
27 show_data_dependencies: False
28 # Optional - the "rule" used to determine how the filters are
29 # applied to the assembly graph. If not specified, then the rule
30 # defaults to "and"-ing all the subgraphs from all the filters
31 # together.
32 rule: include_commands
33 # Optional - A list of filters used to filter components and
34 # connections from the assembly graph.
35 filters:
36     # Required - The name of the filter. Any name can be used but
37     # it must be unique.
38     - name: include_commands
39         # Required - The filter type. Filter types available are:
40         # component_name - filter by component names
41         # component_type - filter by component types
42         # component_execution - filter by component execution types (active or
43             ↳ passive)
        # component_context - filter by component and all it's immediate
            ↳ connections
```

```

44      # connector_name - filter by connector_names
45      # connecter_type - filter by connector type
46      # connecter_kind - filter by connector kinds (recv_async, recv_sync, send,
47      # request, service, get, return, provide, modify)
48      type: connector_type
49      # Required (unless "exclude" is specified instead) -
50      # Describes which items to INCLUDE in the view using the
51      # filter. If "exclude" was used instead then the list would
52      # described the items to EXCLUDE from the view.
53      include:
54          - Command.T

```

This YAML model file above contains comments explaining whether or not each field is optional or required. The model begins with an optional description of what the view shows. Using the optional `layout` tag, the developer can tell Graphvis the direction they would like the graph drawn. This is equivalent to setting the `rankdir` attribute in Graphvis. The options for this tag are: `left-to-right`, `top-to-bottom`, `right-to-left`, or `bottom-to-top`.

Following is a set of `show_*` switches. By default, if left unspecified, these tags are set to `True`. A description of each is shown below:

- **`show_component_type`** - Shows the component type name inside each component box.
- **`show_component_execution`** - Shows the component outlined in bold for active components and not outlined in bold for passive components.
- **`show_component_priority`** - Shows the component priority number inside each active component box.
- **`show_component_name`** - Shows the component instance name inside each component box.
- **`show_connector_type`** - Shows the connector type next to each connection.
- **`hide_group_outline`** - If set to `False`, it shows a dotted line around any “groups” defined in the view
- **`show_data_dependencies`** - This flag is an exception, it is default set to `False`. If set to `True`, any data dependencies defined in the assembly are displayed.

Below the switches is the `rule` definition. Defining a rule is optional, and usually is not necessary. However, the rule feature gives a powerful way to define very specific subgraphs that otherwise would not be possible. It is easiest to describe rules after understanding `filters`, which will be described next. Rules are explained in Section 7.4.2. In this case, the view only has a single filter, so a rule is unnecessary, since we just want to apply this filter to the assembly graph to produce our subgraph.

Below the `rule` definition is the `filters`. Any number of filters can be defined. Each filter takes the entire assembly graph as input and produces a subgraph as output. By default, if no rule is specified, the intersection of these subgraphs is taken in order to produce the view. That is, any shared connections and components that appear in ALL of the subgraphs is included in the view.

In this case we define a single filter. A name must be provided for the filter, and it must be unique within the view. A filter type must also be provided. Along with the filter type is one of two lists (but never both): an `include` list or an `exclude` list. If an `include` list is provided, then the assembly graph is filtered to *include* any components or connections that meet the filter criteria to create the subgraph. If an `exclude` list is provided, then the assembly graph is filtered to include any components or connections that not NOT meet the filter criteria to create the subgraph.

There are three different types of view filters: 1) those that filter components, 2) those that filter connectors, and 3) those that filter data dependencies. A component filter will remove any components that do not meet the filter criteria. Any connections that are shared between the components are left, and any connections that connect to removed components are pruned from the subgraph. The

component filters are shown below:

- **component_name** - Filter by component instance names, ie. Command_Router_Instance.
- **component_type** - Filter by component type names, ie Example_Command_Router.
- **component_execution** - Filter by component execution, either active or passive.
- **component_name_context** - Keep (or remove in the case of an exclude list) all components that have immediate connections with the specified component instances.
- **component_type_context** - Keep (or remove in the case of an exclude list) all components that have immediate connections with the specified component types.

A connector filter will remove any connectors that do not meet the filter criteria. If any connector of a connection is filtered out, then both connectors of the connection are filtered out. After all the connections are removed, any component that is left with no connections to another component is also pruned. The connector filters are shown below:

- **connector_name** - Filter by connector name in the form component_instance_name.connector_name.
- **connector_type** - Filter by connector type ie. something like Command.T.
- **connector_kind** - Filter by connector kind, ie. recv_async, recv_sync, send, request, service, get, return, provide, or modify

In this case we use the connector_type filter type with an include list. In this way we are trying to include any connectors that are of type Command.T. Any components that do not have a Command.T connector will be removed.

Data dependency filters should always be used in conjunction with **show_data_dependencies** flag set to True. These filters operate independently of connector filters, so data dependency filters will leave all connectors unfiltered, and vice versa.

- **data_dependency_name** - Filter by data dependency or data product names, ie. Register_Stuffer_Consumer_Instance.Last_Register_Written or Register_Stuffer_Instance.Last_Register_Written.
- **data_dependency_type** - Filter by type passed along a data dependency, ie Register_Value.T.

The presence of the view model allows Adamant to provide new build rules:

```
> redo what
redo what
redo all
redo clean
redo clean_all
redo templates
redo publish
redo targets
redo prove
redo analyze
redo style
redo pretty
redo test_all
redo analyze_all
redo coverage_all
redo build/dot/science_assembly_command_view.dot
```

```

redo build/eps/science_assembly_command_view.eps
redo build/png/science_assembly_command_view.png
redo build/svg/science_assembly_command_view.svg

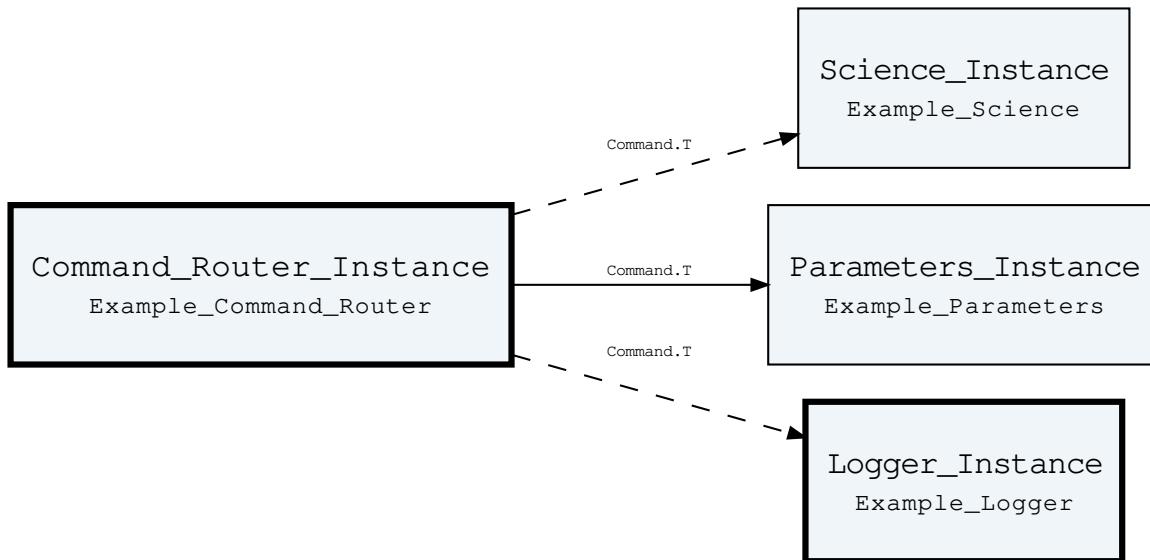
```

As can be seen there is a rule to build diagrams as .eps, .svg, or .png. Each of these diagrams will also build the .dot file which is autocoded Graphvis DOT language that describes the graph.

The most convenient to build usually the .svg which can be built with:

```
> redo build/svg/science_assembly_command_view.svg
```

When opened in your favorite web browser the view will look like:



As expected, we see a view that only shows the command paths of the architecture.

Building a view based on connector types of interest is a very common way to construct views. Another common pattern is construct a view around a single component of interest. For this we can use the `component_name_context` filter which will create a subgraph of the component and only its immediate connections.

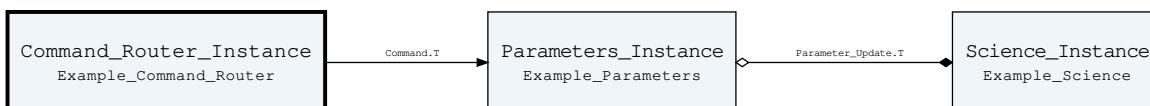
`parameters_context_view.science_assembly.view.yaml:`

```

1  ---
2  description: This is a parameters view.
3  show_component_priority: False
4  filters:
5    - name: include_component_types
6      type: component_name_context
7      include:
8        - parameters_Instance

```

The above YAML produces the following view, which shows how the `parameters_Instance` is situated in the assembly.



This is an example of building a view using data dependency filters with the proper flag.

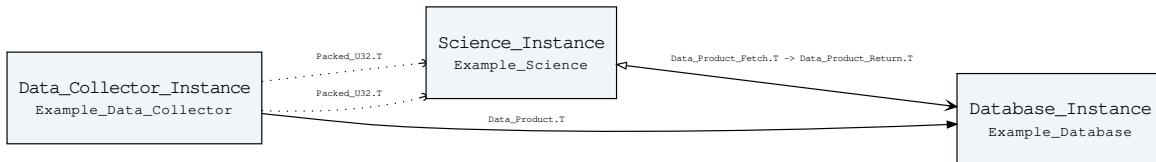
`data_collector_view.science_assembly.view.yaml:`

```

1  ---
2  description: This is view showing data dependencies.
3  show_component_priority: False
4  show_data_dependencies: True
5  filters:
6    - name: include_data_dependencies
7      type: data_dependency_name
8      include:
9        - Data_Collector_Instance.Sensor_1_Data
10       - Science_Instance.Sensor_2_Reading
11    - name: include_specific_components
12      type: component_name
13      include:
14        - Data_Collector_Instance
15        - Science_Instance
16        - Database_Instance

```

In the below diagram the dependencies are represented by the thin dotted lines. The necessity of excluding extraneous components/connectors in the view is as a result of the exclusivity of connector and data dependency filters.



Data dependency "connectors" are different than traditional Adamant connectors. They are not direct connections between two components, they only indirectly depict data product flow which is actually routed through the database. Even though these are not direct models of how data is passed in Adamant, the data dependency "connectors" still provide useful information for understanding data flow. Depicted in the view above is both the true path the data product takes, through the database, and the data dependency "connections". Visually it is much easier to tell that the Science_Instance consumes the data from the Data_Collector_Instance with the data dependency "connectors" as opposed to the actual connections routed through the database.

7.4.2 Using View Rules

Now let's create some views that demonstrate the power of view rules. A rule tells Adamant how to apply the filters specified in a view model. Defining a rule uses a simple language with the following operands.

- **&** - Perform an *intersection*, or *and*, between the two subgraphs. Only components and connections found in both subgraphs will be included in the resulting subgraph.
- **|** - Perform a *union*, or *or*, between the two subgraphs. Any components and connections in the subgraphs will be combined together to form a larger subgraph which contains both.
- **~** - Perform a *not* operation on the subgraph. The resulting subgraph will contain only components and connections not included in the input.
- **()** - Parenthesis can be used to enforce order of operations.
- **@** - Define the included subgraph as a "group" which will be graphed independently of the rest of the view. See Section 7.4.3 for more details.

For clarity a few rules are written below and explained in English:

- **f1 & ~f2** - Perform an *and* operation, or *intersection*, between the subgraphs created by filter f1 and the subgraph created by the *not* of filter f2. The *not* of f2 will include every component and connection not included in the subgraph created by f2.
- **f3 & (f1 | f2)** - Perform an *and*, *intersection*, between the two subgraphs f3 and the subgraph resulting from the *or*, *union*, of f1 and f2.
- **@f3 | f1** - Perform an *or*, *union*, between f3 and f1. When producing the graph, graph f3 by itself as a Graphvis *group* before adding the elements in f1. See Section 7.4.3 for more details.

Let's demonstrate the use of rules with a few examples. First, consider the *parameters_context_view.science_assembly.view.yaml* view model that was created at the end of the last section. There are other ways to make this same view diagram with a different model. Consider the model file below.

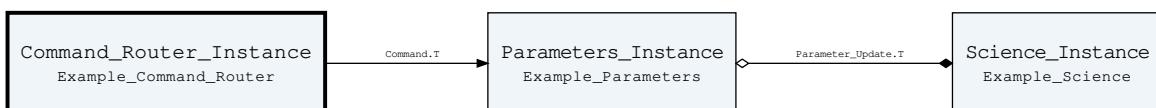
parameters_view.science_assembly.view.yaml:

```

1  ---
2  description: This is a parameters view.
3  show_component_priority: False
4  rule: include_component_types & exclude_connector
5  filters:
6    - name: include_component_types
7      type: component_type
8      include:
9        - Example_Parameters
10       - Example_Science
11       - Example_Command_Router
12    - name: exclude_connector
13      type: connector_name
14      exclude:
15        - Science_Instance.Command_T_Recv_Async

```

In this case we create a large subgraph, created by filter `include_component_types`, that includes all the components that we are interested in. However, we do not want to show the `Command.T` connection from the `command_Router_Instance` to the `science_Instance`. We remove this connection by *anding* `include_component_types` with the `exclude_connector` filter, which creates a subgraph without the connector we want to remove. The resulting view is what we expect.



Note that this view would still work without specifying a `rule`. If left unspecified, the default rule is to *and* all the filters together in order, which is exactly what we have specified manually here.

Let's try generating the same view in yet another way. This time we will use the *or* operand.

parameters_view2.science_assembly.view.yaml:

```

1  ---
2  description: This is also a parameters view.
3  show_component_priority: False
4  rule: include_param_connectors | include_parameter_command
5  filters:
6    - name: include_param_connectors
7      type: connector_type

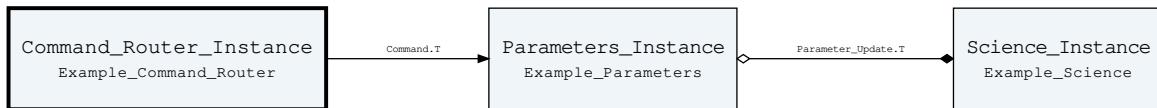
```

```

8   include:
9     - Parameter_Update.T
10  - name: include_parameter_command
11    type: component_name
12    include:
13      - command_Router_Instance
14      - parameters_Instance

```

We create two subgraphs from the two filters. `include_param_connectors` creates a subgraph that includes any `Parameter.T` typed connections. This only applies to two components, `parameters_Instance` and `science_Instance`. The `include_parameter_command` filter simply creates a subgraph with the two components `command_Router_Instance` and `parameters_Instance`. We then combine, via `or`, the two subgraphs together to create the desired view:



Finally, to demonstrate the `not` rule operand, see the following model file. In this case we are trying to create a view that contains everything but the `command_Router_Instance` and `parameters_Instance`. One way to do this is to create a filter that includes both of these components and then use the `not` operator to invert it.

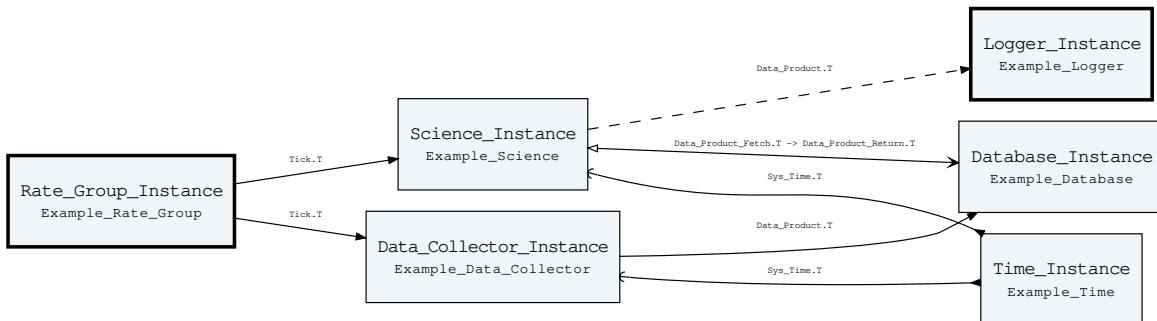
`no_command_params.science_assembly.view.yaml`:

```

1  ---
2  description: This is also a view without commands or parameters.
3  show_component_priority: False
4  rule: ~include_parameter_command
5  filters:
6    - name: include_parameter_command
7      type: component_name
8      include:
9        - command_Router_Instance
10       - parameters_Instance

```

The rule creates the view that we wanted.



7.4.3 Configuring View Graph Layout

When creating views, sometimes the resulting computer-generated graph is not exactly how you imagined it looking. While you cannot drag and drop components and connections into different locations, Adamant does provide a few ways to tweak the layout of a graph to make it more presentable.

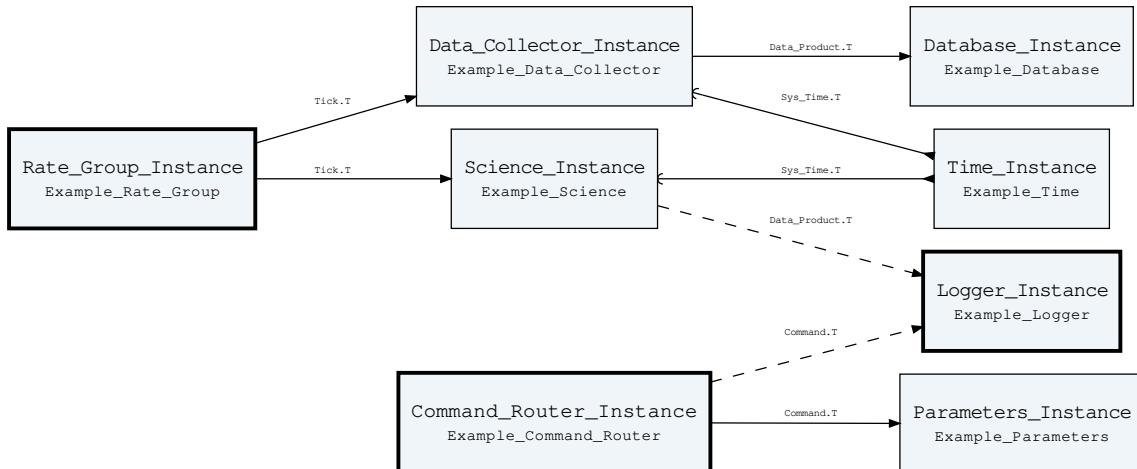
The first tool we will talk about is a `rule` operand for creating subgraph *groups* using the `@` symbol. Note this operand uses the underlying `subgraph cluster` feature of the DOT language.

grouped_view.science_assembly.view.yaml:

```
1  ---
2  description: This is a grouped view.
3  show_component_priority: False
4  rule: "@science | everything"
5  filters:
6    - name: science
7      type: component_name
8      include:
9        - rate_Group_Instance
10       - science_Instance
11       - time_Instance
12       - logger_Instance
13    - name: everything
14      type: component_type_context
15      exclude:
16        - Example_Science
```

Note that we used quotes around the rule to make it valid YAML. The YAML schema will complain if a value begins with @.

In this view model we combine the subgraphs of two filters together using *or* and we tell Adamant to create a *group* out of the first subgraph. The resulting view looks like:



As can be seen, there appears to be two graphs drawn that are connected together. The first graph is centered around the *science_Instance* and the second is centered around the *command_Router_Instance*. In this way, a user can use this feature to better organize their views.

Adamant also provides features that allow for the insertion of inline DOT at the beginning or end of the generated DOT file using the preamble and postamble tags, respectively. Any DOT code can be specified found in the [DOT language specification](#).

Most commonly, in Adamant, you will use the postamble tag to specify the rank of different components in the view graph. Specifying rank attributes allows us to change where components are drawn in the diagram. To demonstrate, we create a view that uses the *rank* attribute to make a cleaner version of the *science_assembly* diagram. No components or connections are filtered out in this case.

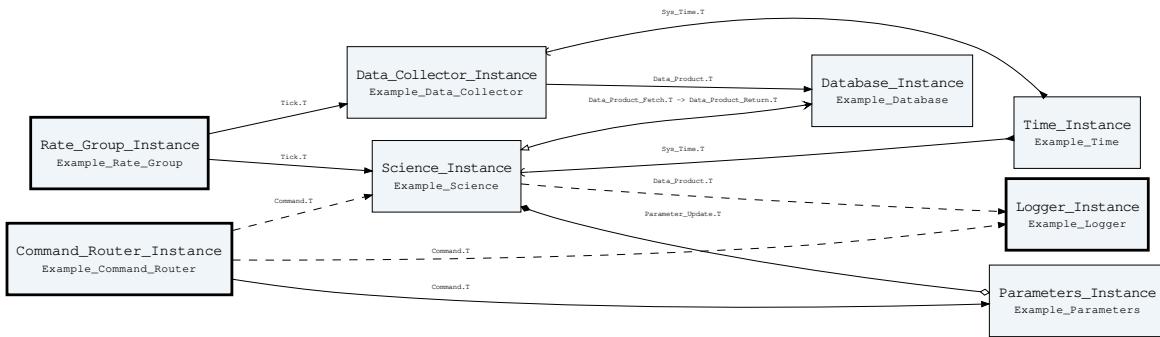
science_assembly_view.science_assembly.view.yaml:

```

1  ---
2  description: This is the assembly view.
3  show_component_priority: False
4  postamble: "{ rank=sink; Parameters_Instance, Time_Instance, Logger_Instance }"

```

which produces the view:



For this view model we specified 3 components at the *sink rank*. This tells Graphvis to draw the graph with these three components on the far right, one on top of the other. The resulting view is nice, compact, and easy to understand. Note that when using this feature you MUST spell the component instance names exactly correct, including the correct capitalization, otherwise an error or odd looking view will result.

More than one rank and group can be specified by adding another set of “{}” with DOT inside. Other useful rank values include *source*, which is the opposite of *sink*, and *same* which groups the specified components at the same level in the graph.

For more information on DOT rank specifications see [this documentation](#).

Creating good view diagrams takes practice and trial and error. Often you will need to create the view, see how it looks, and then slowly make adjustments until it looks the way you want. For more examples, take a look at the [example repository views](#).

7.5 Assembly Documentation

One of the benefits of the extensive modeling required by Adamant is that rich, informative documentation can be generated with little extra effort. This documentation is beneficial at all stages of the development process, from working out the details of the design, to reviewing the implementation.

It is advisable to generate assembly documentation as soon as a solid assembly model has been developed for a system. In this way, the document can be used to explain the design to other engineers and gain feedback before the implementation has even begun. During the development process, the design document should periodically be regenerated to reflect any changes. Finally, when the assembly is complete, the design document should be reviewed to ensure that it contains all the information necessary for another engineer to use or make changes to the design. Sometimes this requires adding custom sections to the design document that the document autocoder would not be able to create on its own.

This section walks through how to generate documentation for assemblies.

7.5.1 Assembly HTML

Adamant provides some handy HTML tables for collating assembly design information. Some of the most useful HTML products were described for assembly IDed entities in Section 7.3.4. In addition, the following rules are provided, using the *science_assembly* as an example:

Show information about all components in assembly:

```
> redo build/html/science_assembly_components.html
```

Show information about all connections in assembly:

```
> redo build/html/science_assembly_connections.html
```

Show information about all interrupts in an assembly:

```
> redo build/html/science_assembly_interrupts.html
```

Show information about all tasks and their priorities:

```
> redo build/html/science_assembly_priorities.html
```

Show information about all queued components:

```
> redo build/html/science_assembly_queues.html
```

Below are some of sample outputs using the *science_assembly*.

science_assembly_components.html:

Science Assembly Components

Description: This is the example assembly.

Summary:

- Number of Components: 6
- Number of Component Types: 6
- Number of Active Components: 3
- Number of Passive Components: 3
- Number of Components with Queue: 3
- Number of Components without Queue: 3
- Number of Components with Events: 3
- Number of Components with Commands: 2
- Number of Connections: 7
- Number of Events: 7
- Number of Commands: 3

Name	Type	Has Queue	Execution	# Connectors	# Commands	# Events	# Data Products	# Packets	Description
rate_Group_Instance	Example_Rate_Group	no	active	2	0	2	0	0	
command_Router_Instance	Example_Command_Router	yes	active	2	1	3	0	0	
science_Instance	Example_Science	yes	passive	5	2	2	2	0	
time_Instance	Example_Time	no	passive	1	0	0	0	0	
parameters_Instance	Example_Parameters	no	passive	2	0	0	0	0	
logger_Instance	Example_Logger	yes	active	2	0	0	0	0	

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/science_assembly.assembly.yaml on 2020-05-09 02:37.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

science_assembly_connections.html:

Science_Assembly Connections

Description: This is the example assembly.

Summary:

- Number of Components: 6
- Number of Connections: 7

Number	From	To	Kind	Description
1	rate_Group_Instance.Tick_T_Send[1]	science_Instance.Tick_T_Recv_Sync	send-recv_sync	This is the first connection in the model
2	command_Router_Instance.Command_T_Send[1]	science_Instance.Command_T_Recv_Async	send-recv_async	
3	science_Instance.Sys_Time_T_Get[1]	time_Instance.Sys_Time_T_Return	get-return	
4	parameters_Instance.Parameter_T_Request[1]	science_Instance.Parameter_T_Service	request-service	
5	science_Instance.Data_Product_T_Send[1]	logger_Instance.Data_Product_T_Recv_Async	send-recv_async	
6	command_Router_Instance.Command_T_Send[2]	logger_Instance.Command_T_Recv_Async	send-recv_async	
7	command_Router_Instance.Command_T_Send[3]	parameters_Instance.Command_T_Recv_Sync	send-recv_sync	

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/science_assembly.assembly.yaml on 2020-05-09 02:37.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

science_assembly_priorities.html:

Science_Assembly Priorities

Description: This is the example assembly.

Task Number	Task Name	Priority Rank	Priority Value
0	rate_Group_Instance.Active_Task	1	3
1	command_Router_Instance.Active_Task	2	2
2	logger_Instance.Active_Task	3	1

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/science_assembly.assembly.yaml on 2020-05-09 02:43.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

science_assembly_queues.html:

Science_Assembly Queues

Description: This is the example assembly.

Summary:

- Number of Components with Queue: 3
- Number of Components without Queue: 3

Component Name	Component Type	Queue Size
command_Router_Instance	Example_Command_Router	1000
science_Instance	Example_Science	1000
logger_Instance	Example_LOGGER	1000

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/science_assembly.assembly.yaml on 2020-05-09 02:38.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

7.5.2 Assembly Design Document

One of the benefits of the extensive modeling required by Adamant is that rich, informative documentation can be generated with little extra effort. To generate documentation for an assembly, an assembly model must first exist. This section demonstrates how to generate documentation for the *science_assembly* that has been discussed in previous sections. Generating documentation for an assembly follows the same pattern as generating documentation for a component, which is discussed in Section 6.16.2.

Assembly documentation must be generated and stored in a subdirectory *doc/* of the assembly's main directory. In fact, Adamant only produces rules to build assembly documentation within a *doc/* directory. This convention makes it easy to locate the documentation for any assembly in the repository.

To create documentation for the *science_assembly* we first create the doc directory:

```
> mkdir doc # make doc/ directory under science_assembly/
> cd doc
```

After we are inside the new *doc/* directory we can view the build rules that are available.

```
> redo what
redo what
redo all
redo clean
redo clean_all
redo templates
redo publish
redo targets
redo prove
redo analyze
redo style
redo pretty
redo test_all
redo analyze_all
redo coverage_all
redo build/pdf/science_assembly_commands.pdf
redo build/pdf/science_assembly_components.pdf
redo build/pdf/science_assembly_connections.pdf
redo build/pdf/science_assembly_data_products.pdf
redo build/pdf/science_assembly_description.pdf
redo build/pdf/science_assemblyEnums.pdf
redo build/pdf/science_assembly_events.pdf
redo build/pdf/science_assembly_packets.pdf
redo build/pdf/science_assembly_priorities.pdf
redo build/pdf/science_assembly_stats.pdf
redo build/pdf/science_assembly_types.pdf
redo build/pdf/science_assembly_views.pdf
redo build/template/science_assembly.tex
redo build/tex/science_assembly_commands.tex
redo build/tex/science_assembly_components.tex
redo build/tex/science_assembly_connections.tex
redo build/tex/science_assembly_data_products.tex
redo build/tex/science_assembly_description.tex
redo build/tex/science_assemblyEnums.tex
redo build/tex/science_assembly_events.tex
redo build/tex/science_assembly_packets.tex
redo build/tex/science_assembly_priorities.tex
```

```
redo build/tex/science_assembly_stats.tex
redo build/tex/science_assembly_types.tex
redo build/tex/science_assembly_views.tex
```

As can be seen, there are many different *.tex* files that can be created, each that can be used to produce a small PDF stub of that piece of documentation. Most importantly there is a rule to build the documentation template, which is the master L^AT_EX file for the assembly documentation. To build and use the template we run:

```
> redo build/template/science_assembly.tex
> cp build/template/* . # copy the template file into doc/
```

This template provides an excellent start to our assembly documentation and often provides sufficient detail to be used without modification. However, like all template files in Adamant, this is meant to be modified, if necessary, to add additional hand-written documentation. Let's take a look at the autogenerated *science_assembly.tex* file.

science_assembly.tex:

```
1 \input{common_packages.tex}
2
3 \begin{document}
4
5 \title{\textbf{Science Assembly} \\
6 \large\textit{Assembly Design Document}}
7 \date{}
8 \maketitle
9
10 \section{Description}
11 \input{build/tex/science_assembly_description.tex}
12
13 \section{Design}
14
15 \subsection{At a Glance}
16 \input{build/tex/science_assembly_stats.tex}
17
18 \subsection{Components}
19 \input{build/tex/science_assembly_components.tex}
20
21 \subsection{Views}
22 \input{build/tex/science_assembly_views.tex}
23
24 \subsection{Task Priorities}
25 \input{build/tex/science_assembly_priorities.tex}
26
27 \subsection{Commands}
28
29 \input{build/tex/science_assembly_commands.tex}
30
31 \subsection{Events}
32
33 \input{build/tex/science_assembly_events.tex}
34
35 \subsection{Data Products}
36
37 \input{build/tex/science_assembly_data_products.tex}
38
39 \section{Appendix}
40 \subsection{Connections}
41 \input{build/tex/science_assembly_connections.tex}
```

```

42 \subsection{Packed Types}
43 \input{build/tex/science_assembly_types.tex}
44
45 \subsection{Enumerations}
46
47 \input{build/tex/science_assemblyEnums.tex}
48
49 \end{document}

```

As can be seen, this file defines the sections of the assembly documentation and imports most of the text from external autogenerated *.tex* files. Feel free to modify this file to add extra sections to the assembly documentation.

To generate a PDF from this file run:

```
> redo build/pdf/science_assembly.pdf
```

By default, PDF files are generated in the *build/pdf/* directory. This allows the developer to review the output documentation before “publishing” it, ie. saving it in the repository. Once the documentation looks good, you can publish it manually by copying it into the *doc/* directory:

```
> cp build/pdf/science_assembly.pdf .
```

or alternatively you can run the special Adamant command:

```
> redo publish
```

which will build the documentation and perform the copy for you. Note that any assembly views, Section 7.4, found in subdirectories of the model directory will automatically be added to the documentation with a caption that matches the view’s description.

The produced documentation is not shown here for brevity, but is linked [here](#). For a more complete example, check out the [example repository assembly design document](#).

7.6 Assembly Metrics

Since an executable assembly is usually used to build the final production binary for a project, it is often valuable to keep track of certain code metrics such as the number of lines of code, packages, files, etc. for each assembly. Adamant provides the ability to generate metrics for any executable produced in the system. Steps to produce different metrics for any file, object, or binary in Adamant is discussed in Section 9.5. The same example there can be applied for an assembly binary to produce the SLOC and other metrics of interest for a final executable.

7.7 Assembly Ground System Integration

When an assembly is compiled into an executable binary and deployed to the target embedded system, often the only way to communicate with it is through some external system. In the domain of spacecraft flight software, this communication system is the ground. Since Adamant is model driven and generators can be constructed to produce any desired output, see Section 9.11.2, it is conceivable that any ground system can be supported. Currently there is support for integration with OpenC3 COSMOS, or the LASP-developed ground system Hydra, both of which are presented in this section.

7.7.1 OpenC3 COSMOS Integration

OpenC3 COSMOS is publicly available at <https://openc3.com/>.

Adamant provides support for OpenC3 COSMOS plugin generation, which allows interfacing with COSMOS. Any Adamant assembly can be controlled by COSMOS via commands and receive data from the Adamant system through the plugin interface configuration and packets.

COSMOS plugins are configured via plaintext configuration files. Command and telemetry packet definitions can be generated from an Adamant assembly model. For example, we can run this from the *science_assembly* directory created in previous sections (some output omitted for brevity):

```
> redo what
redo what
redo all
redo clean
redo clean_all
redo templates
redo publish
redo targets
redo prove
redo analyze
redo style
redo pretty
redo test_all
redo analyze_all
redo coverage_all
redo build/cosmos/plugin/science_assembly_ccsds_commands.txt
redo build/cosmos/plugin/science_assembly_ccsds_telemetry.txt
```

As can be seen, two COSMOS configuration files are created in subdirectories of *build/cosmos*. These configuration files can be used in conjunction with OpenC3 tools to generate and compile a COSMOS plugin, allowing COSMOS to parse and display telemetry from as well as send commands to an Adamant assembly.

A fully functional configuration is provided in the [example repository](#). The COSMOS plugin can be configured, compiled, and installed with the Linux assembly by following the [README.md](#).

By digging into the example project COSMOS configuration you can explore how it works and adapt it for your own system, or translate it for a different ground system.

7.7.2 Hydra Integration

Note that Hydra is not yet publicly available, but will be made so in the future.

Adamant provides support for interfacing with Hydra at the assembly level. Specifically, any assembly can be controlled by Hydra via commands and receive data from the Adamant system through data products and packets.

Hydra is configured via special XML files. Many of these files can be generated from an Adamant assembly model. To see what is available we can run `redo what` from an assembly directory. For example, we can run this from the *science_assembly* directory created in previous sections (some output omitted for brevity):

```
> redo what
redo what
redo all
redo clean
```

```

redo clean_all
redo templates
redo publish
redo targets
redo prove
redo analyze
redo style
redo pretty
redo test_all
redo analyze_all
redo coverage_all
redo build/hydra/Config/science_assembly.xml
redo build/hydra/Config/science_assembly_ccsds_commands.xml
redo build/hydra/Config/science_assembly_ccsds_packets.xml
redo build/hydra/Pages/science_assembly_cpu_monitor.xml
redo build/hydra/Pages/science_assembly_queue_monitor.xml
redo build/hydra/Pages/science_assembly_stack_monitor.xml
redo build/hydra/Scripts/science_assembly_packet_pages.prc

```

As can be seen, all hydra configuration files are created in subdirectories of *build/hydra*. The subdirectory names map to the standard Hydra configuration file structure.

The *science_assembly.xml* file is the main Hydra configuration file. It includes definitions for all the types, data products, events, etc. that are defined in the assembly. The *science_assembly_ccsds_commands.xml* and *science_assembly_ccsds_packets.xml* files are created from special generators for the CCSDS_Command_Depacketizer and CCSDS_Packetizer components, respectively (*src/components/CCSDS*). They add CCSDS support to Hydra for sending commands and receiving LASP formatted CCSDS packets.

Next, we can see that 3 Hydra pages can be generated for viewing packets from the Adamant monitor components (*src/components/monitors*).

Lastly, Adamant can generate the *science_assembly_packet_pages.prc* script which automatically generates pages for every packet that an assembly can produce.

Coordinating these autocoded files into a fully functional Hydra configuration is too complicated to describe in this document. However, a fully functional example is provided in the example repository [hydra configuration directory](#). Hydra can be spawned and interfaced with the Linux assembly by running the single command:

```
> redo run_with_hydra
```

from *src/assembly/Linux/main* within the example repository.

By digging into the example project Hydra configuration you can explore how it works and adapt it for your own system, or translate it for a different ground system.

8 Ground Tools

The ground-based testing and analysis tools included with Adamant are in their infancy. Current ground-based tools are powered by the autocoded python classes that can be generated for any packed type found in the software, see Section 5.1.8. Since Adamant has a way to decode/encode any data that comes from or needs to go to the Adamant-based embedded system, a myriad of tools can be produced. The current tools available can be found in *gnd/*. Included are tools for decommissioning events from a post mortem log and from a socket, as well as a tool for analyzing CCSDS packets and reporting errors. This section of the User Guide will be expanded when Adamant's ground tools become more extensive and mature.

Recently, support for a MATLAB version of the python tools have been added and can be found in *gnd/matlab*. These tools rely on autocoded .m files for Adamant packed records, in the same way that the python tools rely on autocoded .py files for Adamant packed records.

9 Advanced Topics

This section provides discussion on various advanced usages of Adamant. These topics are not necessarily "hard" to understand. However, the features presented are not as commonly used as those presented in other sections. Knowledge of this section is often only required by one member of a team using Adamant. Where appropriate, the sections above link to sections here to provide additional information.

9.1 Interfacing with C and C++

Ada includes extensive facilities to support multi-language development. See, [AdaCore Learn](#) for specific details. In Adamant, we have formalized Ada calling into external C or C++ libraries and provided some tooling to make this easier. Note, this assumes Ada is still being used as the *main program*. If you would rather learn by example see the `c_demo` or `cpp_demo` components in the *example repository*.

To include a C or C++ library we must first include the library in the Adamant build path, just like we would with Ada source code, ie.

```
> cd c_lib
> ls -a c_lib
. . . .all_path c_lib.c c_lib.h
```

The Adamant build system recognizes that there are C files in this repository because they are named `.h` and `.c`. C++ files should be named `.hpp` and `.cpp` to be recognized by Adamant. Running `redo what` shows us that we can compile the C files, as well as generate automatic Ada bindings to call into this C library.

```
> redo what
redo what
redo all
redo clean
redo build/obj/Linux/c_lib.o
redo build/template/Linux/c_lib_h.ads
```

Note that binding generation is specific to your build target, since different compilers will often bind to symbols with different naming. To build the Ada bindings run:

```
> redo templates
redo templates
redo build/template/Linux/c_lib_h.ads
```

Let's take a look at the C source code:

`c_lib.h:`

```
1 // C-lib data type
2 typedef struct {
3     unsigned int count;
4     unsigned int limit;
5 } c_data;
6
7 // C-lib function
8 unsigned int increment(c_data* data);
```

`c_lib.c:`

```

1 #include "c_lib.h"
2
3 unsigned int increment(c_data* data) {
4     // Increment count if it is less than limit,
5     // otherwise set equal to zero.
6     if (data->count < data->limit) {
7         data->count = data->count + 1;
8     } else {
9         data->count = 0;
10    }
11
12    // Return the current count.
13    return data->count;
14}
15

```

which provides a simple data structure and function that increments a value up to a limit before rolling over. Let's also look at the generated Ada bindings:

c_lib_h.ads:

```

1 pragma Ada_2012;
2
3 pragma Style_Checks (Off);
4 pragma Warnings (Off, "-gnatwu");
5
6 with Interfaces.C; use Interfaces.C;
7
8 package C_Lib_H is
9
10   -- C-lib data type
11   type C_Data is record
12     Count : aliased unsigned; --
13     ↳ /__w/adamant/adamant/doc/example_architecture/c_lib/c_lib.h:3
14     Limit : aliased unsigned; --
15     ↳ /__w/adamant/adamant/doc/example_architecture/c_lib/c_lib.h:4
16   end record with
17     Convention => C_Pass_By_Copy; --
18     ↳ /__w/adamant/adamant/doc/example_architecture/c_lib/c_lib.h:5
19
20   -- C-lib function
21   function Increment
22     (Data : access C_Data)
23     return unsigned --
24     ↳ /__w/adamant/adamant/doc/example_architecture/c_lib/c_lib.h:8
25   with
26     Import => True, Convention => C, External_Name => "increment";
27
28 end C_Lib_H;
29
30 pragma Style_Checks (On);
31 pragma Warnings (On, "-gnatwu");
32

```

As you can see, *c_lib_h.ads* includes Ada bindings for the C struct and the call to `increment`. Below is an example *main.adb* that uses the bindings to call the C library.

main.adb:

```

1 with c_lib_h;
2 with Interfaces; use Interfaces;
3 with Ada.Text_IO; use Ada.Text_IO;

```

```

4
5 procedure Main is
6     Data : aliased c_lib_h.c_data := (count => 0, limit => 3);
7     Result : Unsigned_32;
8 begin
9     for Idx in 0 .. 10 loop
10        Result := Unsigned_32(c_lib_h.increment(Data'Access));
11    end loop;
12    Put_Line(Unsigned_32'Image(Result));
13 end Main;

```

The same pattern described above works for C++ libraries except the binding file would be named `<cpp_lib_name>.hpp.ads`. Note that these files are provided as templates because it is quite likely that you may want to tweak these files to fit your specific usage of the C/C++ library. At the very least, the templates can give you a big head start.

If the naming convention, `<c_lib_name>.h.ads` for C libraries and `<cpp_lib_name>.hpp.ads` for C++ libraries, is used, then the Adamant build system will be able to automatically build both the Ada and C/C++ and link them together. If you deviate from this convention, you are on your own, and you may need to write custom `.do` files to get everything to compile and link appropriately.

More advanced patterns, like having the C/C++ call back into Ada, are possible but are not discussed here. See [AdaCore Learn](#) if the pattern presented above does not meet your needs.

9.2 IDE Integration

Adamant has been designed to not depend on any particular Integrated Development Environment (IDE) for development. Since interaction with the system is provided via a commandline interface, developers can choose to use any editor or IDE to develop code for Adamant. Modern IDEs provide much in terms of helping the developer, from push-button compilation to code block suggestions. Full integration of Adamant with an Ada-equippt IDE is on the framework roadmap, however some work in the vein has already been completed.

Most Ada IDEs require the use of a central `.gpr` project file in order to determine the source code and settings for a project. Since Adamant uses a custom redo-based build system, it does not require a project file in the same form. However, to allow easy integration of Adamant within an IDE, a `.gpr` file can be produced for any binary (`test.adb` or `main.adb`) that can be compiled in the system. For example, if you run:

```
> redo build/gpr/test.gpr
```

from a unit test directory or

```
> redo build/gpr/main.gpr
```

from the project main directory, a `.gpr` file will be produced that can be opened with an Ada-enabled IDE such as [GNAT Studio](#). As an example, running `redo built/gpr/main.gpr` from the `simple_package` unit test directory produces the following `.gpr` file:

`simple_package/test/build/gpr/test.gpr:`

```

1 -----
2 -- Generated on .
3 -----
4
5 project test is
6
7     for Object_Dir use "./obj";

```

```

8      for Source_Dirs use (
9          "../../..",
10         "../..",
11         "../../../demo_conf/build/src",
12         "../../../demo_conf"
13     );
14
15     for Main use ("../../../test.adb");
16
17 end test;

```

Opening this file with an IDE will allow you to view and modify any source code used to generate the unit test binary.

9.3 Coding Style

A coding style is a set of rules or guidelines used when writing the source code for a program. The rules generally pertain to the visual appearance of the code and the aim is to improve uniformity in the "look" of the code. Once developers become accustomed to a particular coding style, they can more easily read and understand source code written in that style. Adamant enforces a coding style for all Ada, Python, and YAML to enforce consistency and improve understandability.

The main tool for enforcing style is `redo style`. When run, this will check any Ada, Python, YAML, or autocoded Ada, Python, or YAML and ensure that the style guidelines are met. The following example is run in `doc/example_architecture/style_demo`. Which includes the following poorly written code:

main.adb:

```

1 With Ada.Text_IO; use Ada.Text_io;
2
3 procedure Main is
4     PROCEDURE print_hello(str : in String)      is
5         begin
6             put_Line ( "Hello, " & str & "!" );
7         end;
8     BeGiN
9     print_hello("world");
10    end Main;

```

main.py:

```

1 #!/usr/bin/env python3
2 import argparse
3 import sys
4
5 #
6 # This python utility is meant demonstrate non PEP8 (bad) style.
7 #
8 if __name__ == "__main__":
9     # Parse the commandline arguments:
10     parser=argparse.ArgumentParser(description="Test program that demonstrates
11         ↪ bad style.")
12     parser.add_argument(
13         "file",
14         metavar= "file.bin",
15         type=str,
16         help="A file.",
17     )

```

```

17     args = parser.parse_args()
18
19     # Read the post mortem file:
20     print("%s"%(args.file))

component_model_to_lint.component.yaml:

1  description : This component has a malformed YAML model.
2  execution: active
3  init:
4      description: An initialization function for this component.
5  parameters:
6      - name: is_Enabled
7          type: Boolean
8          default: "True"
9  connectors:
10     - type: Data_Product.T
11         kind: recv_async

```

We can run `redo style` on this code to see what lines of code are problematic:

```

> redo style
main.adb:1:01: (style) reserved words must be all lower case
main.adb:1:10: (style) bad casing of "Text_IO" declared at a-textio.ads:58
main.adb:1:32: (style) bad casing of "Text_IO" declared at a-textio.ads:58
main.adb:4:02: (style) reserved words must be all lower case
main.adb:4:02: (style) bad indentation
main.adb:4:12: (style) bad capitalization, mixed case required
main.adb:4:23: (style) space required
main.adb:4:24: (style) bad capitalization, mixed case required
main.adb:4:39: (style) space not allowed
main.adb:5:04: (style) "begin" in wrong column, should be in column 2
main.adb:6:16: (style) bad casing of "Put_Line" declared at a-textio.ads:566
main.adb:6:26: (style) space not allowed
main.adb:6:48: (style) space not allowed
main.adb:7:05: (style) bad indentation
main.adb:7:05: (style) "end print_hello" required
main.adb:7:05: (style) "end" in wrong column, should be in column 2
main.adb:8:01: (style) reserved words must be all lower case
main.adb:9:01: (style) incorrect layout
main.adb:9:12: (style) space required
main.py:3:1: F401 'sys' imported but unused
main.py:10:11: E225 missing whitespace around operator
main.py:13:17: E251 unexpected spaces around keyword / parameter equals
main.py:20:15: E228 missing whitespace around modulo operator
component_model_to_lint.component.yaml:1:1: [error] missing document start "---" (document_start)
component_model_to_lint.component.yaml:1:14: [error] too many spaces before colon (colon)
component_model_to_lint.component.yaml:10:8: [error] wrong indentation: expected 2 but found 1 (wrong_indentation)
component_model_to_lint.component.yaml:11:26: [error] trailing spaces (trailing_spaces)

```

The example above shows style warnings found for an Ada source file `main.adb`, a Python source file `main.py`, and a YAML model file `component_model_to_lint.component.yaml`. Style warnings are printed to the terminal and also stored in `build/style/style.log`.

The warnings can be removed by manually correcting the code to meet the standard. However, this can become cumbersome if there are many warnings. To aid in this process, Adamant provides the special command `redo pretty` which will "pretty" format any source code found in the current directory. Currently `redo pretty` only supports Ada and Python code, not YAML. Newly formatted

code will be produced in *build/pretty* and will adhere to the coding style.

```
> redo pretty
redo      doc/example_architecture/style_demo/pretty
redo  Prettifying:
/_w/adamant/adamant/doc/example_architecture/style_demo/main.adb
Traceback (most recent call last):
  File "/_w/adamant/adamant/default.do", line 93, in <module>
    rule.build(*sys.argv[1:])
  File "/_w/adamant/adamant/redo/base_classes/build_rule_base.py", line 70, in build
    to_return = self._build(redo_1, redo_2, redo_3)
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/_w/adamant/adamant/redo/rules/build_pretty.py", line 47, in _build
    gnatpp_cmd_prefix() + " --output-dir " + + out_dir + " "
                           ^^^^^^
TypeError: bad operand type for unary +: 'str'
Error: Redo script '/_w/adamant/adamant/default.do' failed to build '/_w/adamant/adamant/main.adb'
```

This will produce properly formatted Ada and Python versions of any code found in the current directory or autocoded in *build/src* or *build/py*.

The Adamant Ada coding style is based on style rules for the GNAT Ada compiler source code. Small modifications to this standard have been made to enforce some rules deemed important for writing safety critical software. Others rules have been loosened where the style was deemed overly strict, especially in regard to the style of comments. Examples of the Adamant coding style can be seen wherever the Ada programming language is reproduced in this User Guide. The coding style is enforced during compilation via the `-gnatyx` switch. The available style options are documented [here](#). Adamant uses the following configuration checking coding style: `-gnaty3aABbdDefhik1L12nOprStux`. To format proper Ada, `redo pretty` utilizes the Ada pretty printer tool [gnatpp](#).

The Adamant Python coding style adheres to [PEP 8](#). Adherence to the standard via `redo style` is provided by [flake8](#). To format proper looking Python, `redo pretty` utilizes [black](#).

Adamant YAML style is enforced when running `redo style` by [yamllint](#).

By default, coding style checks are not performed when compiling Ada source code, only when running `redo style`. However, these checks can be turned on while compiling by setting the appropriate environment variable.

```
> export CHECK_STYLE=True
```

This can be useful to ensure that all code you are currently working on and depending on meets the coding standard.

Using the `CHECK_STYLE` variable will enforce style checks during every compile. If this becomes tiresome, you can disable style checking by unsetting the variable:

```
> export CHECK_STYLE=
```

9.4 Static Analysis

Static analysis is used to search for potential vulnerabilities or defects in the source code. Adamant provides support for static analysis through the [GNAT SAS](#) tool developed by AdaCore. To make GNAT SAS easier to run on Adamant code, it has been integrated into the build system and can be

run via redo calls.

To invoke GNAT SAS on a set of code simply run:

```
> redo analyze
```

from any directory. Adamant supports running GNAT SAS in two modes: a) *library* mode and b) *binary* mode. If redo analyze is run in a directory where a *test.adb* or *main.adb* is found, it will be run in *binary* mode, otherwise it will be run in *library* mode. In *binary* mode, all source code found in the compiled binary is run through GNAT SAS (except for test-only source code which is never analyzed). In *library* mode, only the source code found (or generated) in the current directory is analyzed. Which mode is being utilized is printed in the output when running *redo analyze*.

Consider the example code below:

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Main is
4
5      Outer_Var : Integer := 6;
6      Old : Integer;
7
8      procedure Set_Global (New_Val : in Integer; Old_Val : out Integer) is
9      begin
10         if New_Val < 20 then
11             Old_Val := Outer_Var;
12         end if;
13         Outer_Var := New_Val;
14     end Set_Global;
15
16 begin
17     Set_Global (29, Old);
18     Put_Line ("Set to " & Integer'Image (Outer_Var) &
19               ", old value was " & Integer'Image (Old));
20 end Main;
```

The code above compiles with no warnings, but it has a defect that GNAT SAS can readily find. Let's run redo analyze.

```
> redo analyze
redo    analyze
redo  Analyzing Binary:
/home/user/adamant/doc/example_architecture/analyze_demo/main.adb
[gnatsas]: launching GNATcheck
[gnatsas]: launching Infer
[gnatsas]: launching Inspector
Compile
    [Ada]          main.adb
no partitioning needed.
starting the analysis of main.scil
analyzed main.scil in 0.00 seconds
starting the analysis of main_body.scil
analyzed main_body.scil in 0.00 seconds
re-starting the analysis of main.scil
re-analyzed main.scil in 0.00 seconds
analysis complete.
2 .scil files processed; 4 subprograms analyzed.
[gnatsas]: Analysis results stored in timeline "fast".
```

```
[gnatsas]: Run `gnatsas report -P../.gnatsas/home/user/adamant/doc/example_architecture
[sam-to-security]: Report generated at /home/user/adamant/../.gnatsas/home/user/adamant

-----
----- Analysis Output -----
-----

main.adb:8:4: high: validity check [CWE 457] (Inspector): out parameter Old_Val might be
main.adb:17:16: high: validity check [CWE 457] (Inspector): main.set_global.Old_Val is
main.adb:19:43: high: validity check [CWE 457] (Infer): `Old` is read without initializing it

-----
-----
```

GNAT SAS output directory located at /home/user/.gnatsas/home/user/adamant/doc/example_architecture
GNAT SAS run log saved in /home/user/.gnatsas/home/user/adamant/doc/example_architecture
GNAT SAS analysis text output saved in /home/user/.gnatsas/home/user/adamant/doc/example_architecture
GNAT SAS analysis CSV output saved in /home/user/.gnatsas/home/user/adamant/doc/example_architecture
GNAT SAS security report output saved in /home/user/.gnatsas/home/user/adamant/doc/example_architecture

From the output we can see that `redo analyze` was run in binary mode since this program is a `main.adb`. However, since the program only contains one file, only this single file is analyzed. From the results, we can see that a “high” warning was produced for line 8, indicating that our `out` parameter might not be initialized. This pinpoints a defect in our code which is the `if` conditional that only sets the `out` parameter if the input is less than 20. Otherwise, the `out` parameter is left uninitialized, which could potentially cause problems for the rest of the program.

In fact, when running the program, this bug is very evident:

```
> redo run
redo      doc/example_architecture/analyze_demo/run
redo      doc/example_architecture/analyze_demo/build/bin/Linux/main.elf
redo      doc/example_architecture/analyze_demo/build/obj/Linux/main.o
Set to 29, old value was -2147483648
```

The old value of `Outer_Var` is reported to be some random number that is not at all the value of 6 like we would expect.

By default, GNAT SAS is run with the currently set build target specified by the `TARGET` environment variable and runs in `fast` mode. This is the recommended mode for everyday running of GNAT SAS by developers. GNAT SAS can also be run in `deep` mode, ideal for overnight runs on a continuous integration server, by specifying `--mode=deep` in your build target’s `.gpr` file. A deep analysis target for Linux is provided in `redo/targets/gpr/linux_analyze.gpr` and can be used by running `export TARGET=Linux_Analyze` prior to running `redo analyze`.

Alternatively, the analysis mode can be controlled using the `REDO_ANALYZE_MODE` environment variable. If set, this variable will pass the `--mode=<VALUE>` parameter to the `gnatsas analyze` command. For example:

```
> export REDO_ANALYZE_MODE=deep
> redo analyze
```

This approach allows users to easily switch analysis modes without changing build targets. If `REDO_ANALYZE_MODE` is not set, no mode parameter is passed and GNAT SAS runs in its default `fast` mode.

For automated testing and continuous integration, the `redo analyze_all` command provides additional control over exit behavior through the `REDO_ANALYZE_ALL_FAIL_LEVEL` environment

variable. This variable determines the analysis warning level that causes the command to exit with a non-zero code:

- **HIGH** - Exit with failure only on high-severity warnings or errors (default)
- **MEDIUM** - Exit with failure on medium, high-severity warnings, or errors
- **LOW** - Exit with failure on low, medium, high-severity warnings, or errors
- **ERROR** - Exit with failure only on analysis errors (not warnings)

Example usage:

```
> export REDO_ANALYZE_ALL_FAIL_LEVEL=MEDIUM
> redo analyze_all # Will exit non-zero if any medium or high warnings found
```

The `redo analyze` command copies all source files to `$HOME/.gnatsas`, analyzes it, and produces results in the same directory. It is recommended that you save the results stored in `$HOME/.gnatsas` in a version control system so you can compare results of different static analysis runs over time. For more information on GNAT SAS, how to view and compare results between runs, and produce different style reports, see AdaCore's [documentation](#).

9.5 Code Metrics

This section presents different metrics that can be produced for code in Adamant. This includes raw Ada packages, components, and assemblies.

9.5.1 SLOC and Cyclomatic Complexity

To produce most code metrics, Adamant uses `gnatmetric`. This program provides logical SLOC (source lines of code), cyclomatic complexity, and other metrics for Adamant packages, objects, and binaries.

To see what metrics we can build in any directory we can simply run `redo what`. For this example we will use the `test_better` unit test directory created in Section 4.3.2. Some of the output has been removed for brevity.

```
> redo what # show "redo" commands that can be run in this directory
redo all
redo clean
redo clean_all
redo templates
redo publish
redo targets
redo prove
redo analyze
redo style
redo pretty
redo test_all
redo analyze_all
redo coverage_all
redo build/metric/Linux_Test/simple_package_tests-implementation-suite.adb.txt
redo build/metric/Linux_Test/simple_package_tests-implementation-suite.ads.txt
redo build/metric/Linux_Test/simple_package_tests-implementation.adb.txt
redo build/metric/Linux_Test/simple_package_tests-implementation.ads.txt
redo build/metric/Linux_Test/simple_package_tests-implementation.o.txt
redo build/metric/Linux_Test/simple_package_tests.adb.txt
redo build/metric/Linux_Test/simple_package_tests.ads.txt
```

```

redo build/metric/Linux_Test/test.adb.txt
redo build/metric/Linux_Test/test.elf.txt
redo build/metric/Linux_Test/test.o.txt
redo coverage
redo test

```

As can be seen, there are many rules to build files in the *build/metric* directory. Each of these rules uses *gnatmetric* to produce output for that file type. The most basic metric calculation is to calculate the metrics for a single Ada file. As an example, let's build the metrics for the unit test implementation package body, *simple_package_tests-implementation.adb*.

```
> redo build/metric/Linux_Test/simple_package_tests-implementation.adb.txt
```

Which will produce the report in a file called *build/metric/Linux_Test/simple_package_tests-implementation.adb.txt*.

simple_package_tests-implementation.adb.txt:

```

1 Sources analyzed:
2 /__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/simple_package_tests-
3
4 Command:
5 $ cat /tmp/tmpzgy4kxc9 | xargs gnatmetric -d /__w/adamant/adamant/doc/example_architecture/simple_
6
7 Line metrics summed over 1 units
8   all lines          : 52
9   code lines         : 29
10  comment lines      : 14
11  end-of-line comments : 0
12  comment percentage : 32.55
13  blank lines        : 9
14
15 Average lines in body: 6.25
16
17 Element metrics summed over 1 units
18   all statements     : 7
19   all declarations    : 15
20   logical SLOC        : 22
21
22 4 subprogram bodies in 1 units
23
24 Average cyclomatic complexity: 1.00
25
26 Coupling metrics:
27 =====

```

As you can see, many useful metrics are reported. The most important is the "logical SLOC", which is used to track the lines of code of a package, and "average cyclomatic complexity", which is used to measure source code complexity.

Note that the *gnatmetric* line shows the actual *gnatmetric* command used to generate the metrics. The line breaks are disabled on this line on purpose, since the command is quite large and would span many pages of this document.

We can also build metrics for an object or an executable. In either case, instead of reporting metrics for a single file, these commands will report the metrics for that object or executable and any dependencies that it has. In this way, building the metrics for an executable, such as an assembly binary, will provide metrics for the entire project. As an example, let's build the metrics for the entire unit test program. We can do this by building the metrics for the *test.elf*.

```
> redo build/metric/Linux_Test/test.elf.txt
```

Which produces the metrics for the executable in the output file *build/metric/Linux_Test/test.elf.txt*.

test.elf.txt:

```
1 Autocoded Metrics:  
2 Sources analyzed:  
3 /__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/build/src/simple_package/test_better3/test.elf  
4  
5 Command:  
6 $ cat /tmp/tmp3jp1qo36 | xargs gnatmetric -d /__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/build/src/simple_package/test_better3/test.elf  
7  
8 Line metrics summed over 4 units  
9   all lines          : 174  
10  code lines         : 87  
11  comment lines      : 61  
12  end-of-line comments : 0  
13  comment percentage  : 41.21  
14  blank lines        : 26  
15  
16 Average lines in body: 7.16  
17  
18 Element metrics summed over 4 units  
19   all statements     : 18  
20   all declarations    : 54  
21   logical SLOC        : 72  
22  
23 2 public types in 1 units  
24  including  
25    1 private types  
26  
27 2 type declarations in 1 units  
28  
29 5 public subprograms in 2 units  
30  
31 6 subprogram bodies in 2 units  
32  
33 Average cyclomatic complexity: 1.00  
34  
35 Coupling metrics:  
36 =====  
37  Unit Simple_Package_Tests (/__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/test.elf)  
38    tagged fan-out coupling  : 0  
39    hierarchy fan-out coupling: 0  
40    tagged fan-in coupling   : 0  
41    hierarchy fan-in coupling: 0  
42    control fan-out coupling : 0  
43    control fan-in coupling  : 0  
44    unit fan-out coupling    : 0  
45    unit fan-in coupling     : 0  
46  
47  Unit Simple_Package_Tests.Implementation.Suite (/__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/test.elf)  
48    control fan-out coupling : 0  
49    control fan-in coupling  : 0  
50    unit fan-out coupling    : 0  
51    unit fan-in coupling     : 0  
52  
53 Handcoded Metrics:  
54 Sources analyzed:
```

```

55 /__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/test.adb /__w/adamant/
56
57 Command:
58 $ cat /tmp/tmphhnm_26m | xargs gnatmetric -d /__w/adamant/adamant/doc/example_architecture/simple_
59
60 Line metrics summed over 17 units
61     all lines          : 849
62     code lines         : 511
63     comment lines      : 217
64     end-of-line comments : 5
65     comment percentage : 30.49
66     blank lines        : 121
67
68 Average lines in body: 6.48
69
70 Element metrics summed over 17 units
71     all statements      : 91
72     all declarations     : 486
73     logical SLOC        : 577
74
75 9 public types in 3 units
76 including
77     2 private types
78
79 9 type declarations in 3 units
80
81 42 public subprograms in 7 units
82
83 47 subprogram bodies in 8 units
84
85 Average cyclomatic complexity: 1.40
86
87 Coupling metrics:
88 =====
89     Unit Basic_Assertions (/__w/adamant/adamant/src/unit_test/smart_assert/basic_assertions.ads)
90         unit fan-out coupling      : 3
91         unit fan-in coupling      : 1
92
93     Unit Basic_Types (/__w/adamant/adamant/src/types/basic_types/basic_types.ads)
94         unit fan-out coupling      : 0
95         unit fan-in coupling      : 7
96
97     Unit Basic_Types.Representation (/__w/adamant/adamant/src/types/basic_types/basic_types-repre
98         control fan-out coupling   : 1
99         control fan-in coupling   : 1
100        unit fan-out coupling    : 2
101        unit fan-in coupling    : 2
102
103    Unit File_Logger (/__w/adamant/adamant/src/unit_test/file_logger/file_logger.ads)
104        tagged fan-out coupling   : 0
105        hierarchy fan-out coupling: 0
106        tagged fan-in coupling   : 0
107        hierarchy fan-in coupling: 0
108        control fan-out coupling : 1
109        control fan-in coupling  : 0
110        unit fan-out coupling   : 2
111        unit fan-in coupling   : 0
112
113    Unit Serializer (/__w/adamant/adamant/src/core/serializer/serializer.ads)
114        control fan-out coupling : 0
115        control fan-in coupling  : 1

```

```

116     unit fan-out coupling      : 1
117     unit fan-in coupling      : 1
118
119 Unit Simple_Package (/__w/adamant/adamant/doc/example_architecture/simple_package/simple_package.ads)
120     control fan-out coupling  : 0
121     control fan-in coupling   : 1
122     unit fan-out coupling    : 0
123     unit fan-in coupling    : 1
124
125 Unit Simple_Package_Tests.Implementation (/__w/adamant/adamant/doc/example_architecture/simple_package_tests.implementation.ads)
126     tagged fan-out coupling   : 0
127     hierarchy fan-out coupling: 0
128     tagged fan-in coupling   : 0
129     hierarchy fan-in coupling: 0
130     control fan-out coupling : 3
131     control fan-in coupling  : 0
132     unit fan-out coupling   : 5
133     unit fan-in coupling   : 0
134
135 Unit Smart_Assert (/__w/adamant/adamant/src/unit_test/smart_assert/smart_assert.ads)
136     control fan-out coupling  : 1
137     control fan-in coupling   : 1
138     unit fan-out coupling    : 2
139     unit fan-in coupling    : 2
140
141 Unit String_Util (/__w/adamant/adamant/src/unit_test/string_util/string_util.ads)
142     control fan-out coupling  : 1
143     control fan-in coupling   : 3
144     unit fan-out coupling    : 2
145     unit fan-in coupling    : 3
146
147 Unit Test (/__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/test.adb)
148     control fan-out coupling  : 0
149     control fan-in coupling   : 0
150     unit fan-out coupling    : 0
151     unit fan-in coupling    : 0
152
153 Total Metrics:
154 Sources analyzed:
155 /__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/test.adb /__w/adamant/adamant/doc/example_architecture/simple_package/simple_package.ads
156
157 Command:
158 $ cat /tmp/tmpprz92xas8 | xargs gnatmetric -d /__w/adamant/adamant/doc/example_architecture/simple_package/simple_package.ads
159
160 Line metrics summed over 21 units
161     all lines          : 1023
162     code lines         : 598
163     comment lines      : 278
164     end-of-line comments: 5
165     comment percentage : 32.30
166     blank lines        : 147
167
168 Average lines in body: 6.56
169
170 Element metrics summed over 21 units
171     all statements     : 109
172     all declarations   : 540
173     logical SLOC       : 649
174
175 11 public types in 4 units
176 including

```

```

177     3 private types
178
179     11 type declarations in 4 units
180
181     47 public subprograms in 9 units
182
183     53 subprogram bodies in 10 units
184
185     Average cyclomatic complexity: 1.35
186
187     Coupling metrics:
188 =====
189     Unit Basic_Assertions (/__w/adamant/adamant/src/unit_test/smart_assert/basic_assertions.ads)
190         unit fan-out coupling      : 3
191         unit fan-in coupling      : 1
192
193     Unit Basic_Types (/__w/adamant/adamant/src/types/basic_types/basic_types.ads)
194         unit fan-out coupling      : 0
195         unit fan-in coupling      : 8
196
197     Unit Basic_Types.Representation (/__w/adamant/adamant/src/types/basic_types/basic_types-repre
198         control fan-out coupling   : 1
199         control fan-in coupling   : 1
200         unit fan-out coupling     : 2
201         unit fan-in coupling     : 2
202
203     Unit File_Logger (/__w/adamant/adamant/src/unit_test/file_logger/file_logger.ads)
204         tagged fan-out coupling    : 0
205         hierarchy fan-out coupling: 0
206         tagged fan-in coupling    : 2
207         hierarchy fan-in coupling: 2
208         control fan-out coupling  : 1
209         control fan-in coupling  : 4
210         unit fan-out coupling    : 2
211         unit fan-in coupling    : 4
212
213     Unit Serializer (/__w/adamant/adamant/src/core/serializer/serializer.ads)
214         control fan-out coupling  : 0
215         control fan-in coupling  : 1
216         unit fan-out coupling    : 1
217         unit fan-in coupling    : 1
218
219     Unit Simple_Package (/__w/adamant/adamant/doc/example_architecture/simple_package/simple_pac
220         control fan-out coupling  : 0
221         control fan-in coupling  : 1
222         unit fan-out coupling    : 0
223         unit fan-in coupling    : 1
224
225     Unit Simple_Package_Tests (/__w/adamant/adamant/doc/example_architecture/simple_package/test_
226         tagged fan-out coupling   : 1
227         hierarchy fan-out coupling: 1
228         tagged fan-in coupling   : 1
229         hierarchy fan-in coupling: 0
230         control fan-out coupling : 2
231         control fan-in coupling : 3
232         unit fan-out coupling   : 3
233         unit fan-in coupling   : 3
234
235     Unit Simple_Package_Tests.Implementation (/__w/adamant/adamant/doc/example_architecture/simple_
236         tagged fan-out coupling   : 2
237         hierarchy fan-out coupling: 1

```

```

238     tagged fan-in coupling : 0
239     hierarchy fan-in coupling : 0
240     control fan-out coupling : 5
241     control fan-in coupling : 2
242     unit fan-out coupling : 7
243     unit fan-in coupling : 2
244
245 Unit Simple_Package_Tests.Implementation.Suite (/__w/adamant/adamant/doc/example_architecture)
246     control fan-out coupling : 3
247     control fan-in coupling : 1
248     unit fan-out coupling : 3
249     unit fan-in coupling : 1
250
251 Unit Smart_Assert (/__w/adamant/adamant/src/unit_test/smart_assert/smart_assert.ads)
252     control fan-out coupling : 1
253     control fan-in coupling : 1
254     unit fan-out coupling : 2
255     unit fan-in coupling : 2
256
257 Unit String_Util (/__w/adamant/adamant/src/unit_test/string_util/string_util.ads)
258     control fan-out coupling : 1
259     control fan-in coupling : 4
260     unit fan-out coupling : 2
261     unit fan-in coupling : 4
262
263 Unit Test (/__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/test.adb)
264     control fan-out coupling : 4
265     control fan-in coupling : 0
266     unit fan-out coupling : 4
267     unit fan-in coupling : 0

```

Adamant splits the output into three sections, the metrics of the generated code (ie. any code found in a *build/src* directory), the metrics of hand-code, and the total metrics of both generated and hand-coded combined. In this way, you can compare the number of lines of code that are generated vs hand-coded in your program. In Adamant, it is not uncommon for an assembly binary to be made up of 75% autocoded SLOC.

9.5.2 YAML SLOC

Since such a large part of designing software in Adamant is about writing YAML models, it is useful to understand how many lines of model code have been written for a project. To determine the YAML SLOC first build the *main.elf* for your project, and then run:

```
> redo yaml_sloc
```

from your project main directory. The tool will search for all YAML files in the path and total up the lines of code, both hand-written and autocoded. It is important to build your *main.elf* first if you want an accurate count of the autocoded SLOC.

Note that a SLOC of YAML is considered to be a single key-value pair. Raw list entries of strings, numbers, etc. are not considered single lines for the purpose of counting unless the list entry is itself a key-value pair. The search is recursive, so even very deeply nested key-value pairs will be counted.

Example output of `redo yaml_sloc` might look like:

```

1 redo      doc/example_architecture/yaml_sloc/yaml_sloc
2 Note, please run this after building your project main.elf to accurately calculate autocoded yaml
3 Fetching all YAML in build path...
4 Calculating the YAML SLOC for 494 files...
5 Hand-coded SLOC: 7590

```

```
6 Auto-coded SLOC: 0
7 Total SLOC: 7590
```

9.5.3 Build Artifacts

Adamant can provide a text file with any compiled executable that provides useful metadata about how the binary was constructed. This text file is commonly referred to as a *build artifact*. By default, this feature is disabled because it can add some time to the compilation process. However, it can be turned on by setting the correct environment variable.

```
> export CREATE_BUILD_ARTIFACTS=1
```

For this example we will use the *test_better* unit test directory created in Section 4.3.2. Within this directory, we just need to compile the unit test as we would normally do.

```
> redo build/bin/Linux_Test/test.elf
```

or by simply running *redo test*. Adamant will now produce an extra build artifact file next to the executable, *build/bin/Linux_Test/test.elf.txt*. Below are the contents of this file:

test.elf.txt:

```
1 Basic Info:
2 Binary meta data for file: /__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/build/bin/Linux_Test/test.elf
3 Main object file: /__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/build/bin/Linux_Test/test.elf
4 Binary meta data for redo temp file: /tmp/redo-211148/211148/temp/2219816B21BC0AF9B63F8305A255E...
5 Report generation time: 2025-07-15 21:50
6 File time stamp: 2025-07-15 21:50
7 File type:
8 File MD5 hash: 885963a748a3d2560d6e8b8208c0c9fb
9 File size (bytes): 2799952
10
11 Git information:
12 Adamant git repository: /__w/adamant/adamant
13 Adamant git commit: 6f5e96c2a69d6023673aed2987fb1227739fdfe2
14 Adamant git changed files: []
15 Adamant git untracked files: []
16
17 Environment:
18 TARGET = Linux_Test
19 PYTHONPATH = redo:::/__w/adamant/adamant/gen:/__w/adamant/adamant/gnd:/__w/adamant/adamant/redo...
20 ADAMANT_DIR = /__w/adamant/adamant
21 PATH = /github/home/.local/share/alire/toolchains/gprbuild_22.0.1_24dfc1b5/bin:/github/home/.lo...
22 SCHEMAPATH = /__w/adamant/adamant/gen/schemas
23 TEMPLATEPATH = /__w/adamant/adamant/gen/templates
24 BUILD_PATH = (not set)
25 BUILD_ROOTS = (not set)
26 EXTRA_BUILD_PATH = /__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/...
27 EXTRA_BUILD_ROOTS = (not set)
28 REMOVE_BUILD_PATH = (not set)
29 COMPUTED_BUILD_ROOTS = /__w/adamant/adamant
30 COMPUTED_BUILD_PATH = /__w/adamant/adamant/gen/test/enums:/__w/adamant/adamant/src/data_struct...
31
32 Build Target:
33 Target specified: Linux_Test
34 Target instance: <targets.linux.Linux_Test object at 0x7ff7701ed610>
35 Target file: /__w/adamant/adamant/redo/targets/linux.py
36 Target name: Linux_Test
37
```

```

38 GNAT Info:
39 Source in build:
40 ls -l /tmp/tmp.uNAdFFlwrV.adamant/211148/src
41 total 0
42
43
44 Objects in build:
45 ls -l /tmp/tmp.uNAdFFlwrV.adamant/211148/obj
46 total 0
47
48
49 Dependencies for main object /__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/build/src/simple_package_tests.adb
50 /__w/adamant/adamant/src/unit_test/file_logger/file_logger.ads
51 /__w/adamant/adamant/redo/rules/build_object.py
52 /__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/build/src/simple_package_tests.adb
53 /__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/test.adb
54 /__w/adamant/adamant/redo/targets/linux.py
55 /__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/build/src/simple_package_tests.adb
56 /__w/adamant/adamant/doc/example_architecture/simple_package/test_better3/simple_package_tests.adb
57

```

Note that line breaks have been disabled on this output to minimize the length of the file in this document.

Lots of useful information has been provided in this file including the settings of important environment variables, the build path, the current git hash, file dependencies, and more.

To turn the build artifact feature off again, simply unset the variable:

```
> export CREATE_BUILD_ARTIFACTS=
```

9.6 Creating a SPARK Package

The SPARK programming language is a formally analyzable subset of Ada. The language comes with a toolset that brings mathematics-based confidence to software verification. SPARK packages can be mathematically proven to be free of runtime errors and even meet defined specifications prior to compilation of the code itself. For more information on SPARK see [this tutorial](#).

The use of SPARK is encouraged for the most safety critical code of a project. Code written in SPARK can be verified to be free of runtime errors, and thus can be compiled without the standard Ada runtime checks. This makes verified SPARK code more efficient than the Ada equivalent. SPARK code can also be statically verified to ensure it meets functional requirements, lessening the need for unit testing.

The Adamant build system directly supports the verification and integration of SPARK packages into components and assemblies. In this section we will construct a simple SPARK package and show how to statically analyze it using Adamant. Let's start by making a directory for our package:

```
> mkdir spark_package
> cd spark_package
```

Next, we create two files, an Ada specification file called *spark_package.ads* and an Ada implementation file called *spark_package.adb*. Below is shown *spark_package.ads*:

```

1 package Spark_Package with
2   SPARK_Mode => On
3   is
```

```

4
5 procedure Increment (X : in out Integer)
6   with Global => null,           -- Increment does not read or write any
7       -- global variables.
8     Depends => (X => X),      -- The value of the parameter X after
9       -- the call depends only on the (previous) value of X.
10    Pre => X < Integer'Last,   -- Increment is only allowed to be called
11       -- if the value of X prior to the call is less than Integer'Last
12    Post => X = X'Old + 1;     -- The value of X after a call is one
13       -- greater than its value before the call.
14
15 end Spark_Package;

```

and *spark_package.adb*:

```

1 package body Spark_Package with
2   SPARK_Mode => On
3 is
4
5   procedure Increment (X : in out Integer) is
6   begin
7     X := @ + 1;
8   end Increment;
9
10 end Spark_Package;

```

We can see that the package contains a single procedure which safely increments an integer without a fear of overflow. We can easily compile the package to make sure there are no syntax errors:

```

> redo
redo all
redo build/obj/Linux/spark_package.o

```

Because this is SPARK code, we can also statically verify it using GNATprove to ensure that there are no runtime errors. GNATprove is built into the Adamant build system. To run GNATprove on all SPARK code within a directory we can run `redo prove` which produces the following output:

```

redo      doc/example_architecture/spark_package_no_config/prove
redo  Analyzing:
/_w/adamant/adamant/doc/example_architecture/spark_package_no_config/spark_package.adb
/_w/adamant/adamant/doc/example_architecture/spark_package_no_config/spark_package.ads
No GNATprove configuration was found at /_w/adamant/adamant/doc/example_architecture/
Running GNATprove with switches: --level=2 --mode=gold
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
Summary logged in /_w/adamant/adamant/doc/example_architecture/spark_package_no_config
GNATprove command output saved in /_w/adamant/adamant/doc/example_architecture/spark_

```

We can see that GNATprove was run with a mode of gold and produced no errors, meaning that this code has been proven free of runtime errors and also functionally correct. By default, `redo prove` runs GNATprove with a level of 2 and a mode of gold. For more information on GNATprove switches see [this documentation](#).

These defaults can be changed by including a YAML file which alters the configuration for this directory. To do this we can add the file *all.prove.yaml*:

```

1  ---
2  description: This is the GNATprove configuration for the Spark_Package.
3  level: 1
4  mode: "silver"

```

We provide a description as well as specify the new level and mode to use when analyzing any source code found within this directory. Running `redo prove` again with this file present, we can see the new settings have been used.

```

redo      doc/example_architecture/spark_package/prove
redo  Analyzing:
/_w/adamant/adamant/doc/example_architecture/spark_package/spark_package.adb
/_w/adamant/adamant/doc/example_architecture/spark_package/spark_package.ads
Using GNATprove configuration found in /_w/adamant/adamant/doc/example_architecture/sp
Running GNATprove with switches: --level=1 --mode=silver
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
Summary logged in /_w/adamant/adamant/doc/example_architecture/spark_package/build/prov
GNATprove command output saved in /_w/adamant/adamant/doc/example_architecture/spark_p

```

The `redo prove` command can be run from any directory that contains Ada or SPARK source code. Code without the `SPARK_Mode =>` aspect will be ignored by GNATprove and not analyzed. The results of the analysis are printed to the terminal and also saved in `build/prove/prove.txt`. Packages written in SPARK can be included in other SPARK or plain Ada packages using the usual with syntax.

9.7 The Last Chance Handler

In most embedded runtimes for Ada, including the Ravenscar small footprint (SFP) which is commonly used with Adamant, exception handling is limited. In particular, exceptions can only be handled in their local scope and will not be propagated. If an exception is thrown during runtime, due to a Constraint Error, Assertion Error, etc., and is not locally handled, then a special procedure called the `Last_Chance_Handler` is called. The `Last_Chance_Handler` should be defined by the developer to handle error reporting and possible recovery of the system, usually via system reboot.

Below is a simple example of the `Last_Chance_Handler` package body. It prints out the error message and then sleeps indefinitely:

`last_chance_handler.adb:`

```

1  -- Use the low level GNAT.IO package since the system is in a precarious state.
2  with GNAT.IO;
3  with System.Storage_Elements;
4
5  package body Last_Chance_Handler is
6
7      procedure Last_Chance_Handler (Msg : System.Address; Line : Integer) is
8          use System.Storage_Elements;
9          function Peek (Addr : System.Address) return Character is
10             C : Character with
11                 Address => Addr;
12             begin
13                 return C;
14             end Peek;
15             A : System.Address := Msg;
16             begin
17                 GNAT.IO.Put ("LCH called => ");

```

```

18     while Peek (A) /= Ascii.Nul loop
19         GNAT.IO.Put (Peek (A));
20         A := @ + 1;
21     end loop;
22     GNAT.IO.Put (" : ");
23     GNAT.IO.Put (Line); -- avoid the secondary stack for Line'Image
24     GNAT.IO.New_Line;
25
26     -- Spin indefinitely, don't use sleep
27     -- as that can cause a fault.
28     loop
29         null;
30     end loop;
31 end Last_Chance_Handler;
32
33 end Last_Chance_Handler;

```

This package should be modified to handle an error on your system and then added in your program's main .adb file.

For more information see the AdaCore documentation [here](#).

9.8 Creating a Register Map

Often in embedded systems, the software interacts with the hardware through an interface of predefined registers and shared memory regions. These registers and regions usually exist at static addresses that do not change during runtime. In Adamant, we refer to this as a *register map*.

Adamant provides a generator tailored to creating register maps of packed types (*packed records* or *packed arrays*) that are laid out at static address. The generated register map guarantees that none of the items overlap. To create a register map we need to create a model file in the form *name.register_map.yaml* where *name* describes the specific register map. For this example we have two registers that our software needs to interact with. Packed records for these registers have already been defined elsewhere.

system_registers.register_map.yaml:

```

1  ---
2  description: This is the register map for an example system. Both registers and
3  ← register sets (a packed record holding more than one register) can be
4  ← defined and assigned addresses here.
5  items:
6      - address: 0x4000_0004
7          name: First_Register
8          description: An important register.
9          type: Packed_U32.Register_T
10     - address: 0x4000_0016
11         name: Second_Register
12         description: An equally important register.
13         type: Packed_Poly_32_Type.Register_T

```

As can be seen, we provide an address, name, type and optional description for each register. Each register must be a packed type. The type can be a composite type that holds many registers internally, such as a packed array of registers or a packed record containing a register set for a specific hardware interface. Creating this model file provides us with many new build targets which can be seen by running `redo what`.

```
> redo what
redo  what
```

```

redo all
redo clean
redo clean_all
redo templates
redo publish
redo targets
redo prove
redo analyze
redo style
redo pretty
redo test_all
redo analyze_all
redo coverage_all
redo build/html/system_registers.html
redo build/metric/Linux/system_registers.ads.txt
redo build/obj/Linux/system_registers.o
redo build/pdf/system_registers.pdf
redo build/src/system_registers.ads
redo build/tex/system_registers.tex

```

To create the generated Ada package we can run `redo build/src/system_registers.ads` which produces the output.

```

redo      doc/example_architecture/register_map/build/src/system_registers.ads
redo      src/types/packed_types/build/src/packed_u32.ads
redo      src/types/packed_types/build/src/packed_poly_32_type.ads
redo      src/types/packed_types/build/src/packed_u32.adb
redo      src/types/packed_types/build/src/packed_poly_32_type.adb
redo  Analyzing:
/_w/adamant/adamant/doc/example_architecture/register_map/build/src/temp/system_registers.ads
No GNATprove configuration was found at /_w/adamant/adamant/doc/example_architecture/register_map/build/src/temp/system_registers.ads
Running GNATprove with switches: --level=2 --mode=gold
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
Summary logged in /_w/adamant/adamant/doc/example_architecture/register_map/build/prover.log
GNATprove command output saved in /_w/adamant/adamant/doc/example_architecture/register_map/build/prover.log

```

This is not the normal output we expect to see when generating an Ada package. However, the register map produced is special in that it is written mostly in SPARK, the formally provable subset of Ada. GNATprove is then automatically run to statically check the output, which is shown above.

The primary reason for this is that SPARK disallows the definition of bit-constrained types with address clauses. A bit-constrained type is a type that for some bit value can become invalid, causing a `Constraint_Error` to be thrown at runtime. While bit-constrained types are useful for catching bugs in most Ada code, they can be troublesome when interfacing with volatile hardware interfaces, where the compiler is not in control of what might be written to these types. In such cases, a hardware interface may write an invalid bit value to a type within a register map. When this type is later read by the Ada program, a `Constraint_Error` will be thrown. To prevent this, Adamant does not allow bit-constrained types to be defined in register maps. To enforce this it uses the SPARK programming language. If bit-constrained types are used in the register map than an error will be produced and the specification will not be generated.

The generated specification files is shown below:

nonvolatile_store.ads:

```

1 -----  

2 -- System_Registers Register_Map Spec  

3 --  

4 -- Generated from system_registers.register_map.yaml on 2025-07-15 21:47.  

5 -----  

6  

7 -- Standard Includes:  

8 with System.Storage_Elements; use System.Storage_Elements;  

9 with Interfaces; use Interfaces;  

10  

11 -- Item Includes:  

12 with Packed_U32;  

13 with Packed_Poly_32_Type;  

14  

15 -- This is the register map for an example system. Both registers and register  

16 -- sets (a packed record holding more than one register) can be defined and  

17 -- assigned addresses here.  

18 package System_Registers  

19     -- We apply SPARK to this package to help us discover bit-constrained types  

20     -- in  

21     -- the register map. These types should be avoided in register maps at all  

22     -- costs since  

23     -- simply reading a type from the register map that got corrupted to a value  

24     -- that  

25     -- is out of range could be a mission ending failure.  

26     --  

27     -- If a bit-constrained type is detected by SPARK the following "high"  

28     -- warning will  

29     -- be issued as part of the analysis:  

30     --  

31     -- object with constraints on bit representation is unsuitable for  

32     -- aliasing  

33     -- via address clause  

34     --  

35     -- Any line with this issue present should be fixed by modifying the  

36     -- underlying type  

37     -- to NOT be bit-constrained.  

38     with SPARK_Mode => On  

39     is  

40  

41     --  

42     -- Register map constants:  

43     --  

44  

45     -- Memory map num items ():  

46     Num_Items : constant Natural := 2;  

47  

48     --  

49     -- Register map items:  

50     --  

51  

52     -- 0x40000004 - 40000007: First_Register  

53     -- An important register.  

54     First_Register_Offset : constant Unsigned_32 := 16#40000004#; -- 0x40000004  

55     First_Register_Size : constant Unsigned_32 := Unsigned_32  

56     -- (Packed_U32.Size_In_Bytes); -- 4 bytes  

57     First_Register_End : constant Unsigned_32 := First_Register_Offset +  

58     -- First_Register_Size - 1; -- 0x40000007  

59     First_Register_Address : constant System.Address := To_Address  

60     -- (Integer_Address (First_Register_Offset));  

61     pragma Warnings (Off, "writing to ""First_Register"" is assumed to have no  

62     -- effects on other non-volatile objects");

```

```

53  pragma Warnings (Off, "assuming no concurrent accesses to non-atomic object
54    => ""First_Register"");
55  First_Register : aliased Packed_U32.Register_T
56    with Volatile_Full_Access => True, Import, Convention => Ada, Address =>
57      First_Register_Address;
58  pragma Warnings (On, "assuming no concurrent accesses to non-atomic object
59    => ""First_Register""");
60  pragma Warnings (On, "writing to ""First_Register"" is assumed to have no
61    => effects on other non-volatile objects");
62
63  -- 0x40000016 - 40000019: Second_Register
64  -- An equally important register.
65  Second_Register_Offset : constant Unsigned_32 := 16#40000016#; -- 0x40000016
66  Second_Register_Size : constant Unsigned_32 := Unsigned_32
67    => (Packed_Poly_32_Type.Size_In_Bytes); -- 4 bytes
68  Second_Register_End : constant Unsigned_32 := Second_Register_Offset +
69    Second_Register_Size - 1; -- 0x40000019
70  Second_Register_Address : constant System.Address := To_Address
71    (Integer_Address (Second_Register_Offset));
72  pragma Warnings (Off, "writing to ""Second_Register"" is assumed to have no
73    => effects on other non-volatile objects");
74  pragma Warnings (Off, "assuming no concurrent accesses to non-atomic object
75    => ""Second_Register""");
76  Second_Register : aliased Packed_Poly_32_Type.Register_T
77    with Volatile_Full_Access => True, Import, Convention => Ada, Address =>
78      Second_Register_Address;
79  pragma Warnings (On, "assuming no concurrent accesses to non-atomic object
80    => ""Second_Register""");
81  pragma Warnings (On, "writing to ""Second_Register"" is assumed to have no
82    => effects on other non-volatile objects");
83
84  --
85  -- Register map access types:
86  --
87
88  package Accesses
89    -- Note that general access types are not allowed in SPARK,
90    -- but we may want to use these in Ada code, so we provide them here.
91    with SPARK_Mode => Off
92  is
93    First_Register_Access : constant Packed_U32.Register_T_Access :=
94      First_Register'Access;
95    Second_Register_Access : constant Packed_Poly_32_Type.Register_T_Access :=
96      Second_Register'Access;
97  end Accesses;
98
99  --
100 -- Register map compile-time checks. These checks help to ensure that there
101   -- is not a bug in the autocoder.
102   --
103
104  -- 0x40000004 - 40000007: First_Register Checks
105  pragma Compile_Time_Error (First_Register_Offset /= 16#40000004#, "Unexpected
106    -- autocoder error. Item start address not as expected.");
107  pragma Compile_Time_Error (First_Register_Size /= 4, "Unexpected autocoder
108    -- error. Item size not as expected.");
109  pragma Compile_Time_Error (First_Register_End /= 16#40000007#, "Unexpected
110    -- autocoder error. Item end address not as expected.");
111  pragma Compile_Time_Error (First_Register_Offset >= First_Register_End,
112    -- "Unexpected autocoder error. Item end address less than start address.");
113
114  -- 0x40000016 - 40000019: Second_Register Checks

```

```

95  pragma Compile_Time_Error (Second_Register_Offset /= 16#40000016#,
96    => "Unexpected autocoder error. Item start address not as expected.");
97  pragma Compile_Time_Error (Second_Register_Size /= 4, "Unexpected autocoder
98    error. Item size not as expected.");
99  pragma Compile_Time_Error (Second_Register_End /= 16#40000019#, "Unexpected
100   autocoder error. Item end address not as expected.");
101  pragma Compile_Time_Error (Second_Register_Offset >= Second_Register_End,
102    => "Unexpected autocoder error. Item end address less than start address.");
103  pragma Compile_Time_Error (Second_Register_Offset <= First_Register_End,
104    => "Unexpected autocoder error. Item overlap found.");
105  pragma Compile_Time_Error (Second_Register_Offset <= First_Register_Offset,
106    => "Unexpected autocoder error. Item overlap found.");
107
108 end System_Registers;

```

In the package, each register item is declared at the provided address. Following, is a declaration of access types that refer to the registers declared in the previous section. These types of accesses are not allowed within SPARK, so they are declared with SPARK_Mode => Off. The final section specifies a myriad of `Compile_Time_Error` calls which employs the compiler to check the work of the generator to make sure that no registers overlap or exceed the region bounds. These global variables may now be safely used within the system, usually provided to components as part of their initialization.

We can also build PDF and HTML documentation to view the register map definition in a human readable manner. To build HTML documentation run:

```
> redo build/html/system_registers.html
```

which when opened with your favorite web browser looks something like this:

System_Registers Register Map

Description: This is the register map for an example system. Both registers and register sets (a packed record holding more than one register) can be defined and assigned addresses here.

Address	Name	Type	Size (Bytes)	Description
0x40000004 ~ 0x40000007	first_Register	Packed_U32_Register_T	4	An important register.
0x40000016 ~ 0x4000001D	second_Register	Packed_Poly_Type_Register_T	8	An equally important register.

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/register_map/system_registers.register_map.yaml on 2021-04-23 22:07.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

9.9 Creating a Memory Map

A *memory map* in Adamant is similar to a *register map*, discussed in Section 9.8, however, instead of defining the address for each item individually, items are laid out contiguously (packed tightly) at a given starting address. The common use case for a *memory map* is when defining a set of data that needs to be saved in nonvolatile memory located at a specific hardware address.

Often in embedded systems, data needs to be stored persistently, across reboots. This data is usually stored in some kind of nonvolatile memory, such as MRAM, commonly without aid of a file system. In such a case, important variables are stored at predetermined static addresses. In Adamant, we refer to this as a *memory map*.

Adamant provides a generator tailored to creating memory maps of packed types (*packed records* or *packed arrays*) that are contiguously laid out at a given memory address. These memory maps guarantee that none of the items overlap, and that the items do not overrun the memory region

reserved for the memory map. To create a memory map we need to create a model file in the form *name.memory_map.yaml* where *name* describes the specific memory map. For this example we need to store some important variables in a nonvolatile store. We define these variables in a file called *nonvolatile_store.memory_map.yaml*:

nonvolatile_store.memory_map.yaml:

```

1  ---
2  description: This is the memory layout for an example set of variables for a
3    ↳ mission.
4  start_address: 0x0600000
5  length: 2097152 # 2MB
6  items:
7    - name: Time
8      description: An important timestamp.
9      type: Sys_Time.T
10   - name: Counter
11     description: An important counter.
12     type: Packed_U32.T
13   - name: Another_Counter
14     description: An equally important counter.
15     type: Packed_U16.T

```

As can be seen, we provide a *start_address* and a *length* for the memory region devoted to this memory map. Following, we specify each item to include in the store by denoting its *name*, *type*, and an optional *description*. Creating this model file provides us with many new build targets which can be seen by running *redo what*.

```

> redo what
redo what
redo all
redo clean
redo clean_all
redo templates
redo publish
redo targets
redo prove
redo analyze
redo style
redo pretty
redo test_all
redo analyze_all
redo coverage_all
redo build/html/nonvolatile_store.html
redo build/metric/Linux/nonvolatile_store.ads.txt
redo build/obj/Linux/nonvolatile_store.o
redo build/pdf/nonvolatile_store.pdf
redo build/src/nonvolatile_store.ads
redo build/tex/nonvolatile_store.tex

```

To create the generated Ada package we can run *redo build/src/nonvolatile_store.ads* which produces the output.

```

redo      doc/example_architecture/memory_map/build/src/nonvolatile_store.ads
redo  Analyzing:
/_w/adamant/adamant/doc/example_architecture/memory_map/build/src/temp/nonvolatile_sto
No GNATprove configuration was found at /_w/adamant/adamant/doc/example_architecture/r
Running GNATprove with switches: --level=2 --mode=gold

```

```

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
Summary logged in /__w/adamant/adamant/doc/example_architecture/memory_map/build/prove/
GNATprove command output saved in /__w/adamant/adamant/doc/example_architecture/memory_

```

This is not the normal output we expect to see when generating an Ada package. However, the memory map produced is special in that it is written mostly in SPARK, the formally provable subset of Ada. GNATprove is then automatically run to statically check the output, which is shown above.

The primary reason for this is that SPARK disallows the definition of bit-constrained types with address clauses. A bit-constrained type is a type that for some bit value can become invalid, causing a `Constraint_Error` to be thrown at runtime. While bit-constrained types are useful for catching bugs in most Ada code, they can be troublesome when interfacing with volatile hardware interfaces, where the compiler is not in control of what might be written to these types. In such cases, a hardware interface may write an invalid bit value to a type within a memory map. When this type is later read by the Ada program, a `Constraint_Error` will be thrown. To prevent this, Adamant does not allow bit-constrained types to be defined in memory maps. To enforce this it uses the SPARK programming language. If bit-constrained types are used in the register map than an error will be produced and the specification will not be generated.

The generated specification files is shown below:

nonvolatile_store.ads:

```

1  -----
2  -- Nonvolatile_Store Memory_Map Spec
3  --
4  -- Generated from nonvolatile_store.memory_map.yaml on 2025-07-15 21:47.
5  -----
6
7  -- Standard Includes:
8  with System.Storage_Elements; use System.Storage_Elements;
9  with Interfaces; use Interfaces;
10
11 -- Item Includes:
12 with Sys_Time;
13 with Packed_U32;
14 with Packed_U16;
15
16 -- This is the memory layout for an example set of variables for a mission.
17 package Nonvolatile_Store
18     -- We apply SPARK to this package to help us discover bit-constrained types
19     -- in
20     -- the memory map. These types should be avoided in memory maps at all costs
21     -- since
22     -- simply reading a type from the memory map that got corrupted to a value
23     -- that
24     -- is out of range could be a mission ending failure.
25     --
26     -- If a bit-constrained type is detected by SPARK the following "high"
27     -- warning will
28     -- be issued as part of the analysis:
29     --
30     -- object with constraints on bit representation is unsuitable for
31     -- aliasing
32     -- via address clause
33     --
34     -- Any line with this issue present should be fixed by modifying the
35     -- underlying type
36     -- to NOT be bit-constrained.

```

```

31      with SPARK_Mode => On
32  is
33
34      --
35      -- Memory map constants:
36      --
37
38      -- Memory map start (0x00600000):
39      Start_Offset : constant Unsigned_32 := 16#00600000#; -- 0x00600000
40      Start_Address : constant System.Address := To_Address (Integer_Address
41        => (Start_Offset));
42
43      -- Memory map size (2097152):
44      Length : constant Unsigned_32 := 2097152; -- 0x00200000 bytes
45      Num_Items : constant Natural := 3;
46
47      -- Memory map end address (0x007FFFFF):
48      Last_Offset : constant Unsigned_32 := Start_Offset + Length - 1;
49      Last_Address : constant System.Address := To_Address (Integer_Address
50        => (Last_Offset)); -- 0x007FFFFF
51
52      -- Memory map last used address (0x0060000D):
53      Last_Used_Offset : constant Unsigned_32 := 16#0060000D#; -- 0x0060000D
54      Last_Used_Address : constant System.Address := To_Address (Integer_Address
55        => (Last_Used_Offset));
56
57      -- Calculate the space available:
58      Used_Bytes : constant Unsigned_32 := Last_Used_Offset - Start_Offset + 1; --
59        -- 14 bytes
60      Unused_Bytes : constant Unsigned_32 := Last_Offset - Last_Used_Offset; --
61        -- 2097138 bytes
62
63      --
64      -- Memory map items:
65
66
67      -- 0x00600000 - 00600007: Time
68      -- An important timestamp.
69      Time_Offset : constant Unsigned_32 := 16#00600000#; -- 0x00600000
70      Time_Size : constant Unsigned_32 := Unsigned_32 (Sys_Time.Size_In_Bytes); --
71        -- 8 bytes
72      Time_End : constant Unsigned_32 := Time_Offset + Time_Size - 1; -- 0x00600007
73      Time_Address : constant System.Address := To_Address (Integer_Address
74        => (Time_Offset));
75
76      pragma Warnings (Off, "indirect writes to ""Time"" through a potential alias
77        -- are ignored");
78      pragma Warnings (Off, "writing to ""Time"" is assumed to have no effects on
79        -- other non-volatile objects");
80      pragma Warnings (Off, "assuming no concurrent accesses to non-atomic object
81        -- ""Time""");
82      Time : aliased Sys_Time.T
83          with Import, Convention => Ada, Address => Time_Address;
84      pragma Warnings (On, "assuming no concurrent accesses to non-atomic object
85        -- ""Time""");
86      pragma Warnings (On, "writing to ""Time"" is assumed to have no effects on
87        -- other non-volatile objects");
88      pragma Warnings (On, "indirect writes to ""Time"" through a potential alias
89        -- are ignored");
90
91      -- 0x00600008 - 0060000B: Counter

```

```

79      -- An important counter.
80      Counter_Offset : constant Unsigned_32 := 16#00600008#; -- 0x00600008
81      Counter_Size : constant Unsigned_32 := Unsigned_32
82          => (Packed_U32.Size_In_Bytes); -- 4 bytes
83      Counter_End : constant Unsigned_32 := Counter_Offset + Counter_Size - 1; --
84          => 0x0060000B
85      Counter_Address : constant System.Address := To_Address (Integer_Address
86          => (Counter_Offset));
87
88      pragma Warnings (Off, "indirect writes to ""Counter"" through a potential
89          => alias are ignored");
90      pragma Warnings (Off, "writing to ""Counter"" is assumed to have no effects
91          => on other non-volatile objects");
92      pragma Warnings (Off, "assuming no concurrent accesses to non-atomic object
93          => ""Counter""");
94      Counter : aliased Packed_U32.T
95          with Import, Convention => Ada, Address => Counter_Address;
96      pragma Warnings (On, "assuming no concurrent accesses to non-atomic object
97          => ""Counter""");
98      pragma Warnings (On, "writing to ""Counter"" is assumed to have no effects on
99          => other non-volatile objects");
100     pragma Warnings (On, "indirect writes to ""Counter"" through a potential
101         => alias are ignored");
102
103    -- 0x0060000C - 0060000D: Another_Counter
104    -- An equally important counter.
105    Another_Counter_Offset : constant Unsigned_32 := 16#0060000C#; -- 0x0060000C
106    Another_Counter_Size : constant Unsigned_32 := Unsigned_32
107        => (Packed_U16.Size_In_Bytes); -- 2 bytes
108    Another_Counter_End : constant Unsigned_32 := Another_Counter_Offset +
109        => Another_Counter_Size - 1; -- 0x0060000D
110    Another_Counter_Address : constant System.Address := To_Address
111        => (Integer_Address (Another_Counter_Offset));
112
113    pragma Warnings (Off, "indirect writes to ""Another_Counter"" through a
114        => potential alias are ignored");
115    pragma Warnings (Off, "writing to ""Another_Counter"" is assumed to have no
116        => effects on other non-volatile objects");
117    pragma Warnings (Off, "assuming no concurrent accesses to non-atomic object
118        => ""Another_Counter""");
119    Another_Counter : aliased Packed_U16.T
120        with Import, Convention => Ada, Address => Another_Counter_Address;
121    pragma Warnings (On, "assuming no concurrent accesses to non-atomic object
122        => ""Another_Counter""");
123    pragma Warnings (On, "writing to ""Another_Counter"" is assumed to have no
124        => effects on other non-volatile objects");
125    pragma Warnings (On, "indirect writes to ""Another_Counter"" through a
126        => potential alias are ignored");
127
128    --
129    -- Memory map access types:
130    --
131
132    package Accesses
133        -- Note that general access types are not allowed in SPARK,
134        -- but we may want to use these in Ada code, so we provide them here.
135        with SPARK_Mode => Off
136
137    is
138        Time_Access : constant Sys_Time.T_Access := Time'Access;
139        Counter_Access : constant Packed_U32.T_Access := Counter'Access;
140        Another_Counter_Access : constant Packed_U16.T_Access :=
141            => Another_Counter'Access;

```

```

121 end Accesses;

122

123 --
124 -- Memory map compile-time checks. These checks help to ensure that there is
125 -- not a bug in the autocoder.
126 --
127 -- Make sure map boundary addresses makes sense:
128 pragma Compile_Time_Error (Start_Offset /= 16#00600000#, "Unexpected
129 -- autocoder error. Start address not as expected.");
130 pragma Compile_Time_Error (Last_Offset /= 16#007FFFFF#, "Unexpected autocoder
131 -- error. End address not as expected.");
132 pragma Compile_Time_Error (Last_Used_Offset /= 16#0060000D#, "Unexpected
133 -- autocoder error. Last used address not as expected.");
134 pragma Compile_Time_Error (Last_Offset < Start_Offset, "Unexpected autocoder
135 -- error. End address smaller than start address.");
136 pragma Compile_Time_Error (Last_Used_Offset > Last_Offset, "Unexpected
137 -- autocoder error. Last used address larger than last address.");
138 pragma Compile_Time_Error (Unused_Bytes /= 2097138, "Unexpected autocoder
139 -- error. Unused bytes not as expected.");
140 pragma Compile_Time_Error (Used_Bytes /= 14, "Unexpected autocoder error.
141 -- Used bytes not as expected.");

142 -- 0x00600000 - 00600007: Time Checks
143 pragma Compile_Time_Error (Time_Offset /= 16#00600000#, "Unexpected autocoder
144 -- error. Item start address not as expected.");
145 pragma Compile_Time_Error (Time_Size /= 8, "Unexpected autocoder error. Item
146 -- size not as expected.");
147 pragma Compile_Time_Error (Time_End /= 16#00600007#, "Unexpected autocoder
148 -- error. Item end address not as expected.");
149 pragma Compile_Time_Error (Time_Offset >= Time_End, "Unexpected autocoder
150 -- error. Item end address less than start address.");
151 pragma Compile_Time_Error (Time_Offset < Start_Offset, "Unexpected autocoder
152 -- error. Item starts before first address.");
153 pragma Compile_Time_Error (Time_End > Last_Offset, "Unexpected autocoder
154 -- error. Item ends after last address.");
155 pragma Compile_Time_Error (Time_Offset < Start_Offset, "Unexpected autocoder
156 -- error. Item overlap found.");

157 -- 0x00600008 - 0060000B: Counter Checks
158 pragma Compile_Time_Error (Counter_Offset /= 16#00600008#, "Unexpected
159 -- autocoder error. Item start address not as expected.");
160 pragma Compile_Time_Error (Counter_Size /= 4, "Unexpected autocoder error.
161 -- Item size not as expected.");
162 pragma Compile_Time_Error (Counter_End /= 16#0060000B#, "Unexpected autocoder
163 -- error. Item end address not as expected.");
164 pragma Compile_Time_Error (Counter_Offset >= Counter_End, "Unexpected
165 -- autocoder error. Item end address less than start address.");
166 pragma Compile_Time_Error (Counter_Offset < Start_Offset, "Unexpected
167 -- autocoder error. Item starts before first address.");
168 pragma Compile_Time_Error (Counter_End > Last_Offset, "Unexpected autocoder
169 -- error. Item ends after last address.");
170 pragma Compile_Time_Error (Counter_Offset <= Time_End, "Unexpected autocoder
171 -- error. Item overlap found.");
172 pragma Compile_Time_Error (Counter_Offset <= Time_Offset, "Unexpected
173 -- autocoder error. Item overlap found.");

174 -- 0x0060000C - 0060000D: Another_Counter Checks
175 pragma Compile_Time_Error (Another_Counter_Offset /= 16#0060000C|,
176 -- "Unexpected autocoder error. Item start address not as expected.");
177 pragma Compile_Time_Error (Another_Counter_Size /= 2, "Unexpected autocoder
178 -- error. Item size not as expected.");

```

```

158 pragma Compile_Time_Error (Another_Counter_End /= 16#0060000D#, "Unexpected
159   ↳ autocoder error. Item end address not as expected.");
160 pragma Compile_Time_Error (Another_Counter_Offset >= Another_Counter_End,
161   ↳ "Unexpected autocoder error. Item end address less than start address.");
162 pragma Compile_Time_Error (Another_Counter_Offset < Start_Offset, "Unexpected
163   ↳ autocoder error. Item starts before first address.");
164 pragma Compile_Time_Error (Another_Counter_End > Last_Offset, "Unexpected
165   ↳ autocoder error. Item ends after last address.");
166 pragma Compile_Time_Error (Another_Counter_Offset <= Counter_End, "Unexpected
167   ↳ autocoder error. Item overlap found.");
168 pragma Compile_Time_Error (Another_Counter_Offset <= Counter_Offset,
169   ↳ "Unexpected autocoder error. Item overlap found.");
170
171 end Nonvolatile_Store;

```

In the package, many useful constants are declared at the top which specify the boundaries of the memory map. The second section declares each item. The address of each item is based off of the address of the item previous to it. The third section declares a set of access types that refer to the types declared in the second section. These types of accesses are not allowed within SPARK, so they are declared with SPARK_Mode => Off. The final section specifies a myriad of `Compile_Time_Error` calls which employs the compiler to check the work of the generator to make sure that no memory regions overlap or exceed the region bounds. These global variables may now be safely used within the system, usually provided to components as part of their initialization.

We can also build PDF and HTML documentation to view the memory map definition in a human readable manner. To build HTML documentation run:

```
> redo build/html/nonvolatile_store.html
```

which when opened with your favorite web browser looks something like this:

Nonvolatile_Store Memory Map (0x00600000 – 0x007FFFFF) : Using 14 of 2097152 bytes

Description: This is the memory layout for an example set of variables for a mission.

Address	Name	Type	Size (Bytes)	Description
0x00600000 – 0x00600007	time	Sys_Time.T	8	An important timestamp.
0x00600008 – 0x0060000B	counter	Packed_U32.T	4	An important counter.
0x0060000C – 0x0060000D	another_Counter	Packed_U16.T	2	An equally important counter.

*This file was autogenerated from /vagrant/adamant/doc/example_architecture/memory_map/nonvolatile_store.memory_map.yaml on 2021-04-14 21:46.
© The University of Colorado, Laboratory for Atmospheric and Space Physics (LASP)

9.10 Advanced Build System Topics

The following sections provide more information on configuring and using the Adamant build system that was not presented in Section .

9.10.1 Modifying the Build Path

Modifying the *build path* is not a common operation that most users will have to perform, but this section sheds light on how to modify the build path should the need arise. The automatic calculation of the *build path* discussed in Section 3.3.3 can be added to, modified, overridden, or tweaked based on system environment variables which can be set from the command line, within a local .do file, or within a local env.py script. This section discusses modifying the *build path* using the command line, but see section for more details on how to accomplish the same thing using .do files or env.py

files.

The control variables described below can be used to override how Adamant calculates the *build path*.

- **BUILD_PATH** - If this variable is set, the path described here in the form /path/to/directory1:/path/to/directory2:etc is used as the entire build path. In this way the user can customize the entire build path to fit their needs. The automatic calculation of the build path is disabled and not used in this case.
- **BUILD_ROOTS** - If BUILD_PATH is not set, but this variable IS set in the form /path/to/root1:/path/to/root2:etc, then BUILD_PATH is calculated by recursively searching for *path files* from each specified build root directory in BUILD_ROOTS. Every directory found that includes an applicable path file is automatically added to the build path.

If neither of the above variables has been specified by the user, then BUILD_ROOTS is automatically assumed by Adamant to be a path with two roots: 1) The root directory of the Adamant repository, and 2) The root directory of the repository containing the current working directory (as determined by the location of the *.git* directory), if the current working directory is not within the Adamant repository. Calculation of the BUILD_PATH then follows in the same manner from these two directories as if BUILD_ROOTS were set manually by the user.

It is often the case that, for a specific build situation, the user wants to add a path or add a build root to the BUILD_PATH that is calculated. The following environment variables allow the user to append extra directories to a build path.

- **EXTRA_BUILD_PATH** - If this variable is set, the path described here in the form /path/to/directory1:/path/to/directory2:etc is added to the BUILD_PATH variable.
- **EXTRA_BUILD_ROOTS** - If this variable is set, the path described here in the form /path/to/root1:/path/to/root2:etc is added to the BUILD_ROOTS variable.

In addition, it is also common to want to remove a specific directory from an already computed build path. To do this a user can set the REMOVE_BUILD_PATH variable.

- **REMOVE_BUILD_PATH** - If this variable is set, the path described here in the form /path/to/directory1:/path/to/directory2:etc is removed from BUILD_PATH after it is calculated using the above methods.

This variable can be particularly useful if you want to override the functionality of a core package in Adamant, by first writing your own version of it and including it in the build path via EXTRA_BUILD_PATH or EXTRA_BUILD_ROOTS, and then subsequently removing the path to the core package from the build path by adding it to REMOVE_BUILD_PATH.

To be clear, any of the variables described above can be set on the command line via:

```
> export BUILD_PATH=/path/to/dir1:/path/to/dir2 # or  
> export EXTRA_BUILD_ROOTS=/path/to/root_dir1:/path/to/root_dir2
```

or can be added within a local *.do* file or within a local *env.py* script to provide a more permanent setting. See Section for more details about saving a build path configuration.

Configuring the build path correctly can be difficult to debug without being able to view the path that the Adamant build system is calculating. You can view the setting of all of the above variables when running any *redo* command by first setting the build system in debug mode, see Section 9.10.2. In debug mode, the values of all of the above variables will be displayed and the calculated build path and build path roots used for *redo* command will be displayed as the COMPUTED_BUILD_PATH and COMPUTED_BUILD_ROOTS variables, respectively.

Note that you should never need to directly add an autocoded source directory (*build/src*) to the build path. Autocoded source directories are automatically added to the build path if the directory containing *build/src* is in the build path.

9.10.2 Debug Mode

The build system has a debug mode in which verbose information about what is being executed by the build system is printed to the terminal during compilation. To enable this mode simply set the DEBUG variable from the command line to anything. To turn it off, you need to “unset” the variable:

```
> export DEBUG=1 # Turn build system debug mode ON  
> export DEBUG= # Turn build system debug mode OFF
```

Debug mode can be used to view generators that are being executed, view the compilation commands that are being run, or view the *build path* computation, as discussed in the previous section.

9.10.3 Setting the Environment

As seen in previous sections the build system is configured based on the user’s environment, which can be set via environment variables. Setting these using the command line only provides a temporary configuration. Adamant provides the following, more permanent methods for configuring the environment:

1. Adding environment variables to the shell’s configuration file (ie. *.bashrc*).
2. Adding an *env.py* file to a directory to configure the environment for *redo* rules run within that directory.
3. Any custom redo rules written in a *.do* file can tailor their environment by setting the environment variables within.

Option 1 is commonly used in Adamant to set the project-wide build paths if the Adamant defaults do not suffice for a project. See Section 3.1 for more details.

Option 2 is commonly used to configure the environment for *redo* commands run within a specific directory. If a *redo* rule is run within a directory that contains an *env.py* file, the *env.py* file is executed first, before the rule is executed. In this way, the user can set up the environment for all rules that might be run in a specific directory separately from the rest of the system. This method is commonly used to set up a specific unit test configuration that may be different than the default compilation configuration. You will find the following *env.py* file within most Adamant unit test directories.

```
1  from environments import test # noqa: F401
```

which sets the *build target* to the test version for the current target, ie. if the TARGET is currently Linux then the *env.py* will change TARGET for the test directory to Linux_Test instead, unless the current TARGET ends in Coverage. This *env.py* shows the standard Adamant unit test environment and should be included in every unit test directory.

The unit test *env.py* is slightly complicated, but note that anything can be written in an *env.py*, and it will be executed. More often, a user simply uses the python *os* module to set and unset environment variables. For instance, if you always want to be in debug mode in a certain directory you can simply create an *env.py* that looks like:

```
1  import os  
2  os.environ["DEBUG"] = "1"
```

Option 3 mentions that any environment can be set up for custom build rules. Writing custom build rules is an advanced topic discussed in detail in Section 9.10.4.

9.10.4 Adding a Build Target

As discussed in Section 3.3.2, the *build target* determines how objects are compiled using the Adamant build system. To add support for a new processor, platform, or compilation options, a new build target needs to be created within the Adamant build system. The steps for doing this are provided in this section.

The Adamant supported build targets are found in the *redo/targets* directory in the Adamant repository. You most likely will NOT be adding a new build target directly to Adamant, and instead you will be adding a build target to your project repository in an analogous location *redo/targets*. Either way, the same pattern presented below holds.

To compile Ada and C/C++ code, the Adamant build system uses [GPRbuild](#) under the hood. Note that GPRbuild is an extensive build tool that can handle dependency management, cross compilation, etc. Adamant does not use GPRbuild's full capabilities, as the Adamant build system itself manages much of the build process. GPRBuild is instead used only to compile objects and link binaries. The Adamant build system manages the rest of the orchestration of the build system, including code generation. Because GPRBuild handles the compilation, defining build flags, the Ada runtime, and linking external libraries is all done in a *.gpr* file. Adamant interfaces with this *.gpr* file through a small python class that ties the two build systems together.

To be specific, a *build target* is defined by creating a single python class that inherits from `gpr_build_target_base`. The class specifies the following:

1. The GPRBuild file (*.gpr*) used for compilation, which determines the compiler, runtime, configuration pragmas, and options used for compilation, binding, and linking.
2. The *path files* that apply to the given target, see Section 3.3.3.

These two specifications can be easily viewed for any *build target* on the system by running `redo targets`, see Section 3.3.2.

With this in mind, the suggested steps for adding a build target are presented below:

1. Add a new *.gpr* file for your platform in *redo/targets/gpr*. You should look at other examples and follow the same pattern. In particular, your *.gpr* file will need to *extend* from the *a_adamant.gpr* file and include the same first few lines (see comments in *linux.gpr* for example). Note that GPRBuild files that begin with *a_* are abstract project files that must be extended in order to use. After this, you can define, using the standard GPRBuild language, your compiler, binder, and linker options. Note that Adamant handles defining the files used for compiling or linking an object so you should NEVER need to define `Source_Dirs` (or anything equivalent) to specify the files that need to be built.
2. In *redo/targets*, define a new python class that will link the *.gpr* file to the Adamant build system. This class must inherit from `base_classes.gprbuild_target_base` and define a few simple functions. See the Adamant *redo/targets* directory for examples of how this is done.
3. Finally, make sure that your new build target is set up correctly. First run `redo targets` and make sure your new target is listed.
4. Next, set the target to your new build target using `export TARGET=my_new_target` and try to compile an object or executable. From here you should be able to debug your *.gpr* file and python class until your new target works!

9.10.5 Extending the Build System

The Adamant build system utilizes the `redo` build tool. This makes extending its capability straight forward. If you look at the `redo docs` you will see that a redo script, a `.do` file, can be created anywhere and using any interpreted language, the most common being shell. Adamant supports this basic usage of redo. To be specific, you can create a `.do` file anywhere, in any language, and expect it to function according to the redo documentation. You will see examples of `.do` files scattered throughout the Adamant repository that are used to perform specific actions that may be different from the default.

The default behavior of the Adamant build system is provided by the `.do` files found at the Adamant repository root, and any project repository root. These `.do` files provide the main build rules for redo, and direct the execution to the python based build system found in the `redo/` directory. If you have a one-off extension of the build system that you want to support, feel free to simply define a specific `.do` file where you want to support that behavior. If you want to make a more wide-spread change to the Adamant build system, you will want to follow the pattern for adding a new build rule within Adamant. The process for doing this is described below:

1. Add your new build rule in `redo/rules` within the Adamant repository or, add it to your project repository if the new rule only applies to your specific project. Build rules are python classes that must inherit from the `base_classes.build_rule_base` class. Follow the many examples provided in that directory to write your own build rule. Note that you will need to override the `input_file_regex` and `output_filename` functions in order to provide `redo` what support for your new build rule.
2. Next, you need to “wire” your build rule to `redo` by defining or modifying a `.do` file at the top level of the Adamant repository (and your project repository). If your build rule applies to a new file output type then you will need to create a new `.do` file. If your build rule does not produce a concrete output and is *abstract*, ie. similar to `redo what`, `redo test_all`, or `redo publish`, then you will need to modify the `default.do` file to add your new rule.
3. Test your new rule by trying to build something with it using `redo`.

Note that adding some build rules, such as compiling for a new programming language, will require more work than the procedure presents above. In that case, it is recommended to understand the *set up* and *database* portions of the build system located in `redo/database`.

9.10.6 Model Caching

Note: The Adamant persistent model cache is currently disabled by default due to some instability in its behavior. If you want to use this experimental feature you can enable it via:

```
> export ENABLE_PERSISTENT_MODEL_CACHE=1
```

but be aware that sometimes the cache may not always be refreshed correctly when dependencies change, and clearing the cache manually might be needed (see below). The description below applies if this feature is enabled.

When building a target, the Adamant build system spends a lot of time reading YAML models from the file system, validating them, and then using them to generate outputs. Since most of these models do not change often on disk, the build system caches them in a database in `/temp` to speed up build times. It is important when designing python models to correctly track the dependencies so that cached entries can be refreshed when a YAML model, or a YAML model’s dependency, has changed. To do this, ensure that your python model correctly implements the `get_dependencies` method, which should return a list of all YAML files that the current YAML file depends on. For example, a `*.component.yaml` always depends on its IDed entity models: `*.data_products.yaml`, `*.data_commands.yaml`, etc.

If you suspect that the cache is messed up, or you want to ensure a completely clean build, you can clear the cache by running:

```
> redo clear_cache
```

which will remove the entire cache database from the system. It will be recreated next time you build a new target.

9.11 The Generator System

The primary generator/autocoding system for Adamant is contained within the `gen/` directory. Most generators follow a similar pattern. The input is usually a human written and human readable **YAML** file which is first validated by a schema and then ingested into a python data structure called a model. These models are then output into many different autogenerated text files using the **Jinja2** templating engine. Modifying existing generators or adding your own is intended to be a straight forward process, however some investigation will be required on the reader's part, as the minute details cannot be fully documented here.

9.11.1 Anatomy a Generator

Before diving into the details, it is good to have a high level understanding of how generators are created in Adamant. The generator system is made up of a few primary pieces, each which serves a certain purpose during the file generation process. Below is a graphic summary.

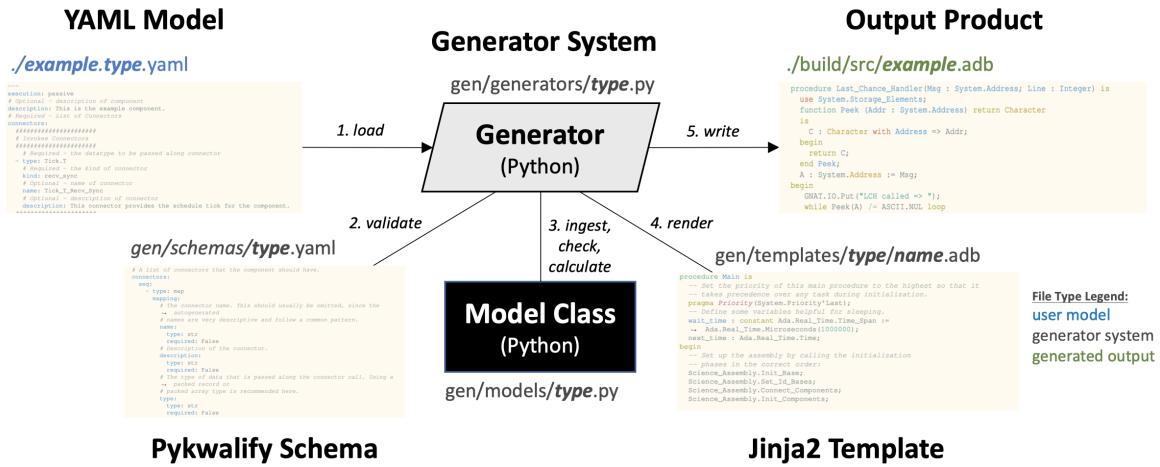


Figure 8: The graphic above shows the Adamant generator system elements. The generator output is a file `example.adb` that is constructed from data found within a model file `example.type.yaml`.

For the example above assume that the current working directory (`./`) includes a model file called `example.type.yaml`. From this model file we are trying to construct an output file `./build/src/example.adb`. Note that any file path in the graphic above that begins with `gen/` is part of the Adamant generator system and will indeed be found in the `gen/` directory inside of the Adamant repository.

Model files are always named as `model_name.model_type.yaml`. So in this case the model name is `example` and the model type is `type`. This is important, because the names of these elements are used by the Adamant generator system to determine how to process the file.

Note that the whole process above begins when the command `redo ./build/src/example.adb` is run. This invokes the Adamant build system, which searches for the correct build rule to construct the file. When the build rule is found, it will indicate that a file named `example.type.yaml` must exist in order to build the desired output. The build system will ensure that this file exists, and then the generator process shown in the diagram commences.

The first step is that a generator python module with the same name as the model file type, ie. *type*, will load the model file from disk. The model file is validated by comparing it against a YAML schema. The schema used is also named by the model file type. Adamant schemas are written in the [pykwalify](#) schema language. Many of the schemas are included in the Appendix of this document.

If validation is successful, the generator loads the yaml model file into a python model class, also with the same name as the model type. This model class consists of object-oriented python that ingests the YAML data into a more suitable data structure for computation. As part of this process some checking of the model integrity may occur that cannot be accomplished at the schema level. Calculations may also be performed to derive extra data from the model data. It is also common for model classes to load additional model classes to gather more data. This is the case with component models, which load IDed entity models to gather information about the component's events, data products, etc.

After the model is ingested into a python model class, the class' data is rendered into an output template using the [Jinja2](#) templating engine. The template to use is, again, determined by the model type. In Adamant, templates will include the text *name* in their template files to signify where a substitution should occur for the model name. In this example, the template *name.adb* will produce an output *example.adb* because the model it was produced from has the model name *example*. After the data is rendered using the output template, it is stored on disk in the correct location and with the correct filename by the Adamant build system.

The next section will provide more detailed information to help you modify existing generators or add your own.

9.11.2 Adding a Generator

To better understand the generator system, it is instructive to understand how the different pieces are laid out in the repository. The following is a description of what you can expect to find in the subdirectories of *gen/*.

- **gen/** - Adamant generator system used for generating structural autocode, configuration files, diagrams, and documentation
- **gen/generators** - python generators which tie in with the Adamant build system to create build rules for each generated file
- **gen/models** - python model classes which ingest and calculate all modeling information used by the generators
- **gen/schemas** - YAML schemas written using [pykwalify](#), used to validate user model files prior to running generators
- **gen/templates** - [Jinja2](#) templates used in outputting various autocoded files
- **gen/test** - a set of regression tests for the generators to ensure their functionality before deployment on projects

There are many generators already existing within Adamant, so the best way to build your own is to use the existing examples. Below is a suggested procedure to help you get started.

1. Take a look at the different files in each of the directories above to familiarize yourself with what they are doing.
2. If you need a new input model file type for your generator, consider designing the input file along with a schema. You can then use [pykwalify](#) from the command line to validate your input file format. Put your schema in *gen/schemas/*.
3. Input files are usually ingested into python data structures called models. If you need a new model to represent the data from your input file, add a new python class file to *gen/models/*. Make sure to look at the *gen/models/base.py* model, from which you will most likely want to

derive your class.

4. For each output file that you would like to produce, add a Jinja2 template in the *gen/templates/* directory, which will reference data elements from the model you created in the previous step.
5. For each template that you created add a generator class in the *gen/generators/* directory, which ties together the YAML ingest, the model, and the templating output. This class must inherit from *base_classes.generator_base* in order to be integrated with the build system and create build rules.
6. If you want your generator to be maintained in the future, it is a very good idea to write regression tests for your generator in the *gen/test/* directory.

Appendix A Model Schemas

This section presents the schemas used to validate Adamant models prior to ingest and autocoding. These schemas are written in `pykwalify` schema format. Even if you are not familiar with `pykwalify`, the schemas are well commented. Looking at these schemas can be a good way to understand all the possible fields that can be specified in an Adamant model.

All schemas can be found in the Adamant `gen/schemas` directory. A filename is included before each schema so that you can find it in the Adamant repository if necessary. They are listed in alphabetical order.

A.1 Assembly Model Schema

YAML schema for model files of the form `name.assembly.yaml`.

`gen/schemas/assembly.yaml`:

```
1  ---
2  # This schema describes the yaml format for an assembly.
3  type: map
4  mapping:
5      # A description for the assembly.
6      description:
7          type: str
8          required: False
9      # Many "with" dependencies are automatically deduced and included by
10     # the generator. If you want to manually add a "with" statement, you
11     # can list the names of the packages here.
12     with:
13         seq:
14             - type: str
15             required: False
16     # Manual "with" statements that should only appear in the .adb generation
17     with_adb:
18         seq:
19             - type: str
20             required: False
21     # Any useful handcode to include in the .ads file can be included here.
22     # You can think of this as inline Ada, which might be useful for declaring
23     # custom enum types and the like. The code here is copied immediately
24     # before the package spec definition.
25     preamble:
26         type: str
27         required: False
28     # Any useful handcode to include in the .ads file can be included here.
29     # You can think of this as inline Ada, which might be useful for declaring
30     # custom enum types and the like. The code here is copied immediately after
31     # the package spec definition.
32     preamble:
33         type: str
34         required: False
35     # A list of subassemblies to include in this assembly. The names listed here
36     # should just be the assembly name. For example:
37     #
38     # subassemblies:
39     #     - core
40     #     - comm
41     #
42     # would import the subassemblies from files named core.assembly.yaml and
43     # comm.assembly.yaml.
44     #
```

```

45 # Make sure subassemblies are in the build path or an error will result.
46 # Subassemblies can be used to split up large assembly files into smaller,
47 # more manageable files. This can facilitate reuse and ease assembly
48 # maintenance.
49 subassemblies:
50     seq:
51         - type: str
52         required: False
53     # The following can be used to set the ID bases of different assembly wide
54     # entities.
55     # This can be used, for instance, to start all event IDs at 0x500, ex:
56     # id_bases:
57     #     - "event_Id_Base => 1280"
58     #     - etc.
59     id_bases:
60         seq:
61             - type: str
62             required: False
63     # A list of components to include in the assembly.
64     components:
65         seq:
66             - type: map
67             mapping:
68                 # The name of the component. It is recommended that you leave this
69                 # blank, as the autogenerated name is usually descriptive enough and
70                 # follows a convention.
71             name:
72                 type: str
73                 required: False
74             # Description of the component and its role in the assembly.
75             description:
76                 type: str
77                 required: False
78             # The component type.
79             type:
80                 type: str
81                 required: True
82             # The execution type of the component.
83             #     active - the component is given an internal task on which to
84             #         execute
85             #     passive - the component is not given an internal task, and will
86             #         only
87             #             execute on the task of it's connector invokers.
88             # Note: if not provided, the execution type defined in the component
89             #         model
90             # will be used. If the type defined in the component model is "either"
91             #         then
92             #             an execution MUST be provided here.
93             execution:
94                 type: str
95                 enum: ['active', 'passive']
96                 required: False
97             # This is only required for active components. The priority number
98             #         that
99             # the component task should execute at.
      priority:
        type: int
        required: False
      # This is only required for active components. The stack size in bytes
      #         that the component task should have.
      stack_size:

```

```

100     type: int
101     required: False
102     # This is only required for active components. The secondary stack
103     # size in bytes
104     # that the component task should have. The secondary stack is used to
105     # return
106     # unconstrained types from function.
107     secondary_stack_size:
108         type: int
109         required: False
110         # If the component has generic types, the actual types that they
111         # should be
112         # resolved to in the assembly should be provided here.
113         generic_types:
114             seq:
115                 - type: str
116                 required: False
117                 # If the component has values to fill in for the discriminant they
118                 # should be
119                 # listed here as name value pairs, ie.
120                 # "Name => Value"
121         discriminant:
122             seq:
123                 - type: str
124                 required: False
125                 # If the component has values to fill in during component base
126                 # initialization they should be
127                 # listed here as name value pairs, ie.
128                 # "Name => Value"
129         init_base:
130             seq:
131                 - type: str
132                 required: False
133                 # If the component has values to fill in during id base initialization
134                 # they should be
135                 # listed here as name value pairs, ie.
136                 # "Name => Value"
137         set_id_bases:
138             seq:
139                 - type: str
140                 required: False
141                 # If the component needs to map data dependencies to an external
142                 # component's data product then
143                 # the data dependencies should be listed here with the corresponding
144                 # data products and stale
145                 # limits. The IDs and stale limits for each data dependency will then
146                 # be correctly set up at assembly
147                 # initialization.
148         map_data_dependencies:
149             seq:
150                 - type: map

```

```

151         required: True
152         # If the data product has a timestamp older than the current
153         # time (time reference)
154         # minus this stale limit in microseconds, then it will be
155         # considered stale. In this
156         # case, a stale status will be returned to the component upon
157         # parsing the data product.
158         # A stale_limit_us of zero indicates that the data product can
159         # never be stale, and
160         # in this case the stale-ness of the data product will not
161         # even be checked, thus a
162         # stale status will never be returned to the component upon
163         # parsing the data product.
164     stale_limit_us:
165         type: int
166         required: True
167         required: False
168         # If the component has values to fill in during component
169         # initialization they should be
170         # listed here as name value pairs, ie.
171         # "Name => Value"
172     init:
173         seq:
174             - type: str
175             required: False
176         # If the component has an internal subtask then it needs to be
177         # initialized here.
178     subtasks:
179         seq:
180             - type: map
181             mapping:
182                 # The name of the subtask.
183                 name:
184                     type: str
185                     required: True
186                 # The priority number that the component subtask should
187                 # execute at.
188                 priority:
189                     type: int
190                     required: True
191                 # The stack size in bytes that the component subtask should
192                 # have.
193                 stack_size:
194                     type: int
195                     required: True
196                 # The secondary stack size in bytes that the component subtask
197                 # should have. The secondary
198                 # stack is used to return unconstrained types from function.
199                 secondary_stack_size:
200                     type: int
201                     required: True
202                 # Set to true if you do not want to start the subtask for this
203                 # component. By default
204                 # this is True:
205                 disabled:
206                     type: bool
207                     required: False
208                     default: False
209                 required: False
210             # At least one component must be provided in the assembly.
211         range:

```

```

200     min: 1
201     required: True
202     # A list of connections to make between components in the assembly.
203     connections:
204       seq:
205         - type: map
206           mapping:
207             # A description of this connection, and what it is for.
208             description:
209               type: str
210               required: False
211             # The from component name.
212             from_component:
213               type: str
214               required: True
215             # The from connector name.
216             from_connector:
217               type: str
218               required: True
219             # If the from connector is an arrayed connector, the array
220             # index should be included here.
221             from_index:
222               type: int
223               required: False
224               range:
225                 min: 1
226                 max: 65535
227             # The to component name.
228             to_component:
229               type: str
230               required: True
231             # The to connector name.
232             to_connector:
233               type: str
234               required: True
235             # If the to connector is an arrayed connector, the array
236             # index should be included here.
237             to_index:
238               type: int
239               required: False
240               range:
241                 min: 1
242                 max: 65535
243               required: False

```

A.2 Commands Model Schema

YAML schema for model files of the form *component_name.commands.yaml*.

gen/schemas/commands.yaml:

```

1   ---
2   # This schema describes the yaml format for a command suite.
3   type: map
4   mapping:
5     # Description of the command suite.
6     description:
7       type: str
8       required: False
9     # List of commands to include in the suite.

```

```

10 commands:
11   seq:
12     - type: map
13       mapping:
14         # Name (mnemonic) of the command.
15         name:
16           type: str
17           required: True
18         # Description of the command.
19         description:
20           type: str
21           required: False
22         # If the command contains arguments, provide a single Ada type
23         # that contains all of the arguments. Using a packed record or packed
24         # array type is recommended here.
25         arg_type:
26           type: str
27           required: False
28         # A command suite must have at least one command.
29         range:
30           min: 1
31         required: True

```

A.3 Component Model Schema

YAML schema for model files of the form *name.component.yaml*.

gen/schemas/component.yaml:

```

1 ---
2   # This schema describes the yaml format for a component.
3   type: map
4   mapping:
5     # The component "execution" which describes how the component is run:
6     # active - the component includes an internal task
7     # passive - the component does not include an internal task and is only
8     # executed by its connector invokers.
9     # either - the component is designed to be executed with its own task or
10    # not. Whether the component is
11    #           active or passive is decided as the assembly level.
12    execution:
13      type: str
14      enum: ['active', 'passive', 'either']
15      required: True
16    # Textual description of the component
17    description:
18      type: str
19      required: False
20    # Many "with" dependencies are automatically deduced and included by
21    # the generator. If you want to manually add a "with" statement, you
22    # can list the names of the packages here.
23    with:
24      seq:
25        - type: str
26        required: False
27    # Many "without" dependencies are automatically deduced and included by
28    # the generator. If you want to manually remove a "without" statement, you
29    # can list the names of the packages here.
30    without:
31      seq:

```

```

30     - type: str
31     required: False
32     # Any useful handcode to include in the .ads file can be included here.
33     # You can think of this as inline Ada, which might be useful for declaring
34     # custom enum types and the like. The code here is copied immediately after
35     # the package spec definition.
36     preamble:
37         type: str
38         required: False
39         # If the component is parameterized by some generic types or generic
40         # → subprograms
41         # they should be listed here. The name of the generic type must be provided.
42         # → The
43         # generic formal type will be included in the base package for the component.
44         # → By
45         # default, the generic formal type will be "Type T is private;" where "T" is
46         # → the
47         # name of the generic type. If you would like to use a different generic
48         # → formal
49         # type definition, then use the "formal_type:" field.
50     generic:
51         type: map
52         mapping:
53             # Description of the generic set of parameters:
54             description:
55                 type: str
56                 required: False
57                 parameters:
58                     seq:
59                         - type: map
60                         mapping:
61                             # Name of the generic type:
62                             name:
63                                 type: str
64                                 required: True
65                             # Declaration of the generic formal type. By default if this is
66                             # → not specified
67                             # then the generic formal type will be "Type T is private;" where
68                             # → "T" is the
69                             # name of the generic type.
70                             formal_type:
71                                 type: str
72                                 required: False
73                                 # Description of the generic type.
74                                 description:
75                                     type: str
76                                     required: False
77                                     # Is the formal type optional? Only set this to true if you have
78                                     # → provided a default
79                                     # in the "formal_type" field above.
80                                     optional:
81                                         type: bool
82                                         required: False
83                                         required: True
84                                         required: False
85                                     # User defined discriminant parameters. This should be used to initialize the
86                                     # → component
87                                     # with any parameters needed for its functioning.
88                                     discriminant:
89                                         type: map
90                                         mapping:

```

```

82     # Description of the user initialization function.
83     description:
84         type: str
85         required: False
86     # A list of the arguments to be passed to the init function:
87     parameters:
88         seq:
89             - type: map
90             mapping:
91                 # Description of the initialization parameter.
92                 description:
93                     type: str
94                     required: False
95                 # Name of the initialization parameter.
96                 name:
97                     type: str
98                     required: True
99                 # Type of the initialization parameter.
100                type:
101                    type: str
102                    required: True
103                # Whether the initialization parameter can be set to null.
104                not_null:
105                    type: bool
106                    required: False
107                required: False
108            required: False
109        # User defined initialization method. This method should allocate any memory
110        ↪ on the heap
111        # that is necessary for the component's functioning.
112        init:
113            type: map
114            mapping:
115                # Description of the user initialization function.
116                description:
117                    type: str
118                    required: False
119                # A list of the arguments to be passed to the init function:
120                parameters:
121                    seq:
122                        - type: map
123                        mapping:
124                            # Description of the initialization parameter.
125                            description:
126                                type: str
127                                required: False
128                            # Name of the initialization parameter.
129                            name:
130                                type: str
131                                required: True
132                            # Type of the initialization parameter.
133                            type:
134                                type: str
135                                required: True
136                            # The default value for the parameter.
137                            default:
138                                type: str
139                                required: False
140                            # Whether the parameter can be set to null.
141                            not_null:
142                                type: bool

```

```

142         required: False
143     required: False
144     required: False
145 # If the component services an interrupt, it should be documented here.
146 interrupts:
147     seq:
148         - type: map
149             mapping:
150                 # The interrupt name, required.
151                 name:
152                     type: str
153                     required: True
154                 # Description of the interrupt.
155                 description:
156                     type: str
157                     required: False
158 # At least one interrupt required in list.
159 range:
160     min: 1
161     required: False
162 # If a component uses an internal subtask, other than the primary component
163     ↪ task,
164 # then its name should be listed here. This allows the assembly to ensure that
165     ↪ the
166 # component subtask is properly created at initialization.
167 subtasks:
168     seq:
169         - type: map
170             mapping:
171                 # The name of the subtask.
172                 name:
173                     type: str
174                     required: True
175                 # The description for the subtask
176                 description:
177                     type: str
178                     required: False
179 # A list of connectors that the component should have.
180 connectors:
181     seq:
182         - type: map
183             mapping:
184                 # The connector name. This should usually be omitted, since the
185                 ↪ autogenerated
186                 # names are very descriptive and follow a common pattern.
187                 name:
188                     type: str
189                     required: False
190                 # Description of the connector.
191                 description:
192                     type: str
193                     required: False
194                 # The type of data that is passed along the connector call. Using a
195                 ↪ packed record or
196                 # packed array type is recommended here.
197                 type:
198                     type: str
199                     required: False
200                 # The type of data that is returned from the connector call.
201                 return_type:

```

```

199      type: str
200      required: False
201      # The connector "kind" which describes how data is passed along the
202      # → connector.
203      # recv_sync - type is received synchronously
204      # recv_async - type is received asynchronously and queued.
205      # send - type is sent synchronously
206      # request - type is sent synchronously and return_type is returned
207      # → back synchronously
208      # service - type is received synchronously and return_type is sent
209      # → back synchronously to the invoker
210      # get - no type is set with this synchronous message, but
211      # → return_type is expected synchronously
212      # return - return_type is sent back to caller synchronously
213      # provide - type is sent synchronously, and could be modified before
214      # → being returned back to the invoker synchronously
215      # modify - type is received synchronously and can be modified before
216      # → being returned back to the invoker synchronously
217      kind:
218          type: str
219          enum: ['recv_sync', 'recv_async', 'send', 'request', 'service',
220              # → 'get', 'return', 'provide', 'modify']
221          required: True
222          # Is this connector an arrayed connector? If so, a value greater than
223          # → 1 should be
224          # provided here. A value of 0 assumes an unconstrained arrayed
225          # → connector, whose size
226          # is determined by the assembly the component is part of.
227          count:
228              type: int
229              range:
230                  min: 0
231                  max: 65535
232                  required: False
233                  # Does this connector have a priority? If so, a value can be supplied
234                  # → here. The larger
235                  # the value, the greater the priority. This only applies to
236                  # → asynchronous receive connectors,
237                  # and an error will be thrown if the connector kind is not async_recv.
238                  # → If the component's
239                  # → asynchronc_recv connectors have different priorities then the component
240                  # → will be instantiated
241                  # with a priority queue. If the asynchronc_recv connectors all have the
242                  # → same priority (unspecified)
243                  # priorites default to 0) then the component will be instantiated
244                  # → with a standard FIFO queue.
245                  priority:
246                      type: int
247                      range:
248                          min: 0
249                          max: 255
250                          required: False
251                      # At least one connector is required per component.
252                      range:
253                          min: 1
254                          required: True

```

A.4 Data Dependencies Model Schema

YAML schema for model files of the form *component_name.data_dependencies.yaml*.

gen/schemas/data_dependencies.yaml:

```
1  ---
2  # This schema describes the yaml format for a data_dependencies suite.
3  type: map
4  mapping:
5      # Description of the data_dependencies suite.
6      description:
7          type: str
8          required: False
9      # List of data_dependency items to include in the suite.
10     data_dependencies:
11         seq:
12             - type: map
13                 mapping:
14                     # Name of the data_dependency item.
15                     name:
16                         type: str
17                         required: True
18                     # Description of the data_dependency item.
19                     description:
20                         type: str
21                         required: False
22                     # The type of the data_dependency item. This must match the type of
23                     # the data product that this data dependency refers to.
24                     type:
25                         type: str
26                         required: True
27                     # A data_products suite must have at least one data_dependency item.
28                     range:
29                         min: 1
30                         required: True
```

A.5 Data Products Model Schema

YAML schema for model files of the form *component_name.data_products.yaml*.

gen/schemas/data_products.yaml:

```
1  ---
2  # This schema describes the yaml format for a data_products suite.
3  type: map
4  mapping:
5      # Description of the data_products suite.
6      description:
7          type: str
8          required: False
9      # List of data_products items to include in the suite.
10     data_products:
11         seq:
12             - type: map
13                 mapping:
14                     # Name of the data_products item.
15                     name:
16                         type: str
17                         required: True
```

```

18     # Description of the data_products item.
19     description:
20         type: str
21         required: False
22     # The type of the data_products item. Using a packed record or packed
23     # array type is recommended here.
24     type:
25         type: str
26         required: True
27     # A data_products suite must have at least one data_products item.
28     range:
29         min: 1
30         required: True

```

A.6 Enumeration Model Schema

YAML schema for model files of the form *name.enums.yaml*.

gen/schemas/enums.yaml:

```

1  ---
2  # This schema describes the yaml format for an enumeration.
3  type: map
4  mapping:
5      # A description for this set of enumerations
6      description:
7          type: str
8          required: False
9      # Many "with" dependencies are automatically deduced and included by
10     # the generator. If you want to manually add a "with" statement, you
11     # can list the names of the packages here.
12     with:
13         seq:
14             - type: str
15             required: False
16     # Any useful handcode to include in the .ads file can be included here.
17     # You can think of this as inline Ada, which might be useful for declaring
18     # custom types and the like.
19     preamble:
20         type: str
21         required: False
22     # A list of the fields to be included in the packed record.
23     enums:
24         seq:
25             - type: map
26             mapping:
27                 # The name of the enumeration
28                 name:
29                     type: str
30                     required: True
31                 # A description for this enumeration
32                 description:
33                     type: str
34                     required: False
35                 # A description for the field.
36                 literals:
37                     seq:
38                         - type: map
39                         mapping:
40                             # The name of the literal

```

```

41      name:
42          type: str
43          required: True
44          # The integer value of the literal. It defaults to the closest
45          # available value to zero remaining in the enumeration.
46          value:
47              type: int
48              required: False
49              # A description for this literal
50              description:
51                  type: str
52                  required: False
53                  range:
54                      min: 1
55                      required: True
56          range:
57              min: 1
58              required: True

```

A.7 Events Model Schema

YAML schema for model files of the form *component_name.events.yaml*.

gen/schemas/events.yaml:

```

1  ---
2  # This schema describes the yaml format for an event suite.
3  type: map
4  mapping:
5      # Description of the event suite.
6      description:
7          type: str
8          required: False
9      # List of events to include in the suite.
10     events:
11         seq:
12             - type: map
13                 mapping:
14                     # Name of the event.
15                     name:
16                         type: str
17                         required: True
18                     # Description of the event.
19                     description:
20                         type: str
21                         required: False
22                     # If the event contains parameters, provide a single Ada type
23                     # that contains all of the parameters. Using a packed record or packed
24                     # array type is recommended here.
25                     param_type:
26                         type: str
27                         required: False
28                     # An event suite must have at least one event.
29                     range:
30                         min: 1
31                         required: True

```

A.8 Faults Model Schema

YAML schema for model files of the form *component_name.faults.yaml*.

gen/schemas/faults.yaml:

```
1  ---
2  # This schema describes the yaml format for a fault suite.
3  type: map
4  mapping:
5      # Description of the fault suite.
6      description:
7          type: str
8          required: False
9      # List of faults to include in the suite.
10     faults:
11         seq:
12             - type: map
13                 mapping:
14                     # Name of the fault.
15                     name:
16                         type: str
17                         required: True
18                     # Description of the fault.
19                     description:
20                         type: str
21                         required: False
22                     # Identifier for the fault. This is optional, but if provided for one
23                     # fault, must be provided for all faults. If not provided then fault
24                     # IDs will be set via the Set_Id_Base initialization function. If IDs
25                     # are provided, then fault IDs are static.
26                     id:
27                         type: int
28                         required: False
29                     # If the fault contains parameters, provide a single Ada type
30                     # that contains all of the parameters. Using a packed record or packed
31                     # array type is recommended here.
32                     param_type:
33                         type: str
34                         required: False
35                     # A fault suite must have at least one fault.
36                     range:
37                         min: 1
38                     required: True
```

A.9 Packed Array Model Schema

YAML schema for model files of the form *name.array.yaml*.

gen/schemas/array.yaml:

```
1  ---
2  # This schema describes the yaml format for a packed array.
3  type: map
4  mapping:
5      # A description for the packed array.
6      description:
7          type: str
8          required: False
```

```

9   # Any useful handcode to include in the .ads file can be included here.
10  # You can think of this as inline Ada, which might be useful for declaring
11  # custom enum types and the like.
12  preamble:
13      type: str
14      required: False
15  # Many "with" dependencies are automatically deduced and included by
16  # the generator. If you want to manually add a "with" statement, you
17  # can list the names of the packages here.
18  with:
19      seq:
20          - type: str
21          required: False
22  # The element type for the array.
23  type:
24      type: str
25      required: True
26  # If the element is a primitive Ada type, provide a format
27  # string that describes the element size in bits and its signedness.
28  # when packed. ie.
29  #   U32, I16, F32, etc.
30  format:
31      type: str
32      required: False
33  # The length of the array, ie. how many items are in the array.
34  length:
35      type: int
36      range:
37          min: 1
38      required: True
39  # There is no predefined 'Image or autocoded .Image available for this element
40  # so instead just print the bytes as an array of unsigned_8 bit integers.
41  byte_image:
42      type: bool
43      required: False
44      default: False
45  # There is no way to validate this element type, so don't try to autocode it.
46  skip_validation:
47      type: bool
48      required: False
49      default: False

```

A.10 Packed Record Model Schema

YAML schema for model files of the form *name.record.yaml*.

gen/schemas/record.yaml:

```

1  ---
2  # This schema describes the yaml format for a packed record.
3  type: map
4  mapping:
5      # A description for the packed record.
6      description:
7          type: str
8          required: False
9      # Any useful handcode to include in the .ads file can be included here.
10     # You can think of this as inline Ada, which might be useful for declaring
11     # custom enum types and the like.

```

```

12  preamble:
13      type: str
14      required: False
15  # Many "with" dependencies are automatically deduced and included by
16  # the generator. If you want to manually add a "with" statement, you
17  # can list the names of the packages here.
18  with:
19      seq:
20          - type: str
21          required: False
22  # A list of the fields to be included in the packed record.
23  fields:
24      seq:
25          - type: map
26              mapping:
27                  # The name of the field.
28                  name:
29                      type: str
30                      required: True
31                  # A description for the field.
32                  description:
33                      type: str
34                      required: False
35                  # The type of the field.
36                  type:
37                      type: str
38                      required: True
39                  # If the field is a primitive Ada type, provide a format
40                  # string that describes its size in bits and its signedness when
41                  # packed. ie.
42                  # U2, U32, I16, F32, U16x20 etc.
43                  format:
44                      type: str
45                      required: False
46                  # Provide a default value for this field to be initialized to when
47                  # a record of this type is instantiated.
48                  default:
49                      type: str
50                      required: False
51                  # This field is an arrayed type and has a variable length. The current
52                  # length
53                  # of valid data in the field is determined by the field listed here.
54                  # For example
55                  # if another field in the record contains the usage length of this
56                  # field you might
57                  # input the name of that length field, ie. "Length" or
58                  # "Header.Secondary_Header.Length"
59                  # Note that variable length fields can only occur as the last field in
60                  # a packed record,
61                  # otherwise the record will violate the "overlayable" property of all
62                  # Adamant packed
63                  # records.
64                  variable_length:
65                      type: str
66                      required: False
67                  # If variable_length is set, this offset (either negative or positive
68                  # integer) will be
69                  # applied to the length field to calculate the actual length of the
70                  # variable field. So
71                  # if the length field is read as 10 and the offset is -1, then the
72                  # autocode will

```

```

63      # expect 9 values in the variable length field to be valid. By
64      # default, the offset is
65      # set to zero if not specified.
66      variable_length_offset:
67          type: int
68          required: False
69          # There is no predefined 'Image or autocoded .Image available for this
70          # field
71          # so instead just print the bytes as an array of unsigned_8 bit
72          # integers.
73      byte_image:
74          type: bool
75          required: False
76          default: False
77          # There is no way to validate this field, so don't try to autocode it.
78          skip_validation:
79          type: bool
80          required: False
81          default: False
82          # At least one field must be included in the record.
83      range:
84          min: 1
85      required: True

```

A.11 Packets Model Schema

YAML schema for model files of the form *component_name.packets.yaml*.

gen/schemas/packets.yaml:

```

1  ---
2  # This schema describes the yaml format for a packet suite.
3  type: map
4  mapping:
5      # Description of the packet suite.
6      description:
7          type: str
8          required: False
9      # Many "with" dependencies are automatically deduced and included by
10     # the generator. If you want to manually add a "with" statement, you
11     # can list the names of the packages here.
12     with:
13         seq:
14             - type: str
15             required: False
16     # List of packet items to include in the suite.
17     packets:
18         seq:
19             - type: map
20             mapping:
21                 # Name of the packet.
22                 name:
23                     type: str
24                     required: True
25                 # Description of the packet.
26                 description:
27                     type: str
28                     required: False
29                 # Identifier for the packet (in CCSDS this would be the APID).
30                 id:

```

```

31      type: int
32      required: False
33      # The type of the packet item. Using a packed record or packed
34      # array type is recommended here. If a type is not provided,
35      # the user must construct the packet themselves without
36      # a helper function. There will also be no generation of
37      # decommutation information for the ground system.
38      type:
39          type: str
40          required: False
41      # A packet suite must have at least one packet item.
42      range:
43          min: 1
44      required: True

```

A.12 Parameters Model Schema

YAML schema for model files of the form *component_name.parameters.yaml*.

gen/schemas/parameters.yaml:

```

1  ---
2  # This schema describes the yaml format for a parameters suite.
3  type: map
4  mapping:
5      # Description of the parameters suite.
6      description:
7          type: str
8          required: False
9      # List of parameters items to include in the suite.
10     parameters:
11         seq:
12             - type: map
13                 mapping:
14                     # Name of the parameter item.
15                     name:
16                         type: str
17                         required: True
18                     # Description of the parameter item.
19                     description:
20                         type: str
21                         required: False
22                     # The type of the parameter item. Using a packed record or packed
23                     # array type is recommended here.
24                     type:
25                         type: str
26                         required: True
27                     # The default value of the parameter. This is required.
28                     default:
29                         type: str
30                         required: True
31                 # A parameter suite must have at least one parameter item.
32                 range:
33                     min: 1
34                 required: True

```

A.13 Requirements Model Schema

YAML schema for model files of the form `component_name.requirements.yaml`.

`gen/schemas/requirement.yaml`:

```
1  ---
2  # This schema describes the yaml format for a requirement
3  type: map
4  mapping:
5      project:
6          type: str
7          required: False
8      description:
9          type: str
10         required: True
11     requirements:
12         seq:
13             - type: map
14                 mapping:
15                     # Text, the requirement text
16                     text:
17                         type: str
18                         required: True
19                     # Description of the requirement
20                     description:
21                         type: str
22                         required: False
23         range:
24             min: 1
25             required: True
```

A.14 Unit Test Model Schema

YAML schema for model files of the form `test_name.tests.yaml` or `component_name.tests.yaml`.

`gen/schemas/tests.yaml`:

```
1  ---
2  # This schema describes the yaml format for a unit test suite.
3  type: map
4  mapping:
5      # A description of the unit test suite.
6      description:
7          type: str
8          required: False
9      # Any useful handcode to include in the .ads file can be included here.
10     # You can think of this as inline Ada, which might be useful for declaring
11     # custom enum types and the like.
12     preamble:
13         type: str
14         required: False
15     # Many "with" dependencies are automatically deduced and included by
16     # the generator. If you want to manually add a "with" statement, you
17     # can list the names of the packages here.
18     with:
19         seq:
20             - type: str
21                 required: False
22             # A list of the tests to be included in the suite.
```

```

23   tests:
24     seq:
25       - type: map
26         mapping:
27           # The name of the test.
28           name:
29             type: str
30             required: True
31           # The description of the test and what it accomplishes.
32           description:
33             type: str
34             required: False
35           # A test suite must contain at least one test.
36           range:
37             min: 1
38             required: True

```

A.15 View Model Schema

YAML schema for model files of the form `name.assembly_name.view.yaml`.

`gen/schemas/view.yaml`:

```

1  ---
2  # This schema describes the yaml format for an assembly view.
3  type: map
4  mapping:
5    # A description describing what the view shows.
6    description:
7      type: str
8      required: False
9    # You can include a "dot" snippet here to be included at the beginning of the
10   ↵ digraph
11  preamble:
12    type: str
13    required: False
14  # You can include a "dot" snippet here to be included at the end of the
15   ↵ digraph
16  postamble:
17    type: str
18    required: False
19  layout:
20    type: str
21    enum: ['left-to-right', 'top-to-bottom', 'right-to-left', 'bottom-to-top']
22    required: False
23  # Show switches. Each of these turn on and off things on a view:
24  # By default, if not specified, these are all set to true.
25  show_component_type:
26    type: bool
27    required: False
28  show_component_execution:
29    type: bool
30    required: False
31  show_component_priority:
32    type: bool
33    required: False
34  show_component_name:
35    type: bool
36    required: False
37  show_connector_type:

```

```

36     type: bool
37     required: False
38 hide_group_outline:
39     type: bool
40     required: False
41 show_data_dependencies:
42     type: bool
43     required: False
44 # A rule which determines how the filters are applied to the assembly.
45 # If not provided, the assumed rule is to "and" all the filters together
46 # in order.
47 rule:
48     type: str
49     required: False
50 # A list of filters to be applied to the assembly model to
51 # limit what is shown in the view.
52 filters:
53     seq:
54         - type: map
55     mapping:
56         # a filter name is required
57         name:
58             type: str
59             required: True
60             # The type of filter to be applied.
61             # component_name - filter by component names
62             # component_type - filter by component types
63             # component_execution - filter by component execution types (active
64             # or passive)
65             # component_name_context - filter by component name and all it's
66             # immediate connections
67             # component_type_context - filter by component type and all it's
68             # immediate connections
69             # connector_name - filter by connector_names
70             # connector_type - filter by connector type
71             # connector_kind - filter by connector kinds (recv_async, recv_sync,
72             # send, request, service, get, return, provide, modify)
73             # data_dependency_name - filter by data_dependency or data_product
74             # pseudo-connector names
75             # data_dependency_type - filter by data_dependency pseudo-connector
76             # types
77         type:
78             type: str
79             enum: ['component_name', 'component_name_context',
80                 'component_type', 'component_type_context',
81                 'component_execution', 'connector_name', 'connector_type',
82                 'connector_kind', 'data_dependency_name',
83                 'data_dependency_type']
84             required: True
85             # Items to include in the view, ie. they are filtered in (white list).
86             include:
87                 seq:
88                     - type: str
89                         unique: True
90                     required: False
91                     # Items which are excluded from the view, ie. they are filtered out
92                     # (black list)
93             exclude:
94                 seq:
95                     - type: str
96                         unique: True

```

```

86     required: False
87     required: False

```

A.16 SPARK Prove Model Schema

YAML schema for model files of the form *all.prove.yaml*.

gen/schemas/prove.yaml:

```

1  ---
2  # This schema describes the yaml format for defining how GNATprove should be run
3  # within a
4  # directory. All files in a directory are run with the same GNATprove
5  # configuration.
6  type: map
7  mapping:
8      # A description for prove file.
9      description:
10         type: str
11         required: False
12         # The GNATprove "level" which configures the --level switch for
13         # GNATprove. 2 is default.
14         #
15         # --level=0 is equivalent to --prover=cvc4 --timeout=1 --memlimit=1000
16         # --steps=0 --counterexamples=off
17         # --level=1 is equivalent to --prover=cvc4,z3,altergo --timeout=1
18         # --memlimit=1000 --steps=0 --counterexamples=off
19         # --level=2 is equivalent to --prover=cvc4,z3,altergo --timeout=5
20         # --memlimit=1000 --steps=0 --counterexamples=on
21         # --level=3 is equivalent to --prover=cvc4,z3,altergo --timeout=20
22         # --memlimit=2000 --steps=0 --counterexamples=on
23         # --level=4 is equivalent to --prover=cvc4,z3,altergo --timeout=60
24         # --memlimit=2000 --steps=0 --counterexamples=on
25         #
26         level:
27             type: int
28             required: False
29             range:
30                 min: 0
31                 max: 4
32             # The GNATprove "mode" which configures the --mode switch for
33             # GNATprove. "gold" is default.
34             mode:
35                 type: str
36                 enum: ['check', 'check_all', 'flow', 'prove', 'all', 'stone', 'bronze',
37                     'silver', 'gold']
38                 required: False

```

A.17 Memory Map Model Schema

YAML schema for model files of the form *name.memory_map.yaml*.

gen/schemas/memory_map.yaml:

```

1  ---
2  # This schema describes the yaml format for a memory_map.
3  type: map
4  mapping:

```

```

5   # Description of the memory_map.
6   description:
7     type: str
8     required: False
9   # Any useful handcode to include in the .ads file can be included here.
10  # You can think of this as inline Ada, which might be useful for declaring
11  # custom enum types and the like.
12  preamble:
13    type: str
14    required: False
15  # Many "with" dependencies are automatically deduced and included by
16  # the generator. If you want to manually add a "with" statement, you
17  # can list the names of the packages here.
18  with:
19    seq:
20      - type: str
21      required: False
22  # The start address of this memory map region.
23  start_address:
24    type: int
25    required: True
26  # The length of this memory map region.
27  length:
28    type: int
29    required: True
30  # List of items to include in the memory map.
31  items:
32    seq:
33      - type: map
34      mapping:
35        # Name of the item.
36        name:
37          type: str
38          required: True
39        # Description of the item.
40        description:
41          type: str
42          required: False
43        # The type of the item. Using a packed record or packed
44        # array type is recommended here.
45        type:
46          type: str
47          required: True
48        # The alignment of an item. If not specified the default
49        # value is an alignment of 1, which means the item starts
50        # on a byte boundary.
51        #alignment:
52        # type: int
53        # required: False
54        # default: 1
55        # The value of the item. This is usually left unspecified, but may
56        # be required in special circumstances where you want to initialize
57        # some value in non-volatile storage at every boot.
58        value:
59          type: str
60          required: False
61  # A memory_map must have at least one item.
62  range:
63    min: 1
64    required: True

```