

Designing for Scalability with Erlang/OTP

IMPLEMENTING ROBUST,
FAULT-TOLERANT SYSTEMS

Early Release

RAW & UNEDITED



Francesco Cesarini & Steve Vinoski

Designing for Scalability with Erlang/OTP

Francesco Cesarini and Stephen Vinoski

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Designing for Scalability with Erlang/OTP

by Francesco Cesarini and Stephen Vinoski

Copyright © 2010 Francesco Cesarini and Stephen Vinoski. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Mike Loukides and Andy Oram

Indexer: FIX ME!

Production Editor: FIX ME!

Cover Designer: Karen Montgomery

Copyeditor: FIX ME!

Interior Designer: David Futato

Proofreader: FIX ME!

Illustrator: Rebecca Demarest

November 2014: First Edition

Revision History for the First Edition:

2014-05-07: Early release revision 1

2014-06-24: Early release revision 2

2015-04-09: Early release revision 2

See <http://oreilly.com/catalog/errata.csp?isbn=9781449320737> for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. !!FILL THIS IN!! and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-32073-7

[?]

Table of Contents

Preface.....	ix
1. Introduction.....	1
Defining the Problem	2
OTP	4
Erlang	5
Tools and Libraries	6
System Design Principles	8
Erlang Nodes	9
Distribution, Infrastructure and Multi-core	11
Summing Up	12
What You'll Learn in This Book	13
2. Introducing Erlang.....	19
Recursion and Pattern Matching	19
Functional Influence	23
Fun with Anonymous Functions	23
List Comprehensions: generate and test	24
Processes and Message Passing	26
Fail Safe!	29
Links and Monitors for Supervision	30
Links	31
Monitors	32
Records	34
Maps	36
Macros	37
Upgrading Modules	38
ETS: Erlang Term Storage	40
Distributed Erlang	43

Naming and Communication	43
Node Connections and Visibility	44
Summing Up	46
What's Next?	46
3. Behaviours.....	47
Process Skeletons	47
Design Patterns	50
Callback Modules	51
Extracting Generic Behaviours	54
Starting The Server	56
The Client Functions	58
The Server Loop	61
Functions Internal to the Server	62
The Generic Server	63
Message Passing: Under the hood	66
Summing up	69
What's Next?	70
4. Generic Servers.....	71
Generic Servers	71
Behaviour Directives	73
Starting a Server	73
Message Passing	75
Synchronous Message Passing	76
Asynchronous Message Passing	77
Other Messages	79
Using From	80
Unhandled Messages	81
Termination	83
Call Timeouts	84
Deadlocks	87
Generic Server Timeouts	89
Hibernating Behaviours	90
Going Global	91
Linking Behaviours	92
Summing Up	93
What's Next?	93
5. Controlling OTP Behaviours.....	95
The sys Module	95
Tracing and Logging	95

System Messages	96
Your Own Trace Functions	97
Statistics, Status, and State	98
The sys module Recap	101
Performance Tuning Options	102
Memory Usage	102
Options to Avoid	106
Summing Up	106
What's Next?	107
6. Finite State Machines.....	109
Finite State Machines the Erlang Way	110
Coffee Finite State Machine	112
The Hardware Stub	113
The Erlang Coffee Machine	113
Generic Finite State Machines	117
A Behaviour Example	118
Starting the FSM	118
Sending Events	122
Summing Up	132
Get Your Hands Dirty	133
The Phone Controllers	133
Let's Test It	135
What's Next?	137
7. Event Handlers.....	139
Events	139
Generic Event Managers and Handlers	141
Starting and Stopping Event Managers	141
Adding Event Handlers	142
Deleting an Event Handler	144
Sending Synchronous and Asynchronous Events	145
Retrieving Data	148
Handling Errors and Invalid Return Values	149
Swapping Event Handlers	152
Wrapping it all up	154
The SASL Alarm Handler	157
Summing Up	158
What's Next?	159
8. Supervisors.....	161
Supervision Trees	162

OTP Supervisors	166
The Supervisor Behaviour	167
Starting the Supervisor	168
The Supervisor Specification	171
Dynamic Children	178
Non OTP-Compliant Processes	185
Scalability and Short-Lived Processes	187
Synchronous Starts for Determinism	190
Testing Your Supervision Strategy	191
How Does This Compare?	192
Summing Up	193
What's Next?	194
9. Applications.....	195
How Applications Run	196
The Application Structure	198
The Callback Module	201
Starting Applications	202
Stopping Applications	203
Application Resource Files	205
The Base Station Controller Application File	208
Starting an Application	208
Environment Variables	211
Distributed Applications	214
Start Phases	217
Included Applications	219
Start Phases in Included Applications	219
Combining Supervisors and Applications	221
The SASL Application	222
Progress Reports	226
Error Reports	226
Crash Reports	227
Supervisor Reports	228
Summing Up	229
What's Next?	230
10. Special Processes and Your Own Behaviours.....	231
Special Processes	232
The Mutex	232
Starting Special Processes	234
The Mutex States	236
Handling Exits	237

System Messages	238
Trace and Log Events	240
Putting it Together	241
Dynamic Modules and Hibernating	244
Your Own Behaviours	245
Rules for creating behaviours	245
A Example Handling TCP Streams	246
Summing Up	249
What's Next?	250
11. System Principles and Release Handling.....	251
System Principles	251
Release Directory Structure	253
Release Resource Files	256
Creating a Release	259
Creating the boot file	260
Creating a Release Package	269
Start Scripts and Configuring on Target	273
Arguments and Flags	275
The init module	287
Other Tools	287
Rebar	288
Relx	297
Reltool	300
Wrapping Up	303
What's Next?	306
12. Release Upgrades.....	307
Software Upgrades	308
The First Version of the Coffee FSM	310
Adding a state	312
Creating a Release Upgrade	315
The Code To Upgrade	319
Application Upgrade Files	322
High-Level Instructions	325
Release Upgrade Files	327
Low-level instructions	330
Installing an Upgrade	331
The Release Handler	333
Upgrading Environment Variables	337
Upgrading Special Processes	337
Upgrading In Distributed Environments	338

Upgrading The Emulator and Core Applications	339
Summing Up	340
What's Next?	342

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Book Title* by Some Author (O'Reilly). Copyright 2011 Some Copyright Holder, 9781449373191.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online



Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at <http://my.safaribooksonline.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

<http://www.oreilly.com/catalog/9781449373191>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *<http://www.oreilly.com>*.

Find us on Facebook: *<http://facebook.com/oreilly>*

Follow us on Twitter: *<http://twitter.com/oreillymedia>*

Watch us on YouTube: *<http://www.youtube.com/oreillymedia>*

CHAPTER 1

Introduction

You need to implement a fault tolerant, scalable soft real time system with requirements for high availability. It has to be event driven and react to external stimulus, load and failure. It must always be responsive. You have heard, rightfully so, of many success stories telling you Erlang is the right tool for the job. And indeed it is—but while Erlang is a powerful programming language, it's not enough on its own to group these features all together and build complex reactive systems. To get the job done correctly, quickly and efficiently, you also need middleware, reusable libraries, tools, and design principles that tell you how to architect and distribute your system.

Our goal with this book is to explore multiple facets of scalability, as well as related topics such as concurrency, distribution, and fault tolerance, in the context of the Erlang programming language and its OTP framework. Erlang/OTP was created when the Ericsson Computer Science Laboratory (CSL) set out to investigate how they could efficiently develop the next generation of telecommunications systems in an industry where time to market was becoming critical. This was before the web, before tablets and smartphones, massively multiuser online gaming, messaging, and The Internet of Things.

At that time, the only systems that required the levels of scalability and fault tolerance we take for granted today were boring phone switches. They had to handle massive traffic spikes on New Year's Eve, fulfill regulatory obligations for the availability of calls to emergency services, and avoid painfully expensive contractual penalties forced on the infrastructure suppliers who caused outages. In layman's terms, if you picked up the phone and did not hear the dial tone on the other end, you could be sure of two things: top level management got into serious trouble and the outage would make the front page news in the papers. No matter what, those switches were not allowed to fail. Even when components were failing, requests had to be handled.

As a result, telecoms switches had to react to failure as much as they had to react to load and internal events. So while the Ericsson Computer Science Lab did not set out to invent a programming language, the solution to the problem they were out to solve

happened to be one. It's a great example of inventing a language that facilitates the task of solving specific, well-defined problems.

Defining the Problem

As we show throughout this book, Erlang/OTP is unique among programming languages and frameworks in the breadth, depth, and consistency of the features it provides for scalable, fault tolerant systems with requirements for high availability. Designing, implementing, operating and maintaining these systems is challenging. Teams that succeed in building and running them do so by continuously iterating through those four phases, constantly using feedback from production metrics and monitoring to help find areas they can improve not only in their code, but also in their development and operating processes. Successful teams also learn how to improve scalability from other sources such as testing, experimentation, benchmarking, and they keep up on research and development relevant to their system characteristics. Non-technical issues such as organizational values and culture can also play a significant part in determining whether teams can meet or exceed their system requirements.

We used the terms *distributed*, *fault tolerant*, *scalable*, *soft real time* and *highly available* to describe the systems we plan on building with OTP. But what do these words actually mean?

Scalable refers to how well a computing system can adapt to changes in load or available resources. Scalable websites, for example, are able to smoothly handle traffic spikes without dropping any client requests, even when hardware fails. A scalable chat system might be able to accommodate thousands of new users per day without disruption of the service it provides to its current users.

Distributed refers to how systems are clustered together and interact with each other. Clusters can be architected to scale horizontally by adding commodity (or regular) hardware, or on a single machine, where additional instances of stand-alone nodes are deployed to better utilize the available cores. Single machines can also be virtualized, so that instances of an operating system run on other operating systems or share the bare metal resources. Adding more processing power to a database cluster could enable it to scale in terms of the amount of data it can store or how many requests per second it can handle. Scaling downward is often equally as important; for example, a web application built on cloud services might want to deploy extra capacity at peak hour and release unused computing instances as soon as usage drops.

Systems that are *fault tolerant* behave predictably when things in their environment are failing. Fault tolerance has to be designed into a system from the start; don't even consider adding it as an afterthought. What if there is a bug in your code or your state gets corrupted? Or what if you experience a network outage or hardware failure. If a user is

sending a message causes a process to crash, the user is notified if the message was delivered or not, and he can be assured that the notification he or she gets is correct.

By *soft real time*, we mean the predictability of our system, handling a constant throughput and guaranteeing a response within an acceptable time frame. This throughput has to remain constant regardless of traffic spikes and number of concurrent requests. No matter how many simultaneous requests are going through the system, throughput must not degrade under heavy loads. Its response times, also known as latency, has to be relative to the number of simultaneous requests, avoiding large variances in requests caused by “stop the world” garbage collectors or other sequential bottlenecks. If your system throughput is a million messages per second and a million simultaneous requests happen to be processed, it should take a second to process and deliver a request to its recipient. But if during a spike, two million requests are sent, there should be no degradation in the throughput; not some, but all of the requests should be handled within two seconds.

High availability minimizes or completely eliminates downtime as a result of bugs, outages, upgrades or other operational activities. What if a process crashes? What if the power supply to your data center is cut off? Do you have a redundant supply or battery backup that gives you enough time to migrate your cluster and cleanly shut down the affected servers? Or network and hardware redundancy? Have you dimensioned your system ensuring that, even after losing part of your cluster, the remaining hardware has enough CPU capacity to handle peak loads? It does not matter if you lose part of your infrastructure, or your cloud provider is experiencing an embarrassing outage or you are doing maintenance work; a user sending a chat message wants to be reassured that it reaches its intended recipient. They expect it to just work. This is in contrast to *fault tolerance*, where the user is told it did not work, but the system itself is unaffected and continues to run. Erlang’s ability to do software upgrades during runtime helps. But if you start thinking of what is involved when dealing with database schema changes, or upgrades to non-backward-compatible protocols in potentially distributed environments handling requests during the upgrade, simplicity fades very quickly. When doing your online banking on weekends or at night, you want to be sure you will not be met with an embarrassing “closed for routine maintenance” sign posted on the website.

Erlang indeed facilitated solving many of these problems. But at the end of the day, it is still just a programming language. For the complex systems you are going to implement, you need ready-built applications and libraries you can use out of the box. You also need design principles that inform the architecture of your system with an aim to create distributed, reliable clusters, and guidelines on how to design your system, together with tools to implement, deploy, operate and maintain it. Thus, although picking the right programming language will make life easier and increase your productivity, you need much more than just a language: you need a library for distributed computing, together with architectural design patterns. OTP is that library for Erlang. You need to think hard about your requirements and properties, ensuring you pick the right libraries

and design patterns which ensure the final system behaves the way you want it and does what you originally intended. No ready made library can help you if you do not know what you want to get out of your system.

OTP

OTP, in brief, is a domain-independent set of frameworks, principles, and patterns that guide and support the structure, design, implementation, and deployment of Erlang applications.

Using OTP in your projects will help you avoid accidental complexity: things that are difficult because you picked inadequate tools. But other problems remain difficult, irrespective of programming tools and middleware you choose. You want to avoid accidental difficulties that have already been solved, and instead focus your energy on the hard problems.

Ericsson realized this very early on. In 1993, alongside the development of the first Erlang product, Ericsson started a project to tackle tools, middleware and design principles. The result was BOS, the Basic Operating System. In 1995, it was merged with the development of Erlang, bringing everything under one roof to form Erlang/OTP as we know it today. You might have heard of the dream team that supports Erlang being referred to as the OTP team. This group was a spin-off of this merge, when Erlang was moved out of a research organization and a product group was formed to further develop and maintain it.

Spreading knowledge of OTP can promote Erlang adoption in more “tried and true” corporate IT environments. Just knowing there is a stable and mature platform available for application development helps technologists sell Erlang to management, a crucial step in making its industrial adoption more widespread. Startups, on the other hand, just get on with it, with Erlang/OTP allowing them to achieve speed to market and reduce their development and operations costs.

OTP is said to consist of three building blocks ([Figure 1-1](#)) which, when used together, provide a solid approach to designing and developing systems in the problem domain we've just described. They are Erlang itself, tools and libraries, and a set of design principles. We'll look at each in turn.

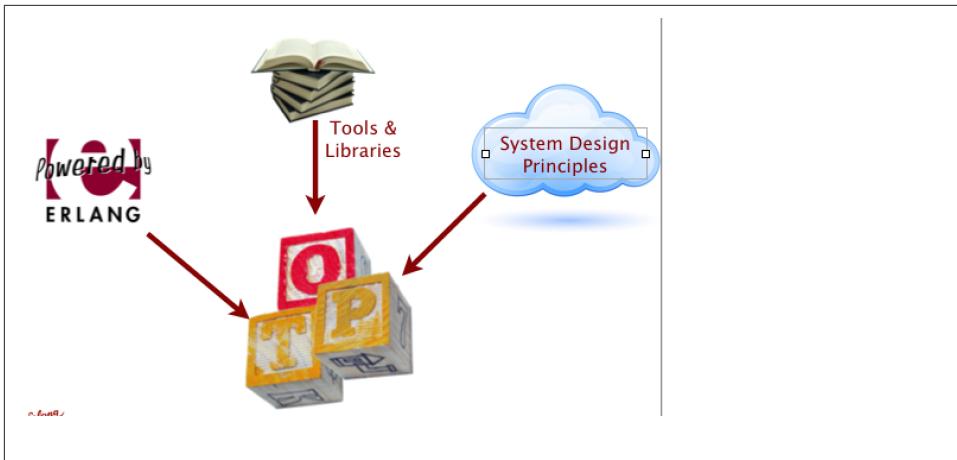


Figure 1-1. OTP Components

Erlang

The first building block is Erlang itself, which includes the semantics of the language and its underlying virtual machine. Key language features such as lightweight processes, lack of shared memory, and asynchronous message passing will bring you a step closer to your goal. Just as important are links and monitors between processes, and dedicated channels for the propagation of the error signals. The monitors and error reporting allow you to build, with relative ease, complex supervision hierarchies with built-in fault recovery. Because message passing and error propagation are asynchronous, the semantics and logic of a system that were developed to run in a single Erlang node can be easily distributed without having to change any of the code base.

One significant difference between running on a single node and running in a distributed environment is the latency with which messages and errors are delivered. But in soft real time systems, you have to consider latency regardless of whether the system is distributed or not. Erlang lets you run all of this on top of a virtual machine highly optimized for concurrency, with a per-process garbage collector, yielding a predictable and simple system behaviour.

Other programming environments do not have this luxury because they need an extra layer to emulate Erlang's concurrency model and error semantics. To quote Joe Armstrong, co-inventor of Erlang, "You can emulate the logic of Erlang, but if it is not running on the Erlang virtual machine, you cannot emulate the semantics". The only languages that today get away with this are built on the BEAM emulator, the prevailing Erlang virtual machine. There is a whole ecosystem of them, and at the time of writing, the Elixir and Lisp Flavoured Erlang languages seem to be gaining the most traction.

Tools and Libraries

The second building block, which came about before open source became the widespread norm for software projects, includes applications that ship as part of the standard Erlang/OTP distribution. You can view each application as a way of packaging resources in OTP, and applications may depend on other applications. The applications include tools, libraries, interfaces towards other languages and programming environments, databases and database drivers, standard components and protocol stacks. The OTP documentation does a fine job in separating them into the following subsets:

- The basic applications include the following:
 - the Erlang runtime system (*erts*)
 - the *kernel*
 - the standard libraries (*stdlib*)
 - the system application support libraries (*sasl*)

They include tools and basic building blocks needed to architect, create, start and upgrade your system. We will be covering the basic applications in detail throughout this book. Together with the compiler, they form the minimal subset of applications needed in any system written in Erlang/OTP needed to do anything meaningful.

- The database applications include *mnesia*, Erlang's distributed database, and *odbc*, an interface used to communicate with relational SQL databases. Mnesia is a popular choice because it is fast because it runs and stores its data in the same memory space as your applications, and easy to use, as it is accessed through an Erlang API.
- The operations and maintenance applications include *os_mon*, an application that allows you to monitor the underlying operating system, a Simple Network Management Protocol (*snmp*) agent and client, and *otp_mibs*, management information bases that allow you to manage Erlang systems using SNMP.
- The collection of interface and communication applications provide protocol stacks and interfaces to work with other programming languages, including an ASN.1 (*asn1*) compiler and runtime support, direct hooks into C (*ei* and *erl_interface*) and Java (*jinterface*) programs, along with an XML parser (*xmerl*). Security applications include SSL/TLS, SSH, crypto, and public key infrastructure. Graphics packages include a port of wxWidgets (*wx*), together with an easy to use interface. The *eldap* application provides a client interface towards the Lightweight Directory Access Protocol (LDAP). And for telecom aficionados, there is a *Diameter* stack (as defined in RFC 6733), used for policy control and authorisation, alongside authentication and accounting. Dig even deeper and you will find the *megaco* stack. Megaco/H.248 is a protocol for controlling elements of a physically decomposed multimedia gateway, separating the media conversion from the call control. If you have ever

used a smart phone, you have very likely indirectly taken the Erlang diameter and megaco applications for a spin.

- The collection of tools applications facilitate the development, deployment, and management of your Erlang system. We cover only the most relevant ones in this book, but outline them all here so you are aware of their existence.
 - The *debugger* is a graphical tool that allows you to step through your code whilst influencing the state of the functions.
 - The *observer* integrates the application monitor and the process manager, alongside basic tools to monitor your Erlang systems as they are being developed and in production.
 - *Dialyzer* is a static analysis tool that finds type discrepancies, dead code, and other issues.
 - The event tracer (*et*) uses ports to collect trace events in distributed environments and *percept* allows you locate bottlenecks in your system by tracing and visualizing concurrency-related activities.
 - Erlang Syntax Tools (*syntax_tools*) contains modules for handling Erlang syntax trees in a way that is compatible with other language-related tools. It also includes a module merger allowing you to merge Erlang modules into a single one, together with a renamer, solving the issue of clashes in a non-hierarchical module space.
 - The *parsetools* application contains the parse generator (*yacc*) and a lexical analyzer generator for Erlang (*lex*).
 - *Reltool* is a release management tool that provides a graphical front-end together with back-end hooks that can be used by more generic build systems.
 - *Runtime_tools* is a collection of utilities including DTrace and SystemTap probes, and *dbg*, a user-friendly wrapper around the trace BIFs.
 - Finally, the *tools* application is a collection of profilers, code coverage, and module cross-reference analysis tools, as well as the Erlang mode for emacs.
- The test applications provide tools for unit testing (*eunit*), system testing, and black box testing. The Test Server (packaged in the *testserver* application) is a framework that can be used as the engine of a higher level test tool application. Chances are that you will not be using it, because OTP provides one of these higher level test tools in the form of *common_test*, an application suited for black box testing. *common_test* supports automated execution of Erlang-based test cases towards most target systems irrespective of programming languages.
- We need to mention the Object Request Brokers and IDL applications for nostalgic reasons, reminding one of the co-authors of his past sins. They include a broker

called *orber*, an IDL compiler called *ic*, and a few other CORBA Common Object Services no longer used by anyone.

We cover and refer to some of these applications and tools in this book. Some of the tools we do not cover are described in *Erlang Programming* (O'Reilly), and those that aren't have a set of reference manual pages and a user's guide that comes as part of the standard Erlang/OTP documentation.

These applications are not the full extent of tool support for Erlang; they are enhanced by thousands of other applications implemented and supported by the community and available as open source. Once you've read this book and before starting your project, review the standard Erlang/OTP reference manuals and user's guides, because you never know when they will come in handy.

System Design Principles

The third building block of OTP consists of a set of abstract principles, design rules, and generic behaviours. The abstract principles describe the software architecture of an Erlang system, using processes in the form of generic behaviours as basic ingredients. Design rules keep the tools you use compatible with the system you are developing. Using this approach provides a standard way of solving problems, making code easier to understand and maintain, as well as providing a common language and vocabulary among the teams.

OTP generic behaviours can be seen as formalisations of concurrent design patterns. Behaviours are packaged into library modules containing generic code that solves a common problem. They have built-in support for debugging, software upgrade, generic error handling, and built-in functionality for upgrades.

Behaviours can be worker processes, which do all of the hard work, and supervisor processes, whose only task is to monitor workers or other supervisors. Because supervisors can monitor other supervisors, the functionality within an application can be chained so that it can be more easily developed in a modular fashion. The processes monitored by a supervisor are called its children.

OTP provides predefined libraries for workers and supervisors, allowing you to focus on the business logic of the system. We structure processes into hierarchical supervision trees, yielding fault tolerant structures that isolate failure and facilitate recovery. OTP allows you to package a supervision tree into an application, as seen in [Figure 1-2](#).

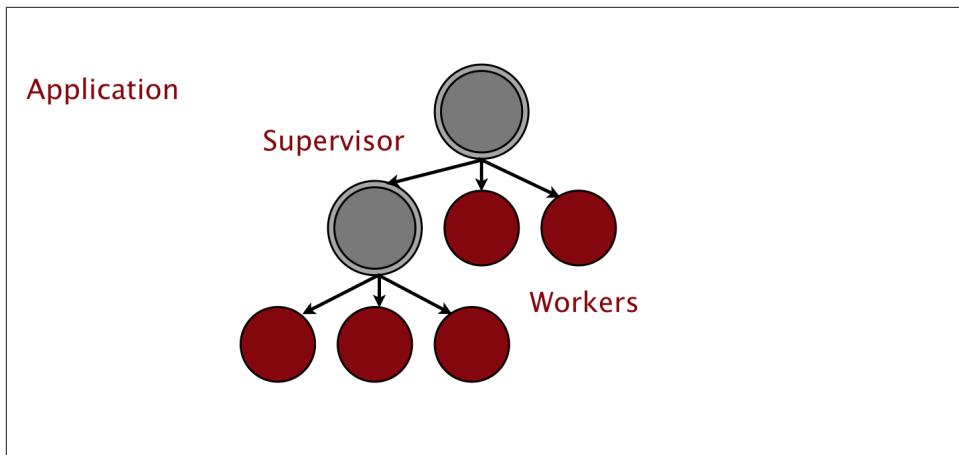


Figure 1-2. OTP Application

Generic behaviours that come as part of the OTP middleware include:

- Generic Servers, providing a client-server design pattern
- Generic Finite State Machines, allowing you to implement FSMs
- Event Handlers and Managers, allowing you to generically deal with event streams
- Supervisors, monitoring other worker and supervision processes
- Applications, allowing you to package resources, including supervision trees

We cover them all in detail in this book, as well as explain how to implement your own.

Erlang Nodes

An Erlang node consists of several loosely coupled applications, which might consist of some of the applications described in “Tools and Libraries” on page 6 combined with other third-party applications and applications written by you specific to the system you are trying to implement. These applications could be independent of each other or rely on the services and APIs of other applications. Figure 1-3 illustrates a typical release of an Erlang node with the VM dependent on the hardware and operating system, Erlang applications running on top of the VM interfacing non-Erlang components which are OS and hardware dependent.

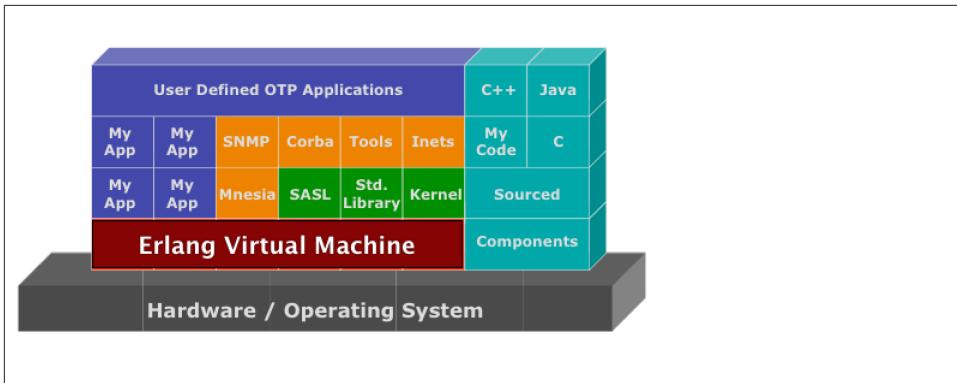


Figure 1-3. An Erlang node

Group together a cluster of Erlang nodes—potentially pairing them up with nodes written in other programming languages—and you have a distributed system. You can now scale your system by adding nodes until you hit certain physical limits. These may be dictated by how you shared your data, by hardware or network constraints, or by external dependencies that act as bottlenecks.

What's In a Name?

What does OTP stand for? We'd rather not tell you. If you search for the “OTP song” you might be led to believe it means *One True Pair*. Or let your imagination go wild, and guess *Oh This is Perfect, On The Phone* or *Open Transaction Platform*. Some might think OTP stands for *Online Transaction Processing* but that's normally abbreviated as OLTP. More politically incorrect suggestions have also been made when hipsters were enlisted in an attempt to make Erlang more cool in the Erlang the Movie sequel. Alas, none of these is correct. OTP is short for Open Telecom Platform, a name coined by Bjarne Däcker, head of the Computer Science lab (the birthplace of Erlang) at Ericsson.

Open was a buzzword at Ericsson in the mid-90s, believing everything had to be open: open systems, open hardware, open platforms. Ericsson's marketing department went as far as to print posters of open landscapes, hanging them in the corridors with the text Open Systems. No one really understood what was meant by Open Systems (or any of the other openness), but it was a buzzword, so why disappoint and not jump on an opportunity and (for once) be buzzword compliant? As a result, the Open in OTP became a no brainer.

Today, we say that Open stands for the openness of Erlang towards other programming languages, APIs, and protocols, a far cry from the openness of the days it was first released. OTP R1 was in fact everything but open. Today, think of openness being about JInterface, ei and erl_interface, HTTP, TCP/IP, UDP/IP, IDL, ASN1, CORBA, SNMP, and other integration-oriented support provided by Erlang/OTP.

Telecom was chosen when Erlang was used only internally within Ericsson for telecom products, long before open source would change the world. It might have made sense in the mid-90s, but no rebranding ever took place, so today we say that the Telecom in the name refers to the distributed, fault tolerant, scalable, soft real-time characteristics with requirements of high availability. These are characteristics present in telecom systems, but equally valid in a wide range of other verticals. The developers of OTP were solving a problem for telecom systems that became relevant to the rest of the software industry only when the Web was invented and everything had to be webscale. Erlang was webscale even before the web itself!

Platform, whilst boring, is the only word truly describing the OTP middleware. It was chosen at a time when Ericsson management was going over the top developing a variety of platforms. Everything software related had to be developed on a (preferably open) platform.

So indeed, Bjarne picked an acronym that made sense and would keep higher management happy, ensuring they kept on funding the project. They might not have understood what we were working on and the trouble we were all about to cause, but at least, they were pleased and allowed it all to happen.

Since Erlang/OTP was released as open source in 1998, many discussions on rebranding have taken place, but none of them were conclusive. In the early days, developers outside of the Telecoms sector mistakenly bypassed OTP, because—using their own words—“they were not developing Telecom applications”. The community and Ericsson have today settled for using OTP, toning down the Telecom, but stressing its importance. This seems to be a fair compromise. In this book, this sidebar will be the only place where Telecom will be mentioned as being part of OTP.

Distribution, Infrastructure and Multi-core

Fault tolerance—one of Erlang’s fundamental requirements from its telecom roots—has distribution as its mainspring. Without distribution, the reliability and availability of an application running on just a single host would depend heavily on the reliability of the hardware and software comprising that host. Any problems with the host’s CPU, memory, persistent storage, peripherals, power supply or backplane could easily take down the entire machine and the application along with it. Similarly, problems in the host’s operating system or support libraries could bring down the application or otherwise render it unavailable. Achieving fault tolerance requires multiple computers with some degree of coordination between them, and distribution provides the avenue for that coordination.

For decades, the computing industry has explored how programming languages can support distribution. Designing general-purpose languages is difficult enough; designing them to support distribution significantly adds to that difficulty. Because of this, a common approach is to add distribution support to non-distributed programming

languages through optional libraries. This approach has the benefit of allowing distribution support to evolve separately from the language itself, but it often suffers from an impedance mismatch with the language, feeling to developers as if it were “bolted on”. Since most languages use function calls as the primary means of transferring control and data from one part of an application to another, add-on distribution libraries often model exchanges between distributed parts of an application as function calls as well. While convenient, this approach is fundamentally broken because the semantics of local and remote function calls, especially their failure modes, are markedly different.

In Erlang, processes communicate via asynchronous message passing. This works even if a process is on a remote node because the Erlang virtual machine supports passing messages from one node to another. When one node joins another, it also becomes aware of any nodes already known to the other. In this manner, all the nodes in a cluster form a mesh, enabling any process to send a message to another process on any other node in the cluster. Each node in the cluster also automatically tracks liveness of other nodes in order to become aware of non-responsive nodes. The advantages of asynchronous message passing in systems running on a node is extended to systems running in clusters, as replies can be received alongside errors and timeouts.

Erlang’s message passing and clustering primitives can serve as the basis for a wide variety of distributed system architectures. For example, service-oriented architecture (SOA) is a natural fit for Erlang given the ease of developing and deploying server-like processes. Clients treat such processes as services, communicating with them by exchanging messages. As another example, consider that Erlang clusters do not require master or leader nodes, which means that using them for peer-to-peer systems of replicas works well. Clients can send service request messages to any peer node in the cluster, and the peer can either handle the request itself or route it to another peer.

Message passing and clustering also support Erlang’s ability to scale. Erlang’s virtual machine takes advantage of today’s multi-core systems by allowing processes to execute with true concurrency, running simultaneously on different cores. Because of the symmetric multiprocessing (SMP) capabilities of the Erlang virtual machine, Erlang is already prepared to help applications scale vertically as the number of cores per CPU continues to increase. And because adding new nodes to a cluster is easy—all it takes is to have that node contact just one other node to join the mesh—horizontal scaling is also well within reach. Erlang’s distribution features mean that adding a new peer to a peer-to-peer system, for example, is trivial from an Erlang perspective.

Summing Up

To make design, implementation, operation, and maintainability easier and more robust, your programming language and middleware have to be compact, their behaviour in runtime predictable, and the resulting code base maintainable. We keep talking about fault tolerant, scalable soft real-time system with requirements for high availability. The

problems you have to solve do not have to be complicated in order to benefit from the advantages Erlang/OTP brings to the table. Advantages will be evident if you are developing solutions targeted for embedded hardware platforms such as the ParallelA board, the Beagleboard or the Raspberry Pi. You will find Erlang/OTP ideal for the orchestration code in embedded devices, for server-side development where concurrency comes in naturally, and all the way to scalable and distributed multi-core architectures and supercomputers. They ease the development of the harder software problems while making simpler programs even easier to implement.

What You'll Learn in This Book

This book is divided into two sections. The first part covers [Chapter 3](#) to [Chapter 10](#), dealing with the design and implementation of a single node. You should read these chapters sequentially, because their examples and explanations build on prior ones. The remaining half of the book is focused on tools, techniques and architectures used for deployment, monitoring, and operations, while explaining how to embed features such as reliability, scalability, and high availability into the architecture. The second half builds on the examples covered in the first half of the book. When you get to the second half, you can pick and choose what to read in any order.

We begin [Chapter 2](#) with an overview of Erlang, intended not to teach you the language but rather as a refresher course. If you do not yet know Erlang, we recommend that you first consult one or more of the excellent books designed to help you learn the language, such as Simon St. Laurent's *Introducing Erlang*, *Erlang Programming* by Cesarini and Thompson, or any of the other books we mention in [Chapter 2](#). Our overview touches on the major elements of the language, such as lists, functions, processes and messages, and the Erlang shell, as well as those features that make Erlang unique among languages such as process linking and monitoring, live upgrades, and distribution.

Following the Erlang overview, [Chapter 3](#) dives into process structures. Erlang processes can handle a wide variety of tasks, yet regardless of the particular tasks or their problem domains, similar code structures and process life cycles surface, akin to the common design patterns that have been observed and documented for popular object-oriented languages like Java and C++. OTP captures and formalizes these common process-oriented structures and life cycles into *behaviours*, which serve as the base elements of OTP's reusable frameworks.

In [Chapter 4](#) we explore in detail our first worker process. It is the most popular and frequently used OTP behaviour, the `gen_server`. As its name implies, it supports generic client-server structures, with the server governing particular computing resources—perhaps just a simple Erlang Term Storage (ets) instance, or a pool of network connections to a remote non-Erlang server—and granting client access to them. Clients communicate with generic servers synchronously in a call-response fashion, asynchronously via a one-way message called a cast, or via regular Erlang messaging primitives. Full

consideration of these modes of communication requires us to scrutinize various aspects of the processes involved, such as what happens if the client or server dies in the middle of a message exchange, how timeouts apply, and what might happen if a server receives a message it does not understand. By addressing these and other common issues, the `gen_server` handles a lot of details independently of the problem domain, allowing developers to focus more of their time and energy on their applications. The `gen_server` behaviour is so useful it appears not only in most non-trivial Erlang applications, but is used throughout OTP itself as well.

Prior to examining more OTP behaviours, we follow our discussion of `gen_server` with a look at some of the control and observation points the OTP behaviours provide ([Chapter 5](#)). These features reflect another aspect of Erlang/OTP that sets it apart from other languages and frameworks: built-in observability. If you want to know what your `gen_server` process is doing, you can simply enable debug tracing for that process, either at compile time or at run time from an Erlang shell. Enabling traces causes it to emit information that indicates what messages it is receiving and what actions it is taking to handle them. Erlang/OTP also provides functions for peering into running processes to see their backtraces, process dictionaries, parent processes, linked processes, and other details. There are also OTP functions for examining status and internal state specifically for behaviour and other system processes. Because of these debug-oriented features, Erlang programmers often forego the use of traditional debuggers and instead rely on tracing to help them diagnose errant programs, as it is typically both faster to set up and more informative.

We then examine another OTP behaviour, `gen_fsm` ([Chapter 6](#)), which supports generic finite state machines. As you may already know, a finite state machine is a system that has a finite number of states, and incoming messages can advance the system from one state to another, with side effects potentially occurring as part of the transitions. For example, you might consider your television set top box as being an FSM where the current state represents the selected channel and whether any on-screen display is shown. Pressing buttons on your remote causes the set top box to change state, perhaps selecting a different channel, or changing its on-screen display to show the channel guide or list any on-demand shows that might be available for purchase. FSMs are applicable to a wide variety of problem domains because they allow developers to more easily reason about and implement the potential states and state transitions of their applications. Knowing when and how to use `gen_fsm` can save you from trying to implement your own ad hoc state machines, which often quickly devolve into spaghetti code that is hard to maintain and extend.

Logging and monitoring are critical parts of any scalability success story, since they allow you to glean important information about your running systems that can help pinpoint bottlenecks and problematic areas that require further investigation. The Erlang/OTP `gen_event` behaviour ([Chapter 7](#)) provides support for subsystems that emit and manage event streams reflecting changes in system state that can impact operational

characteristics, such as sustained increases in CPU load, queues that appear to be growing without bound, or the inability of one node in a distributed cluster to reach another. These streams do not have to stop with your system events. They could handle your application specific events originating from user interaction, sensor networks or third party applications. In addition to exploring the `gen_event` behaviour, we also take a look at the OTP System Application Support Libraries (SASL) error logging event handlers, which provide flexibility for managing supervisor reports, crash reports, and progress reports.

Event handlers and error handlers are staples of numerous programming languages, and they are incredibly useful in Erlang/OTP as well, but do not let their presence here fool you: dealing with errors in Erlang/OTP is strikingly different from the approaches to which most programmers are accustomed.

After `gen_event`, the next behaviour we study is the `supervisor` ([Chapter 8](#)), which manages worker processes. In Erlang/OTP, supervisor processes start workers and keep an eye on them while they carry out application tasks. Should one or more workers die unexpectedly, the supervisor can deal with the problem in one of several ways that we explain later in the book. This form of handling errors, known as “let it crash,” differs significantly from the defensive programming tactics that most programmers employ. “Let it crash” and supervision, together a critical cornerstone of Erlang/OTP, are highly effective in practice.

We then look into the final fundamental OTP behaviour, the `application` ([Chapter 9](#)), which serves as the primary point of integration between the Erlang/OTP runtime and your code. OTP applications have configuration files that specify their names, versions, modules, the applications upon which they depend, and other details. When started by the Erlang/OTP runtime, your `application` instance in turn starts a top-level supervisor that brings up the rest of the application. Structuring modules of code into applications also lets you perform code upgrades on live systems. A release of an Erlang/OTP package typically comprises a number of applications, some of which are part of the Erlang/OTP open source distribution and others that you provide.

Having examined the standard behaviours, we next turn our attention to explaining how to write your own behaviours and special processes ([Chapter 10](#)). Special processes are processes that follow certain design rules, allowing them to be added to OTP supervision trees. Knowing these design rules can not only help you understand implementation details of the standard behaviours, but can also inform you of their trade-offs and allow you to better decide when to use them and when to write your own instead.

The System Principles section in [Chapter 11](#) describes how Erlang applications are coupled together in a release and started as a whole. You will have to create your own release files, referred to in the Erlang world as *rel* files. It lists the versions of the applications and the runtime system that are used by the `systools` module to bundle up the software into a standalone release directory that includes the virtual machine. This

release directory, once configured and packaged, is ready to be deployed and run on target hosts. Tools we cover that help you build releases include the *RelTool*, included as part of the OTP distribution, alongside community contributed tools such as *rebar* and *relx*.

The Erlang virtual machine has configurable system limits and settings you need to be aware of when deploying your systems. There are many, ranging from limits regulating the maximum number of ets tables or processes to included code search paths and modes used for loading modules. Modules in Erlang can be loaded at startup, or when they are first called. In systems with strict revision control, you will have to run them in *embedded* mode, loading modules at startup, and crashing if modules do not exist, or in *interactive* mode, where if a module is not available, an attempt to load it is made before terminating the process. An external monitoring *heart* process monitors the Erlang virtual machine by sending heartbeats and invoking a script that allows you to react when these heartbeats are not acknowledged. You implement the script yourself, allowing you to decide whether restarting the node is enough, or whether—based on a history of previous restarts—you want to escalate the crash and terminate the virtual instance or reboot the whole machine.

Although Erlang's dynamic typing allows you to upgrade your module at runtime while retaining the process state, it does not coordinate dependencies among modules, changes in process state, or non-backwards-compatible protocols. OTP has the tools to support system upgrades on a system level, including not only the applications, but also the runtime system. The principles and supporting libraries are presented in [Chapter to Come], from defining your own application upgrade scripts to writing scripts that support release upgrades. Approaches and strategies in handling changes to your database schema are provided, as are guidelines for upgrades in distributed environments and non-backward-compatible protocols. For major upgrades in distributed environments where bugs are fixed, protocols improved, and database schema changed, runtime upgrades are not for the faint of heart. They are however incredibly powerful, allowing automated upgrades and nonstop operations. Finding your online banking is unavailable because of maintenance should now be a thing of the past. If it isn't, send a copy of this book to your bank's IT department.

Operating and maintaining Erlang systems requires visibility into what is going on. Scaling clusters require strategies on how you share your data. And fault tolerance requires an approach on how you replicate and persist it. While each of these subjects merits a book of its own, [Chapter to Come] tries to provide an overview of the approaches you can take when designing your nodes and clustering your systems. The foundations to monitoring systems include system and business *metrics*. Examples include the amount of memory your node is using, process message queue length, and hard disk utilisation. Combining these with business metrics, such as the number of failed and successful login attempts, message throughput per second, and session duration, yields full visibility of how your business logic is affecting your system resources.

Complement this with *alarming*, where you detect and report anomalies, allowing the system to take action to trying to resolve them or to alert an operator where human intervention is required. Alarms could include a system running out of disk space (resulting in the automatic invocation of scripts for compressing or deleting logs) or a large number of failed message submissions (requiring human intervention to troubleshoot connectivity problems). Pre-emptive support at its best, detecting and resolving issues before they escalate, is a must when dealing with high availability. If you do not have a real time view of what is going on, resolving issues before they escalate becomes extremely difficult and cumbersome.

And finally, *logging* of major events in the system helps you troubleshoot your system after a crash where you lost its state, so you can retrieve the call flow of a particular request among millions of others to handle a customer services query, or just provide data records for billing purposes.

Once you have the necessary visibility into your system operations, you need to decide how to cluster your system and distribute your data. Are you going for the share-everything, share-something, or share-nothing architecture? The applicability of each approach depends on your system's requirements for fault tolerance and scalability, as trade-offs will have to be made. We conclude [Chapter to Come] by looking at the most common architectural patterns, discussing how nodes can communicate with each other based on the system requirements.

CHAPTER 2

Introducing Erlang

This book assumes a basic knowledge of Erlang, which is best obtained through practice and by reading some of the many excellent introductory Erlang books out there (including two written for O'Reilly). But for a quick refresher, this chapter gives you an overview of important Erlang concepts. We draw attention particularly to those aspects of Erlang that you'll need to know when you come to learn OTP.

Recursion and Pattern Matching

Recursion is the way Erlang programmers get iterative or repetitive behaviour in their programs. It is also what keeps processes alive in between bursts of activity. Our first example shows how to compute the factorial of a positive number.

```
-module(ex1).
-export([factorial/1]).

factorial(0) ->
    1;
factorial(N) when N > 0 ->
    N * factorial(N-1).
```

We call the function `factorial` and indicate that it takes a single argument (`factorial/1`). The trailing `/1` is the *arity* of a function, and simply refers to the number of arguments the function takes—in our example, 1.

If the argument we pass to the function is the integer 0, we match the first clause, returning 1. Any integer greater than 0 is bound to the variable `N`, returning the product of `N` and `factorial(N-1)`. The iteration will continue until we pattern match on the function clause which is the base case. The base case is the clause where recursing stops. If we call `factorial/1` with a negative integer, the call fails as none of the clauses match. But we don't bother dealing with the problems caused by a caller passing a non-integer argument; this is an Erlang principle we'll discuss later.

Erlang definitions are contained in modules, which are stored in files of the same name, but with a *.erl* extension. So the filename of the above module would be *ex1.erl*. Erlang programs can be evaluated in the Erlang shell, invoked by the command `erl` in your Unix shell or by double-clicking on the Erlang icon. Make sure that you start your Erlang shell in the same directory as your source code. A typical session goes like this:

```
% erl                                         Comments on interactions are given in this format.
Erlang R16B03 (erts-5.10.4) [source] [64-bit] [smp:8:8] [async-threads:10] [hipe] [kernel-poll:fal

Eshell V5.10.4 (abort with ^G)
1> c(ex1).
{ok,ex1}
2> ex1:factorial(3).
6
3> ex1:factorial(-3).
** exception error: no function clause matching ex1:factorial(-3)
4> factorial(2).
** exception error: undefined shell command factorial/1
5> q().
ok
%
```

In shell command 1, we compile the Erlang file. We go on to do a fully qualified function call in command line 2, where by prefixing the module name to the function, we are able to invoke it from outside the module itself. The call in shell command 3 fails with a function clause error because none of the clauses match for negative numbers. Before terminating the shell with the shell command `q()`, we call a local function `factorial(2)` in shell command 4. It fails as it is not fully qualified with a module name.

Recursion is not just for computing simple values: we can write imperative programs using the same style. Here's a program to print every element of a list, separated by tabs. As with the previous example, the function is presented in two *clauses*, where each clause has a head and a body, separated by the arrow. In the head we see the function applied to a pattern, and when a function is applied to an argument, the first clause whose pattern matches the argument is used. In this example the `[]` matches an empty list, whereas `[X|Xs]` matches a non-empty list. The `[X|Xs]` syntax assigns the first element of the list, or *head*, to `X` and the remainder of the list, or *tail*, to `Xs`). If you have not yet noted it, Erlang variables such as `X`, `Xs`, and `N` all start with uppercase letters.

```
-module(ex2).
-export([print_all/1]).

print_all([]) ->
    io:format("~n");
print_all([X|Xs]) ->
    io:format("~p\t",[X]),
    print_all(Xs).
```

The effect is to print each item from the list, in the order that it appears in the list, with a tab (\t) after each item. Thanks to the base case, which runs when the list is empty (when it matches []), a newline (~n) is printed at the end. Unlike the ex1:factorial/1 example shown earlier, the pattern of recursion in this example is *tail recursive*. It is used in Erlang programs to give looping behaviour. A function is said to be tail recursive if the only recursive calls to the function occur as the last expression to be executed in the function clause. We can think of this final call as a “jump” back to the start of the function, now called with a different parameter. Tail recursive functions allow *last call optimization*, ensuring stack frames are not added in each iteration. This allows functions to execute in constant memory space and removes the risk of a stack overflow, which in Erlang manifests itself through the virtual machine running out of memory.

If you come from an imperative programming background, writing the function slightly differently to use a `case` expression rather than separate clauses may make tail recursion easier to understand.¹

```
all_print(Ys) ->
  case Ys of
    [] -> io:format("~n");
    [X|Xs] -> io:format("~p\t",[X]),
      all_print(Xs)
  end.
```

When you test either of these print functions, note the `ok` printed out after the new line. Every Erlang function has to return a value. This value is whatever the last executed expression returns. In our case, the last executed expression is `io:format(~n)`: the new line appears as a side-effect of the function, while the `ok` is the return value printed by the shell.

```
1> c(ex2).
{ok,ex2}
2> ex2:print_all([one,two,three]).
one      two      three
ok
3> Val = io:format("~n").

ok
4> Val.
ok
```

The arguments in our example play the role of *mutable variables*, whose values change between calls. Erlang variables are single assignment, once you’ve bound a value to a variable, you can no longer change that value. In a recursive function variables of the same name are considered fresh in every function iteration, where parameter values are assigned to it. We can see the behaviour of single assignment of variables here:

```
1> A = 3.
3
```

1. But uglier, as we are using a `case` expression instead of pattern matching in the function head.

```

2> A = 2+1.
3
3> A = 3+1.
** exception error: no match of right hand side value 4

```

In shell command 1, we successfully assign an unbound variable. In shell command 2, we pattern match an assigned variable to its value. Pattern matching fails in shell command 3, because the value on the right hand side differs from the current value of A.

Erlang also allows pattern matching over binary data, where we match on a bit level. This is an incredibly powerful and efficient construct for decoding frames and dealing with network protocol stacks. How about decoding an IPv4 packet in a few lines of code?

```

-define(IP_VERSION, 4).
-define(IP_MIN_HDR_LEN, 5).

handle(Dgram) ->
    DgramSize = byte_size(Dgram),
    <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16, ID:16, ...,
      Flgs:3, FragOff:13, TTL:8, Proto:8, HdrChkSum:16, ...,
      SrcIP:32, DestIP:32, Body/binary>> = Dgram,
    if
        (HLen >= 5) and (4*HLen == DgramSize) ->
            OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
            <<Opts:OptsLen/binary, Data/binary>> = Body,
            ...
    end.

```

We first determine the size (number of bytes) of `Dgram`, a variable holding an IPv4 packet as binary data previously received from a network socket. Next, we use pattern matching against `Dgram` to extract its fields; the left hand side of the pattern matching assignment defines an Erlang binary, delimited by `<<` and `>>` and containing a number of fields. (The ellipses within the binary are not legal Erlang code; they indicate fields we've left out to keep the example brief.) The numbers following most of the fields specify the numbers of bits (or bytes for binaries) which each field occupies. For example, `Flgs: 3` defaults to an integer which matches 3 bits, the value of which it binds to the variable `Flgs`. At the point of the pattern match we don't yet know the size of the final field, `Body`, so we specify it as a binary of unknown length in bytes which we bind to the variable `Data`. If the pattern match succeeds, it extracts, in just a single statement, all the named fields from the `Dgram` packet. Finally, we check the value of the extracted `HLen` field in an `if` clause, and if it succeeds, we perform a pattern matching assignment against `Body` to extract `Opts` as a binary of `OptsLen` bytes and `Data` as a binary consisting of all the rest of the data in `Body`. Note how `OptsLen` is calculated dynamically. If you've ever written code using an imperative language such as Java or C to extract fields from a network packet, you can see how much easier pattern matching makes the task.

Functional Influence

Erlang was heavily influenced by other functional programming languages. One functional principle is to treat functions as first class citizens; they can be assigned to variables, be part of complex data structures, be passed as function arguments, or be returned as the results of function calls. We refer to the functional data type as anonymous functions, or *fun* for short. Erlang also provides constructs that allow you to define lists by “generate and test”, using the analogue of comprehensions in set theory. Let’s first start with anonymous functions: functions that are not named and not defined in an Erlang module.

Fun with Anonymous Functions

Functions that take funs as arguments are called *higher order functions*. An example of such a function is `filter`, where a predicate is represented by a *fun* that returns `true` or `false`, applied to the elements of a list. `filter` returns a list made up of those elements that have the required property, namely those for which the *fun* returns `true`. We often use the term *predicate* to refer to a *fun* which, based on certain conditions defined in the function, returns the atoms `true` or `false`. Here is an example of how `filter/2` could be implemented.

```
-module(ex3).
-export([filter/2, is_even/1]).  
  
filter(P,[]) -> [];
filter(P,[X|Xs]) ->
    case P(X) of
        true ->
            [X| filter(P,Xs)];
        false ->
            filter(P,Xs)
    end.  
  
is_even(X) ->
    X rem 2 == 0.
```

To use `filter`, you need to pass it a function and a list. One way to pass the function is to use a *fun* expression, which is a way of defining an anonymous function. In shell command 2 below, you can see an example of an anonymous function that tests for its argument being even:

```
2> ex3:filter(fun (X) -> X rem 2 == 0 end, [1,2,3,4]).  
[2,4]  
3> ex3:filter(fun ex3:is_even/1,[1,2,4]).  
[2,4]
```

A *fun* does not have to be anonymous, and could instead refer to a local or global function definition. In shell command 3 above, the function is described by *fun*

`ex3:is_even/1`, i.e. by giving its module, name and arity. Anonymous functions can also be spawned as the body of a process, and passed in messages between processes; we look at processes in general after the next topic.

List Comprehensions: generate and test

Many of the examples we have looked at so far deal with the manipulation of lists. We've used recursive functions on them, as well as higher order functions. Another approach is to use *list comprehensions*, expressions that generate elements and apply tests (or filters) to them. The format is like this:

```
[Expression || Generators, Tests, Generators, Tests, ... ]
```

where each Generator has the format

```
X <- [2,3,5,7,11]
```

The effect of this is to successively bind the variable `X` to the numbers 2, 3, 5, 7 and 11. In other words, it *generates* the elements from the list: the symbol `<-` is meant to suggest the “element of” symbol for sets, `∈`. In this example, `X` is called a *bound variable*. We've shown only one bound variable here, but a list comprehension can be built from multiple bound variables and generators; we show some examples later in this section.

The *Tests* are Boolean expressions, which are evaluated for each combination of values of the bound variables. If all the *Tests* in a group return `true`, then the *Expression* is generated from the current values of the bound variables. The use of *Tests* in a list comprehension is optional. The list comprehension construct as a whole generates a list of results, one for each combination of values of the bound variables that passes all the tests.

As a first example, we could rewrite the function `filter/2` as a list comprehension:

```
filter(P,Xs) -> [ X || X<-Xs, P(X) ].
```

In this list comprehension, the first `X` is the expression, `X<-Xs` is the generator, and `P(X)` is the test. Each value from the generator is tested with the test, and the expression comprises only those values for which the test returns `true`. Values for which the test returns `false` are simply dropped.

We can use list comprehensions directly in our programs, as in the previous `filter/2` example, or in the Erlang shell:

```
1> [Element || Element <- [1,2,3,4], Element rem 2 == 0].  
[2,4]  
2> [Element || Element <- [1,2,3,4], ex3:is_even(Element)].  
[2,4]  
3> [Element || Element <- lists:seq(1,4), Element rem 2 == 0].  
[2,4]  
4> [io:format("~p~n",[Element]) || Element <- [one, two, three]].  
one
```

```

two
three
[ok,ok,ok]

```

Note how, in shell command 4, we are using list comprehensions to create side effects. The expression still returns a list [ok,ok,ok] containing the return values of executing the `io:format/2` expression on the elements.

The next set of examples show the effect of multiple generators and interleaved generators and tests. In the first, for each value of X, the values bound to Y run through 3, 4 and 5. In the second example, the values of Y depend on the value chosen for X (showing that the expression evaluates X before Y). The remaining two apply tests to both of the bound variables.

```

5> [ {X,Y} || X <- [1,2], Y <- [3,4,5] ].
[[1,3],[1,4],[1,5],[2,3],[2,4],[2,5]]
6> [ {X,Y} || X <- [1,2], Y <- [X+3,X+4,X+5] ].
[[1,4],[1,5],[1,6],[2,5],[2,6],[2,7]]
7> [ {X,Y} || X <- [1,2,3], X rem 2 /= 0, Y <- [X+3,X+4,X+5], (X+Y) rem 2 == 0 ].
[[1,5],[3,7]]
8> [ {X,Y} || X <- [1,2,3], X rem 2 /= 0, Y <- [X+3,X+4,X+5], (X+Y) rem 2 /= 0 ].
[[1,4],[1,6],[3,6],[3,8]]

```

We'll leave you with one of our favorite list comprehensions, which we were contemplating on leaving as an exercise.² Given an 8 by 8 chess board, how many ways can you place N queens on it so that they do not threaten each other? In our example, `queens(N)` returns choices of positions of queens in the bottom N rows of the chessboard, so that each of these is a list of column numbers (in the given rows) where the queens lie. To find out the number of different possible placements, we just count the permutations. Note the `--` list difference operator. It complements `++` which appends lists together. We also use `andalso` instead of `and`, as it short circuits the operation if an expression evaluates to false.

```

-module(queens).
-export([queens/1]).

queens(0) -> [[]];
queens(N) -> [[Val | Columns] || Columns <- queens(N-1), Val <- [1,2,3,4,5,6,7,8] -- Columns,
                                         safe(_Val, [], _N) -> true;
                                         safe(Val, [Column|Columns], N) -> (Val /= Column + N) andalso
                                         (Val /= Column - N) andalso
                                         safe(Val, Columns, N)
                                         ]
safe(_, _, 0) -> true;
safe(_, _, _) -> false.

```

2. You should thank us for this example. One of the authors, when still a student, spent two sleepless nights trying to figure this one out after Joe Armstrong told him it was possible to solve it with four lines of code.

Processes and Message Passing

Concurrency is at the heart of the Erlang programming model. Processes are light-weight, meaning that creating them takes negligible time and memory overhead. Processes do not share memory, and instead communicate with each through message passing. Messages are copied from the stack of the sending process to the heap of the receiving one. As processes execute concurrently in separate memory spaces, these memory spaces can be garbage collected separately, giving Erlang programs very predictable soft real time properties, even under sustained heavy loads. Millions of processes can run concurrently within the same VM, each handling a stand-alone task. Processes fail when exceptions occur, but because there is no shared memory, failure can often be isolated as the process was working on a stand-alone task. This allows other processes working on unrelated or unaffected tasks to continue executing and the program as a whole to recover on its own.

So, how does it all work? Process are created via the `spawn(Mod, Func, Args)` *built-in function (BIF)*. The result of a spawn call is a process identifier, normally referred to as a *pid*. Pids are used for sending messages, and can themselves be part of the message, allowing other processes to communicate back. As we see in [Figure 2-1](#), the process starts executing in the function `Func`, defined in the module `Mod` with arguments passed to the `Args` list.

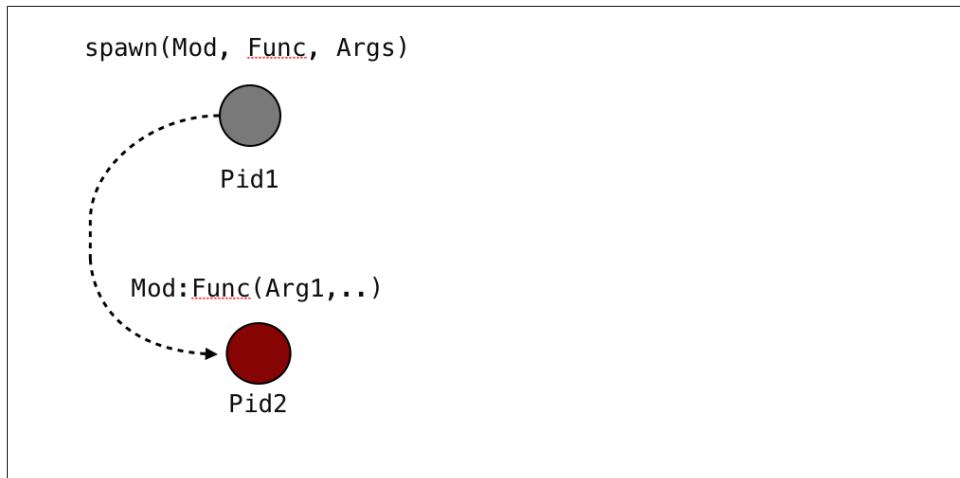


Figure 2-1. Spawning a Process

The following example of an ‘echo’ process shows these basics. The first action of the `go/0` function is to spawn a process executing `loop/0`, after which it communicates with that process by sending and receiving messages. The `loop/0` function receives messages, and depending on their format, either replies to them (and loops) or terminates. To get

this looping behaviour, the function is tail recursive, ensuring it executes in constant memory space.

We know the Pid of the process executing `loop/0` from the spawn, but when we send it a message, how can it communicate back to us? We'll have to send it our pid, which we find using the `self()` BIF.

```
-module(echo).
-export([go/0, loop/0]).

go() ->
    Pid = spawn(echo, loop, []),
    Pid ! {self(), hello},
    receive
        {Pid, Msg} ->
            io:format("~w~n",[Msg])
    end,
    Pid ! stop.

loop() ->
    receive
        {From, Msg} ->
            From ! {self(), Msg},
            loop();
        stop ->
            true
    end.
```

In this echo example, the `go/0` function first spawns a new process executing the `echo:loop/0` function, storing the resulting pid in the variable `Pid`. It then sends to the `Pid` process a message containing the pid of the sender, retrieved using `self()` BIF, along with the atom `hello`. After that, `go/0` waits to receive a message in the form of a pair whose first element matches the pid of the loop process; when such a message arrives, `go/0` prints out the second element of the message, exits the `receive` expression, and finishes by sending the message `stop` to `Pid`.

The `echo:loop/0` function first waits for a message. If it receives a pair containing a pid `From` and a message, it sends a message containing its own pid along with the received `Msg` back to the `From` process and then calls itself recursively. If it instead receives the atom `stop`, `loop/0` returns `true`. When `loop/0` stops, the process that `go/0` originally spawned to run `loop/0` terminates as well, as there is no more code to execute.

Note how, when we run the above program, the `go/0` call returns `stop`. Every function returns a value, that of the last expression it evaluated. In the previous example, the message is `Pid ! go`, which returns the message we just sent to `Pid`.

```
1> c(echo).
{ok,echo}
2> echo:go().
```

```
hello  
stop
```



Bound variables in patterns

Pattern matching is different in Erlang from other languages with pattern matching because variables occurring in patterns can be *bound*. In the `go` function in the `echo` example, the variable `Pid` is already bound to the pid of the process just spawned, so the `receive` expression will accept only those messages in which the first component is that particular pid; in the scenario here, it will be a message from that pid, in fact.

If a message is received with a different first component, then the pattern match in the `receive` will not be successful, and the `receive` will block until a message is received from process `Pid`.

Erlang message passing is *asynchronous*: the expression that sends a message to a process returns immediately and always appears to be successful, even when the receiving process doesn't exist. If the process exists, the messages are placed in the *mailbox* of the receiving process in the same order in which they are received. They are processed using the `receive` expression, which pattern matches on the messages in sequential order. Message reception is selective, meaning that messages are not processed in the order in which they arrive, only the order in which they are matched. Each `receive` clause selects the message it wants to read from the mailbox using pattern matching.

Suppose that the mailbox for the loop process has received the messages `foo`, `stop` and `{Pid, hello}` in that order. The `receive` expression will try to match the first message (here `foo`) against each of the patterns in turn; this fails, leaving the message in the mailbox. We try to do the same with the second message, `stop`; this doesn't match the first pattern, but does match the second, with the result that the process terminates, as there is no more code to execute.

These semantics mean that *we can process messages in whatever order we choose*, irrespective of when they arrive. Code like this:

```
receive  
    message1 -> ...  
end  
receive  
    message2 -> ...  
end
```

will process the atoms `message1` and then `message2`. Without this feature, we'd have to anticipate all the different orders in which messages can arrive, and handle each of those, greatly increasing the complexity of our programs. With selective `receive`, all we do is leave them in the mailbox for later retrieval.

Fail Safe!

In “[Recursion and Pattern Matching](#)” on page 19 we saw the factorial example, and how when it is applied to a negative number the function raises an exception. This also happens when factorial is applied to something that isn’t a number, in this case the atom `zero`:

```
1> ex1:factorial(zero).
** exception error: bad argument in an arithmetic expression
in function ex1:factorial/1
```

The alternative to this would be to program *defensively*, and explicitly identify the case of negative numbers, as well as arguments of any other type, by means of a *catch-all* clause.

```
factorial(0) ->
  1;
factorial(N) when N > 0, is_integer(N) ->
  N * factorial(N-1);
factorial(_) ->
  {error, bad_argument}.
```

The effect of this is would be to require every caller of the function to deal not only with proper results (like `120 = factorial(5)`) but also improper ones of the format `{error, bad_argument}`. If we do this, clients of any function need to understand its failure modes and provide ways of dealing with them, mixing correct computation and error-handling code. A factorial should be called only with non-negative integers. How do you handle errors or corrupt data when you do not know what these errors are or how the data got corrupted?

The Erlang design philosophy says “let it fail!” so that a function, process, or other running entity deals only with the correct case and leaves it to other parts of the system (specifically designed to do this) to deal with failure. One way of doing this is to use the mechanism for exception handling given by the `try - catch` construct. Using the definition:

```
factorial(0) ->
  1;
factorial(N) when N > 0, is_integer(N) ->
  N * factorial(N-1).
```

we can see the construct in action:

```
2> ex1:factorial(zero).
** exception error: no function clause matching ex1:factorial(zero)
3> try ex1:factorial(zero) catch Type:Error -> {Type, Error} end.
{error,function_clause}
4> try ex1:factorial(-2) catch Type:Error -> {Type, Error} end.
{error,function_clause}
5> try ex1:factorial(-2) catch error:Error2 -> {error, Error2} end.
```

```

{error,function_clause}
6> try ex1:factorial(-2) catch error:Error3 -> {error, Error3};
6>                               exit:Reason  -> {exit, Reason} end.
{error,function_clause}

```

The `try - catch` construct gives the user the opportunity to match on the different kinds of exceptions in the clauses, handling them individually. In this example, we match on an `error` exception caused by a pattern match failure. There are also `exit` and `throw` exceptions, the first being the result of a process calling the `exit` BIF. The latter is the result of a user-generated exception using the `throw` expression.

Links and Monitors for Supervision

A typical Erlang system has lots of (possibly dependent) processes running at the same time. How do these dependencies work with the “let it fail” philosophy? Suppose process A interacts with processes B and C ([Figure 2-2](#)); these processes are dependent on A, so if A fails, they can no longer function properly. A’s failure need to be detected, after which B and C need to be terminated before restarting them all. In this section we describe the mechanisms that support this process, namely *process linking*, exit signals, and *monitoring*. These simple constructs allow us to build libraries with complex supervision strategies, allowing us to manage processes that may be subjected to failure at any time.

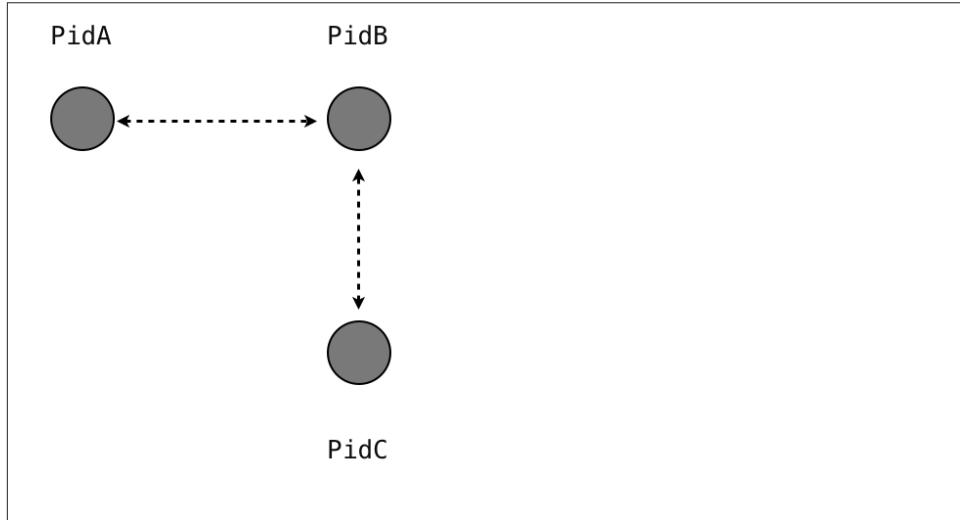


Figure 2-2. Dependent Processes

Links

Calling `link(Pid)` in a process A creates a *bidirectional* link between processes A and Pid. Calling `spawn_link/3` has the same effect as calling `spawn/3` followed by `link/1`, except that it is executed atomically, eliminating the race condition where a process terminates between the spawn and the link. A link from the calling process to Pid is removed by calling `unlink(Pid)`.

The key insight here is that the mechanism needs to be orthogonal to Erlang message passing, but effectuated with it. If two Erlang processes are linked, when either of them terminates, an *exit signal* is sent to the other, which will then itself terminate. The terminated process will in turn send the exit signal to all the processes in its linked set, propagating it through the system. This can be seen in [Figure 2-3](#). The power of the mechanism comes from the ways that this default behaviour can be modified, giving the designer fine control of the termination of the processes within a system. We will now look at this in more detail.

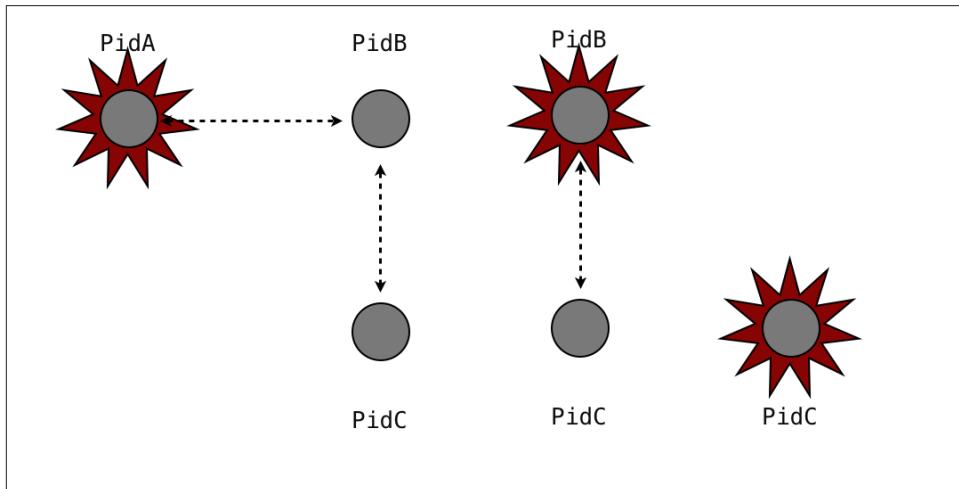


Figure 2-3. Exit signals propagating among linked processes

One pattern for using links is as follows: a server that controls access to resources links to a client while that client has access to a particular resource. If the client terminates, the server will be informed so it can reallocate the resource (or just terminate). If, on the other hand, the client hands back the resource, the server may unlink from the client.

Remember, though, that links are bidirectional, so if the server dies for some reason while client and server are linked, this will by default kill the client too, which we may not want to happen. If that's the case, use a monitor instead of a link, as discussed shortly.

Exit signals can be *trapped* by calling the `process_flag(trap_exit, true)` function. This converts exit signals into messages of the form `{'EXIT', Pid, Reason}`, where `Pid` is the process identifier of the process that has died and `Reason` is the reason it has terminated. These messages are stored in the recipient's mailbox and processed in the same way as all other messages. When a process is trapping exits, the exit signal is *not* propagated to any of the processes in its link set.

Why does a process exit? This can happen for two reasons. If a process has no more code to execute, it terminates *normally*. The `Reason` propagated will be the atom `normal`. *Abnormal termination* is initiated in case of a run-time error, receiving an exit signal when not trapping exits, or by calling the `exit` BIFs. Called with a single argument, `exit(Reason)` will terminate the calling process with reason `Reason`, which will be propagated in the exit signal to any other processes to which the exiting one is linked. When the `exit` BIF is called with two arguments, `exit(Pid, Reason)`, it sends an exit signal with reason `Reason` to the process `Pid`. This will have the same effect as if the calling process had terminated with reason `Reason`.

As we said at the start of this section, users can control the way in which termination is propagated through a system. The options are summarized in [Table 2-1](#).

Table 2-1. Propagation semantics

Reason	Trapping exits (<code>trap_exit = true</code>)	Not trapping exits (<code>trap_exit = false</code>)
normal	Receives <code>{'EXIT', Pid, Normal}</code>	Nothing happens
kill	Terminates with reason <code>killed</code>	Terminates with reason <code>killed</code>
Other	Receives <code>{'EXIT', Pid, Other}</code>	Terminates with reason <code>Other</code>

As the second column of the table shows, a process that is trapping exits will receive an '`EXIT`' message when a linked process terminates, whether the termination is normal or abnormal. The `kill` reason allows one process to force another to exit along with it. This means that there's a mechanism for killing any process, even those that trap exits; note that its reason for termination is `killed` and not `kill`, ensuring that the unconditional termination does not itself propagate.

Monitors

Monitors provide an alternative, unidirectional mechanism for processes to observe the termination of other processes. Monitors differ from links in the following ways.

- A monitor is set up when process A calls `erlang:monitor(process, B)`, where the atom `process` indicates we're monitoring a process and `B` is specified by a `Pid` or registered name. This causes A to monitor B.

- Monitors have an identity given by an Erlang *reference*, which is a unique value returned by the call to `erlang:monitor/2`. Multiple monitors of B by A can be set up, each identified by a different reference.
- A monitor is *unidirectional* rather than bidirectional: if process A monitors process B, this does not mean that B monitors A.
- When a monitored process terminates, a message of the form `{'DOWN', Reference, process, Pid, Reason}` is sent to the monitoring process. This contains not only the Pid and Reason for the termination, but also the Reference of the monitor, and the atom `process` which tells us we were monitoring a process.
- A monitor is removed by the call `erlang:demonitor(Reference)`. Passing a second argument to the function in the format `erlang:demonitor(Reference, [flush])` ensures that any `{'DOWN', Reference, process, Pid, Reason}` messages from the Reference will be flushed from the mailbox of the monitoring process.
- Attempting to monitor a nonexistent process results in a `{'DOWN', Reference, process, Pid, Reason}` message with reason `noproc`; this contrasts with an attempt to link to a nonexistent process, which terminates the linking process.
- If a monitored process terminates, processes who are monitoring it and not trapping exits will not terminate.



References in Erlang are used to guarantee the identity of messages, monitors and other data types or requests. A reference can be generated indirectly by setting up a monitor, but also directly by calling the BIF `make_ref/0`. References are, for all intents and purposes, unique across a multi-node Erlang system. References can be compared for equality and used within patterns, so that it's possible to ensure that a message comes from a particular process, or is a reply to a particular message within a communication protocol.

Taking `monitor/2` and `exit/2` for a trial run, we get the following self-explanatory results.

```

1> Pid = spawn(echo, loop, []).
<0.34.0>
2> erlang:monitor(process, Pid).
#Ref<0.0.0.34>
3> exit(Pid, kill).
true
4> flush().
Shell got {'DOWN',#Ref<0.0.0.34>,process,<0.34.0>,killed}
ok

```

Records

Erlang tuples provide a way of grouping related items, and unlike lists they provide convenient access to elements at arbitrary indexes via the `element/2` BIF. In practice, though, they are most useful and manageable for groups of no more than about five or six items. Tuples larger than that can cause maintenance headaches by forcing you to keep track throughout your code of what field is in which position in the tuple, and using plain numbers to address fields through `element/2` is error-prone. Pattern matching large tuples can be tedious due to having to ensure the correct number and placement of variables within the tuple. Worst of all, though, is that if you have to add or remove a field from a tuple, you have to find all the places your code uses it and make sure to change each occurrence to the correct new size.

Records address the shortcomings of tuples by providing a way to access fields of a tuple-like collection by name. Here's an example of a record used with the Erlang/OTP `inet` module, which provides access to TCP/IP information:

```
-record(hostent,
{
    h_name          %% offical name of host
    h_aliases = []  %% alias list
    h_addrtype     %% host address type
    h_length        %% length of address
    h_addr_list = [] %% list of addresses from name server
}).
```

The `-record` directive is used to define a record, with the record name specified as the directive's first argument. The second argument, which resembles a tuple of atoms, defines the fields of the record. Fields can have specific default values, as shown here for the `h_aliases` and `h_addr_list` fields, both of which have the empty list as their defaults. Fields without specified defaults have the atom `undefined` as their default values.

Record can be used in assignments, pattern matching, and as function arguments similarly to tuples. But unlike tuples, record fields are accessed by name, and any fields not pertinent to a particular part of the code can be left out. For example, the `type/1` function in this module requires access only to the `h_addrtype` field of a `hostent` record:

```
-module(addr).
-export([type/1]).

-include_lib("kernel/include/inet.hrl").

type(Addr) ->
    {ok, HostEnt} = inet:gethostbyaddr(Addr),
    HostEnt#hostent.h_addrtype.
```

First, note that to be able to use a record, we must have access to its definition. The `-include_lib(...)` directive here includes the `inet.hrl` file from the `kernel` applica-

tion, where the hostent record is defined. In the final line of this example, we access the HostEnt variable as a hostent record by supplying the record name after the # symbol. After the record name, we access the required record field by name, h_addr type. This reads the value stored in that field and returns it as the return value of the type/1 function:

```
Erlang R16B03-1 (erts-5.10.4) [source] [64-bit] [smp:8:8] [async-threads:10] [kernel-poll:false]

Eshell V5.10.4 (abort with ^G)
1> addr:type("127.0.0.1").
inet
2> addr:type("::1").
inet6
```

Another way to implement the type function would be to pattern match the h_addr type field against the return value of the `inet:gethostbyaddr/1` function:

```
type(Addr) ->
{ok, #hostent{h_addrtype=AddrType}} = inet:gethostbyaddr(Addr),
AddrType.
```

Here, the AddrType variable within the pattern match captures the value of the h_addr type field as part of the match. This form of pattern matching is quite common with records, and is especially useful within function heads to extract fields of interest into local variables. As you can see, this approach is also cleaner than the field access syntax used in the previous example.

To create a record instance, you set the fields as required:

```
hostent(Host, inet) ->
#hostent{h_name=Host, h_addrtype=inet, h_length=4,
h_addr_list=inet:getaddrs(Host, inet)}.
```

In this example, the `hostent/2` function returns a hostent record instance with specific fields set. Any fields not explicitly set in the code retain their default values specified in the record definition.

Records are just syntactic sugar; under the covers, they are implemented as tuples. We can see this by calling the `inet:gethostbyname/1` function in the Erlang shell:

```
Erlang R16B03-1 (erts-5.10.4) [source] [64-bit] [smp:8:8] [async-threads:10] [kernel-poll:false]

Eshell V5.10.4 (abort with ^G)
1> inet:gethostbyname("oreilly.com").
{ok,{hostent,"oreilly.com",[],inet,4,
[{208,201,239,101},{208,201,239,100}]}}
2> rr(inet).
[connect_opts,hostent,listen_opts,...,udp_opts]
3> inet:gethostbyname("oreilly.com").
{ok,#hostent{h_name = "oreilly.com",h_aliases = [],
h_addrtype = inet,h_length = 4,
h_addr_list = [{208,201,239,101},{208,201,239,100}]}}
```

In shell command 1, we call `gethostbyname/1` to retrieve address information for the host `oreilly.com`. The second element of the result tuple is a `hostent` record, but the shell displays as a tuple, where the first element is the record name and the rest of the elements are the fields of the record in declaration order. Note that the names of the record fields are not part of the actual record instance. To have the record instance be displayed as a record instead of a tuple, we need to inform the shell of the record definition. We do that in shell command 2 using the `rr` shell command, which reads record definitions from its argument and returns a list of the definitions read (we abbreviated the returned list in this example by replacing most of it with an ellipsis). The argument passed to the `rr` command can either be a module name, the name of a source or include file, or a wildcarded name as specified for the `filelib:wildcard/1,2` functions. In shell command 3, we again fetch address information for `oreilly.com`, but this time the shell prints the returned `hostent` value in record format, with field names included.



Correct Record Versions

You need to be extremely careful in dealing with all versions of records once you've changed their definition. You might forget to compile a module using the record (or compile it with the wrong version), load the wrong specification in the shell or send it to a process running code which has not been upgraded. Doing so will in the best case throw an exception when trying to access or manipulate a field which does not exist and in the worse case assign or return the value associated of a different field.

Maps

A `map` is a key-value collection type that resembles dictionary and hash types found in other programming languages. The `map` type differs from records in several ways: it is a built-in type, the number of its fields or key-value pairs is not fixed at compile time, and its keys can be any Erlang term rather than just an atom. While some have touted maps as a replacement for records, in practice they each fulfill different needs and both are useful. Records are fast, so use them when you have a fixed number of fields, whilst maps should be used when you have a need to add fields at runtime.

Creating and manipulating a map is straightforward, as shown below:

```
Erlang/OTP 17 [erts-6.0] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [kernel-poll:f  
Eshell V6.0  (abort with ^G)  
1> EmptyMap = #{}.  
#{}  
2> erlang:map_size(EmptyMap).  
0  
3> RelDates = #{ "R15B03-1" => {2012, 11, 28}, "R16B03" => {2013, 12, 11} }.  
#{ "R15B03-1" => {2012,11,28}, "R16B03" => {2013,12,11} }
```

```

4> RelDates2 = RelDates#{ "17.0" => {2014, 4, 2}}.
#{"17.0" => {2014,4,2},
 "R15B03-1" => {2012,11,28},
 "R16B03" => {2013,12,11}}
5> RelDates3 = RelDates2#"17.0" := {2014, 4, 9}.
#{"17.0" => {2014,4,9},
 "R15B03-1" => {2012,11,28},
 "R16B03" => {2013,12,11}}
6> #{ "R15B03-1" := Date } = RelDates3.
#{"17.0" => {2014,4,2},
 "R15B03-1" => {2012,11,28},
 "R16B03" => {2013,12,11}}
7> Date.
{2012,11,28}

```

In shell command 1, we bind the empty map #{} to variable to `EmptyMap`, and then check its size in shell command 2 using the `erlang:map_size/1` function. As expected, its size is 0 since it contains no key-value pairs. In shell command 3, we create a map with multiple entries, where each key is the name of an Erlang/OTP release paired with a value denoting its release date, using the `=>` map association operator. Shell command 4 takes the existing `RelDates` map and adds a new key-value pair to create a new map, `RelDates2`. Unfortunately, the date we set in shell command 4 is off by one week and we need to change it; shell command 5 shows how we use the `:=` map set-value operator to update the release date. Unlike the `=>` operator, the `:=` operator ensures that the key being updated already exists in the map, thereby preventing errors where the developer misspells the key and accidentally creates a new key-value pair instead of updating an existing key. Finally, shell command 6 shows how using a map in a pattern match allows us to capture the release date associated with the key "R15B03-1" into the variable `Date`, the value of which is accessed in shell command 7. Note that using the `:=` set-value operator is required for map pattern matching.

Macros

Erlang has a macro facility, implemented by the Erlang preprocessor (epp), which is invoked prior to compilation of source code into BEAM code. Macros can be constants, as in

```

#define(ANSWER,42).
#define(DOUBLE,2*).

```

or take parameters, as in

```

#define(TWICE(F,X),F(F(X))).

```

As you can see from the definition of `DOUBLE`, it is conventional (but only conventional) to capitalize names. The definition can be any legal sequence of Erlang tokens; it doesn't have to be a meaningful expression in its own right.

Macros are invoked by preceding them with a ? character, as in

```
test() -> ?TWICE(?DOUBLE,?ANSWER).
```

It is possible to see the effect of macro definitions by compiling with the '`P`' flag in the shell:

```
c(<filename>,['P']),
```

which creates a *filename.P* file in which the previous definition of `test/0` becomes:

```
test() -> 2 * (2 * 42).
```

It is also possible for a macro call to record the text of its parameters. For example, if we define:

```
-define(Assign(Var,Exp), Var=Exp, io:format("~s = ~s -> ~p~n",[??Var,??Exp,Var]) ).
```

then `?Assign(Var,Exp)` has the effect of performing the assignment `Var = Exp`, but also, as a side-effect, prints out a diagnostic message. For example:

```
test_assign() -> ?Assign(X, lists:sum([1,2,3])).
```

behaves like this:

```
1> macros:test_assign().
X = lists : sum ( [ 1 , 2 , 3 ] ) -> 6
ok
```

Using flags, you can define conditional macros, such as:

```
-ifdef(debug).
-define(Assign(Var,Exp), Var=Exp, io:format(~s = ~s -> ~p~n,[??Var,??Exp,Var]) ).
-else.
-define(Assign(Var,Exp), Var=Exp).
-endif.
```

Now, if you use the compiler flags `{d,debug}` to set the debug flag, `?Assign(Var,Exp)` will perform the assignment and print out the diagnostic code. Conversely, leaving the debug flag unset by default or clearing it through `{u,debug}` will cause the program to do the assignment without executing the `io` expression.

Upgrading Modules

One of the advantages of dynamic typing is the ability to upgrade your code during runtime, the need to take down the system. One second, you are running a buggy version of a module, but you can load a fix without terminating the process and it starts running the fixed version, retaining its state and variables ([Figure 2-4](#)). This works not only for bugs, but also for upgrades and new features. This is a crucial property for a system that needs to guarantee “five nines availability”—that is 99.999% uptime including upgrades and maintenance.

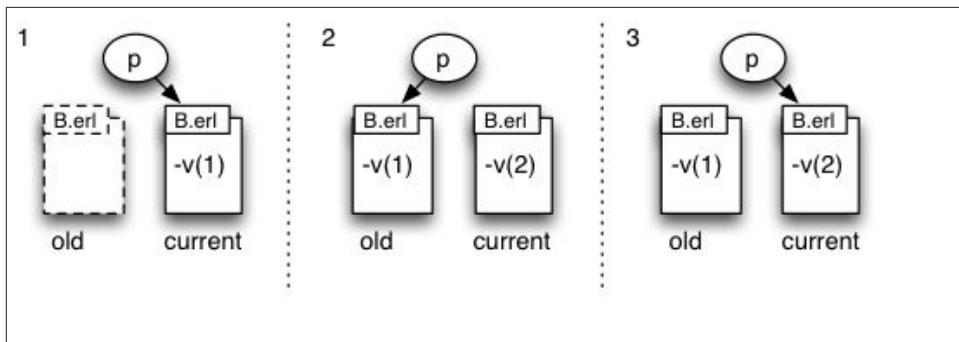


Figure 2-4. A software upgrade

At any one time, two versions of a module may exist in the virtual machine, the *old* and the *current* versions. Frame 1 in Figure 2-4 shows process p executing in the current version of module B. In Frame 2, new code for the module B is loaded, either by compiling the module in the shell or by explicitly loading it. After you load the module, p is still linked to the same version of B, which has now become the old version. But the next time p makes a fully-qualified call to a function in module B, a check will be made to ensure that p is running the latest version of the code. (If you recall from earlier on in this chapter, a fully-qualified call is one where the module name is prefixed to the function name.) If the process is not running the latest version, the pointer to the code will be switched to the new *current* version, as shown in Frame 3. This applies to *all* functions in B, not just the function whose call triggered the switch. While this is the essence of a software upgrade, let's go through the fine print to make sure you understand all the details.

- Suppose that the code for the loop of a running process is itself upgraded. The effect depends on the form of the function call. If the function call is *fully-qualified*, i.e. of the form `B:loop()`, the next call will use the upgraded code; otherwise (when the call is simply `loop()`), the process will continue to run the old code.
- The system holds only two versions of the code, so suppose that process p is executing *v(1)* of module B, and *v(3)* is loaded: the version *v(1)* will be *purged*, and any process (such as p) looping in that version of the module will be unconditionally terminated.
- New code can be loaded in a number of ways. Compiling the module will cause code to be reloaded: this can be initiated in the shell by `c(Module)` or by calling the Erlang function `compile:file(Module)`. Code can also be loaded explicitly in the shell by `l(Module)` or by a call `code:load_file(Module)`. In general, code is loaded by calling a function in a module that is not already loaded: this causes the compiled code, a *.beam* file, to be loaded; this requires the code to have been compiled already,

perhaps using the `erlc` command-line tool. Note that recompiling a module with `erlc` does not cause it to be reloaded.

- While *old* code is purged when a new version is loaded, it is possible to call `code:purge(Module)` explicitly to purge an old version (without loading a new version). This has the effect of terminating all processes running the old code before removing the code. The call returns `true` if any processes were indeed terminated, and `false` if none were. Calling `code:soft_purge(Module)` will remove the code only if no processes were running it: the result is `true` in that case and `false` otherwise.

ETS: Erlang Term Storage

While lists are an important data type, they need to be linearly traversed and, as a result, will not scale. If you need a key-value store where the lookup time is constant, or the ability to traverse your keys in lexicographical order, Erlang Term Storage (ETS) tables come in handy. An ETS table is a collection of Erlang tuples, keyed on a particular position in the tuple.

ETS tables come in four different kinds:

Set

Each key-value tuple can occur only once.

Bag

Each key-value tuple combination can only occur once, but a key can appear multiple times.

Duplicate bag

Tuples can be duplicated.

Ordered set

These have the same restriction as sets, but the tuples can be visited in order by key.

Access time to a particular element is in constant time, except for ordered sets, where access time is proportional to the logarithm of the size of the table ($O(\log n)$ time).

Depending on the options passed in at table creation (`ets:new`), tables have one of the following traits:

`public`

Accessible to all processes.

`private`

Accessible to the owning process only.

`protected`

All processes can read the table, but only the owner can write to it.

Tables can also have their key position specified at creation time ({keypos,N}). This is mainly useful when storing records, as it allows the developer to specify a particular field of the record as the key. The default key position is 1.

Normally, programs access tables through the table ID returned by the call to new, but tables can also be *named* when created, which makes them accessible by that name.

A table is linked to the process that creates it, and is deleted when that process terminates. ETS tables are in-memory only, but long-lived tables are provided by DETS tables, which are stored on disk (hence the “D”).

Elementary table operations are shown in the following interaction.

```
1> TabId=ets:new(tab,[named_table]).  
tab  
2> ets:insert(tab,{haskell, lazy}).  
true  
3> ets:lookup(tab,haskell).  
[{haskell,lazy}]  
4>ets:insert(tab,{haskell, ghci}).  
true  
5> ets:lookup(tab,haskell).  
[{haskell,ghci}]  
6> ets:lookup(tab,racket).  
[]
```

As can be seen, the default ETS table is a set, so that the insertion at line 4 overwrites the insertion at line 2, and the table is keyed at the first position. Note also that looking up a key returns a list of all the tuples matching the key.

Tables can be traversed, as seen here:

```
7> ets:insert(tab,{racket,strict}).  
true  
8> ets:insert(tab,{ocaml,strict}).  
true  
9> ets:first(tab).  
racket  
10> ets:next(tab,racket).  
haskell
```

Since tab is a set ETS, the elements are not ordered by key; instead, their ordering is determined by a hash value internal to the table implementation. In the example here, the first key is racket and the next is haskell. However, using first and next on an ordered set will give traversal in order by key. It is also possible to extract bulk information using the match function:

```
11> ets:match(tab,['$1','$0']).  
[[strict,ocaml],[ghci,haskell],[strict,racket]]  
12> ets:match(tab,['$1','_']).  
[[ocaml],[haskell],[racket]]
```

```
13> ets:match(tab,['$1',strict]).  
[[ocaml],[racket]]
```

The second argument, which is a *symbolic tuple*, is matched against the tuples in the ETS table. The result is a list of lists: each list giving the values matched to the named variables '\$0' etc, in ascending order; these variables match any value in the tuple. The wildcard value '_' also matches any value, but its argument is not reported in the result.

Let's implement code that uses an ETS table to associate phone numbers — or more accurately, mobile subscriber integrated services digital network (MSISDN) numbers — to Pids in a module called `hlrl`. We create the association when phones attach themselves to the network and delete it when they detach. We then allow users to look up the Pid associated with a particular phone number as well as the number associated with a Pid. Read through this code, as we will be using it as part of a larger example in later chapters.

```
-module(hlrl).  
-export([new/0, attach/1, detach/0, lookup_id/1, lookup_ms/1]).  
  
new() ->  
    ets:new(msisdn2pid, [public, named_table]),  
    ets:new(pid2msisdn, [public, named_table]),  
    ok.  
  
attach(Ms) ->  
    ets:insert(msisdn2pid, {Ms, self()}),  
    ets:insert(pid2msisdn, {self(), Ms}).  
  
detach() ->  
    case ets:lookup(pid2msisdn, self()) of  
        [{Pid, Ms}] ->  
            ets:delete(pid2msisdn, Pid),  
            ets:delete(msisdn2pid, Ms);  
        [] ->  
            ok  
    end.  
  
lookup_id(Ms) ->  
    case ets:lookup(msisdn2pid, Ms) of  
        [] -> {error, invalid};  
        [{Ms, Pid}] -> {ok, Pid}  
    end.  
  
lookup_ms(Pid) ->  
    case ets:lookup(pid2msisdn, Pid) of  
        [] -> {error, invalid};  
        [{Pid, Ms}] -> {ok, Ms}  
    end.
```

In our test run of the module, the shell process attaches itself to the network using the number 12345. We look up the mobile handset using both the number and the pid, after

which we detach. When reading the code, note that we are using a named public table, meaning any process can read and write to it as long as they know the table name.

```
2> hlr:new().
ok
3> hlr:attach(12345).
true
4> hlr:lookup_ms(self()).
{ok,12345}
5> hlr:lookup_id(12345).
{ok,<0.32.0>}
6> hlr:detach().
true
7> hlr:lookup_id(12345).
{error,invalid}
```

Distributed Erlang

All of the examples we have looked at so far execute on a single virtual machine, also referred to as a *node*. Erlang has built-in semantics allowing programs to run across multiple nodes: processes can transparently spawn processes on other nodes and communicate with them using message passing. Distributed nodes can reside either on the same physical or virtual host or on different ones.

This programming model is designed to support scaling and fault tolerance on systems running behind firewalls over trusted networks. Out of the box, Erlang distribution is *not* designed to support systems operating across potentially hostile environments such as the Internet or shared cloud instances. Because different systems have different requirements on security, no one size fits all. Varying security requirements can easily (or not so easily) be addressed if you provide your own security layers and authentication mechanisms, or by modifying Erlang's networking and security libraries.

Naming and Communication

In order for an Erlang node to be part of a distributed Erlang system, it needs to be given a name. A *short name* is given by starting Erlang with `erl -sname node`, identifying the node on a local network using the host name. On the other hand, starting a node with the `-name` flag means that it will be given a *long name* and identified by the fully qualified domain name or IP address. In a particular distributed system, all nodes must be of the same kind, i.e., all short or all long.

Processes on distributed nodes are identified in precisely the same way as a local node, using their Pids. This allows constructs such as `Pid!Msg` to send messages to a process running on any node in the cluster. On the other hand, registering a process with an alias is local to each host, so `{bar,'foo@myhost'}!Msg` is used to send the message `Msg` to the process named `bar` on the node '`foo@myhost`'. Note the form of this node

identifier: it is a combination of `foo` (the name of the node) and `myhost` (the short or local network name of the host on which the node `foo` is running).

You can spawn and link to processes on any node in the system, not just locally, using `link(Pid)`, `spawn(Node, Mod, Fun, Args)` and `spawn_link`. If the call is successful, `link` will return the atom `true`, while `spawn` returns the `Pid` of the process on the remote host.



Code is not automatically deployed remotely for you! If you spawn a process remotely, it is your responsibility to ensure that the compiled code for the spawned process is already available on the remote host, and that it is placed in the search path for the node on that host.

Node Connections and Visibility

In order to communicate, Erlang nodes must share a *secret cookie*. By default, each node has a randomly generated cookie, unless there is already a value stored in the `.erlang.cookie` file in your home directory. If this file does not exist, it is created the first time you start a distributed Erlang node, and populated with a random sequence of characters. This behaviour can be overridden by starting the node with the `-setcookie` `Cookie` flag, where `Cookie` is the cookie value. Cookie values can also be changed within a program by calling `erlang:set_cookie(Node, Cookie)`.

In an Erlang distributed system, by default, all nodes know about and can interact with all others so long as they share a cookie. However, starting a node with the `-hidden` flag leaves it unconnected to anything initially, and any connections that it needs to make have to be set up by hand. The `net_kernel` module allows fine-grained control of this and other aspects of interconnections. Hidden nodes can have a variety of uses, including operations and maintenance, as well as serving as bridges between different node clusters.

Messages between two processes on different nodes are guaranteed to be delivered in order: the difference in a distributed system is that it is possible for a remote node to go down. A general mechanism for dealing with this is to *monitor* whether or not the remote node is alive. This is different from monitoring a local process, described in “[Monitors](#)” on page 32.

```
monitor_node(Node, true),
{serve, Node} ! {self(), Msg},
receive
    {ok, Resp} ->
        monitor_node(Node, false),
        ... process Resp ...;
    {nodedown, Node} ->
```

```
... handle the lack of response ...
end.
```

In this fragment, a message—such as a remote procedure call—is sent to the serve process on Node. Before sending the request, the Node is monitored, so that if the node goes down, a {nodedown, Node} message will be received, and the lack of response can be handled. Once a response Resp is successfully received, the code switches off monitoring before processing the response. You can also use the monitor BIF to get notifications of the health of remote nodes.

To test distributed communications, start two distributed Erlang nodes using different names, but the same cookie:

```
erl -sname foo -setcookie abc
erl -sname bar -setcookie abc
```

In the following sequence, shell command 1 pings the remote node, creating a connection. Shell command 2 looks up all of the connected nodes using the nodes() BIF, binding the remote node to the variable Node. Shell command 4 spawns a process on the remote node, which sends the shell process on our local node its pid. We receive that pid in command 5 and inspect its node of origin in command 6 using the node/1 BIF. Shell command 7 spawns a process on a remote node, sending the node identifier back to the local node. Note how node names are atoms, and thus are defined within single quotes.

```
$erl -sname bar -setcookie abc
Erlang R16B03 (erts-5.10.4) [source] [64-bit] [smp:8:8] [async-threads:10] [hipe] [kernel-poll:false]

Eshell V5.10.4 (abort with ^G)
(bar@macbook-pro-2)1> net_adm:ping('foo@macbook-pro-2').
pong
(bar@macbook-pro-2)2> [Node] = nodes().
['foo@macbook-pro-2']
(bar@macbook-pro-2)3> Shell = self().
<0.38.0>
(bar@macbook-pro-2)4> spawn(Node, fun() -> Shell ! self() end).
<5985.46.0>
(bar@macbook-pro-2)5> receive Pid -> Pid end.
<5985.46.0>
(bar@macbook-pro-2)6> node(Pid).
'foo@macbook-pro-2'
(bar@macbook-pro-2)7> spawn(Node, fun() -> Shell ! node() end).
<5985.47.0>
(bar@macbook-pro-2)8> flush().
Shell got 'foo@macbook-pro-2'
ok
```

Summing Up

In this chapter, we've given an overview of the basics of Erlang we believe are important for understanding the examples in the remainder of the book. The concurrency model, error handling semantics, and distributed processing not only make Erlang a powerful tool, but are the foundation of OTP's design. Before you progress, be warned that the better you understand the internals of Erlang, the more you are going to get out of the OTP design principles and this book. We will provide many more examples written in pure Erlang, which for some might be enough to understand the OTP rationale. Try moving ahead, but if you find yourself struggling, we suggest reading *Erlang Programming*, published by O'Reilly and co-authored by one of the authors of this book. The current book can be seen as a continuation of *Erlang Programming*, expanding many of the original examples from that book. Other great titles that will also do the trick include *Introducing Erlang*, also published by O'Reilly, *Learn You Some Erlang For Great Good* from No Starch Press (and also available online free of charge), and *Programming Erlang* written by Erlang co-inventor Joe Armstrong and published by the Pragmatic Programmers.

What's Next?

In the upcoming chapters, we introduce OTP behaviours. We start by providing an Erlang example of a client-server application, breaking it up into generic and specific parts. The generic parts are those that can be reused from one client-server application to another and are packaged in library modules. The specific parts are those that are project specific and have to be implemented for the individual client server applications. In Chapter 4, we migrate the code to an OTP-based generic server behaviour, introducing the first building block of Erlang based systems. As more behaviours are introduced in the subsequent chapters, it will become clear how Erlang systems are architected and glued together.

CHAPTER 3

Behaviours

As a prelude to learning how to structure our process supervision trees and architect our concurrency models, let's spend time understanding the underlying principles behind behaviours. Instead of diving straight into the world of interface functions and callbacks, we will explain what goes on behind the scenes, ensuring you use OTP behaviours efficiently and understand their benefits and advantages. So, what are they?

Erlang processes that solve radically different tasks follow similar design patterns. The most commonly used patterns have been abstracted and implemented in a set of generic library modules called the OTP behaviours. When reading about behaviours, you should see them as a formalisation of process design patterns.

Although the strict concept of design patterns used in OO programming hasn't been applied to Erlang, OTP provides a powerful, reusable solution for concurrent processes that hides and abstracts away all of the tricky aspects and borderline conditions. It ensures that projects do not have to reinvent the wheel, while maximizing reusability and maintainability through a solid, well-tested generic and reusable code base. These behaviours are, in "design pattern speak", implementation libraries of the concurrency models.

Process Skeletons

If you try to picture an Erlang process managing a key-value store and a process responsible for managing the window of a complex GUI system, they might at first glance appear very different in functionality and have very little in common. That is often not the case, though, as both processes will share a common life cycle. Both will:

- Be started.
- Repeatedly receive messages, handle them, and send replies.

- Be terminated (normally or abnormally).

Processes, irrespective of their purpose, have to be spawned. Once spawned, they will initialize their state. The state will be specific to what that particular process does, so in the case of a window manager, it might draw the window and display its contents. In the case of a key-value store, it might create the empty table and fill it with data stored in backup files or populate it using other tables spread across a distributed cluster of nodes.

Once the process has been initialized, it is ready to receive events. These events could, in the case of the window manager, be keystrokes in the window entry boxes, button clicks or menu item selections. They could also be dragging and dropping of widgets, effectively moving the window or objects within it. Events would be programmed as Erlang messages. Upon receiving a particular message, the process would handle the request accordingly, evaluating the content and updating its internal state. Keystrokes would be displayed, clicking buttons or choosing menu items would result in window updates, while dragging and dropping would result in the objects being moved across the screen. A similar analogy could be given for the key-value store. Asynchronous messages could be sent to insert and delete elements in the tables, and synchronous messages—messages that wait for a reply from the receiver—could be used to look up elements and return the value to the client.

Finally, processes will terminate. A user might have picked the *Close* menu entry in the menus or clicked on the *Destroy* button. If that happens, resources allocated to that window have to be released and the window hidden or shut down. Once the clean-up procedure is completed, there will be no more code for the process' to execute, so it should terminate *normally*. In the case of the key-value store, a `stop` message might have been sent to the process, resulting in the table being backed up on another node or saved on a persistent medium.

Abnormal termination of the process might also occur as a result of a trapped exception or an exit signal from one of the processes in the link-set. Where possible, if caught through a `trap_exit` flag or a `try-catch` expression, the exception would prompt the process to call the same set of commands that would have been called as a result of a normal termination. We say “where possible”, as the power cord of the computers might have been pulled out, the hard drive might have failed, the administrator might have tripped on the network cable, or the process might have been terminated unconditionally through an exit signal with the reason `kill`. [Figure 3-1](#) shows a typical process flow diagram outlining the life cycle of a process.

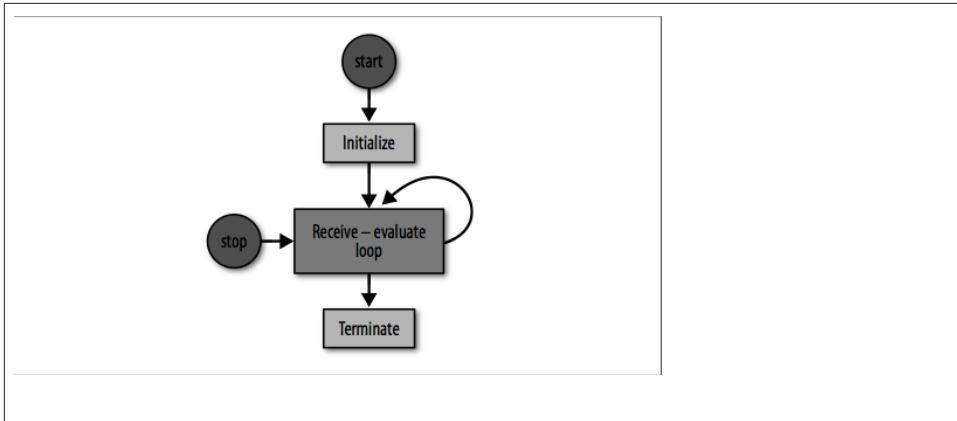


Figure 3-1. The Process Skeleton

As we've described, even if processes perform different tasks, they will perform these tasks in a similar way, following particular patterns. As a result of following these patterns, processes share a similar code base. A typical Erlang process loop, which has to be started, must handle events, and is finally terminated, might look like this:

```

start(Args) -> % Start the server.
    spawn(server, init, [Args]).

init(Args) -> % Initialize the internal process state
    State = initialize_state(Args),
    loop(State).

loop(State) -> % Receive and handle messages.
    receive
        {handle, Msg} ->
            NewState = handle(Msg, State),
            loop(NewState);
        stop -> terminate(State) % Stop the process.
    end.

terminate(State) -> % Cleanup prior to termination.
    clean_up(State).

```

This pattern is typical of a client/server behaviour. The server is started, it receives requests in the `handle/2` function, where necessary sends replies, changes the state, and loops ready to handle the next incoming message. Upon receiving a `stop` message, the process terminates after having cleaned up its resources.

Although we say that this is typical Erlang client/server behaviour, it is in fact the pattern behind all patterns. It is so common that even code written without the OTP behaviour libraries tends to use the same function names. This allows anyone reading the code to

know that the process state is initialized in `init/1`, that in `loop/1`, messages are received and individually handled in the `handle/2` call, and finally, that any cleaning up of resources is managed in the `terminate/1` function. Someone trying to maintain the code later would understand the basic behaviour without needing any knowledge of the communication protocol, underlying architecture, or process structure.

Design Patterns

Let's start drilling down into a more detailed example, focusing on client/server architectures implemented in Erlang. Client and servers are represented as Erlang processes, with their requests and replies sent as messages. Have a look at [Figure 3-2](#) and think of examples of client/server architectures that you have worked with or read about; preferably architectures with few similarities among them (as in our example of a key-value store and a window manager). Focusing on Erlang constructs and patterns in the code of these applications, try to list the similarities and differences between the implementations. Ask yourself which parts of the code are generic and which parts are specific. What code is unique to that particular solution, and what code could be reused in other client/server applications?

Let's give you a hint in the right direction: sending a client request to a server will be generic. It can be done in a uniform manner across any client/server architecture, irrespective of what the server does. What will be specific, however, are the contents of that message.

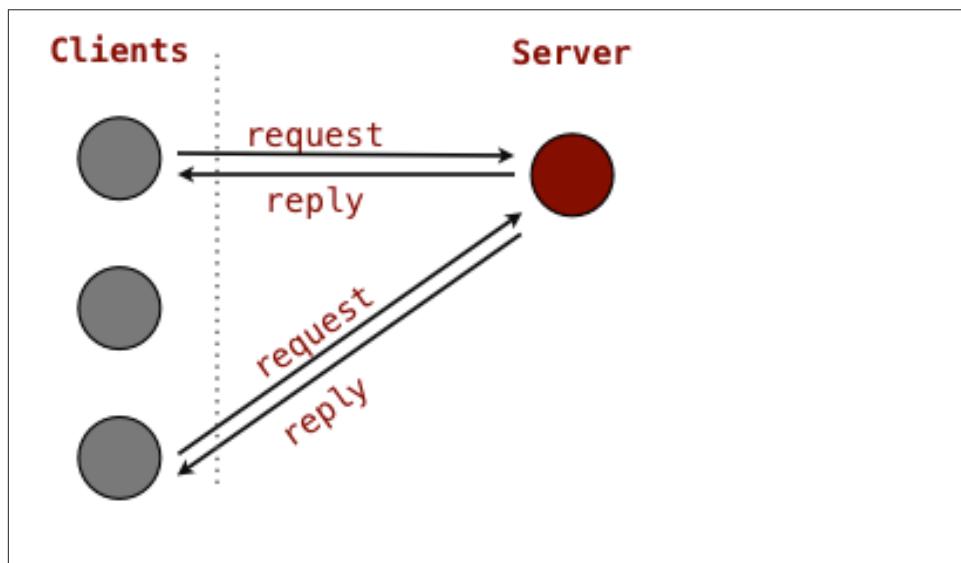


Figure 3-2. The Client/Server Process Architecture

We start off by spawning a server. Creating a process that calls the `init/1` function is generic. What is specific are the arguments passed to the call and the expressions in the function that initialize the process state returning the loop data. The loop data plays the role of a variable that stores process data between calls.

Storing the loop data in between calls will be the same from one process to another, but the contents of the loop data itself will be specific. It changes not only according to the particular task the process might execute, but for each particular instance of the task.

Sending a message to the client will be generic, as is the client/server protocol used to manage replies. What is specific are the requests sent to the server, how they are handled, and the responses sent back to the client. While the response is specific, sending it back to the client process is handled generically.

It should be possible to stop servers. While sending a `stop` message or handling an exception or 'EXIT' signal is generic, the functions called to clean up the state prior to termination will be specific.

Table 3-1. Client/Server Generic and Specific Code

Generic	Specific
<ul style="list-style-type: none">• Spawning the server• Storing the loop data• Sending requests to the server• Sending replies to the client• Receiving server replies• Stopping the server	<ul style="list-style-type: none">• Initialising the server state• The loop data• The client requests• Handling client requests• Contents of server reply• Cleaning up

Callback Modules

The idea behind OTP behaviours is to split up the code into two modules, one for the generic pattern, referred to as the *behaviour module*, and one for specifics, referred to as the *callback module*. The generic behaviour module can be seen as the driver. While it doesn't know anything about what the callback module does, it is aware of a set of exported callback functions it has to invoke and the format of their return values. The callback module isn't aware of what the generic module does either, it only complies with the format of the data it has to return when its callback functions are invoked.

Another way of explaining this is as a contract between the behaviour and callback modules. They have to agree on a set of names types for the functions in the callback API and respect the return values.

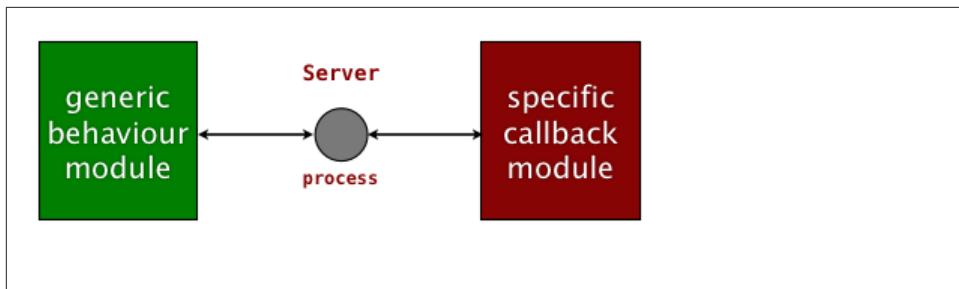


Figure 3-3. The callback Module

The behaviour module contains all of the generic functionality reused from one implementation to another. Behaviours are provided by OTP as library modules. The callback module is implemented by the application developer. It contains all of the specific code for the implementation of that particular process. OTP provides five behaviours that cover the majority of all cases. They are:

- **Generic Server** - Used to model client/server behaviours
- **Generic Finite State Machine** - Used for finite-state machine programming
- **Generic Event Handler/Manager** - Used for writing event handlers
- **Supervisor** - Used for fault-tolerant supervision trees
- **Application** - Used to encapsulate resources and functionality

Generic servers are the most commonly used behaviour. They are used to model processes using the client/server architecture, including the examples of the key-value store and the window manager we've already discussed.

Generic finite state machine behaviours provide all of the generic constructs needed when working with FSMs. The most common use of FSMs is in the implementation of automated control systems, protocol stacks, communication protocols and decision making systems. The code for the FSMs can be implemented manually or generated by another program.

Generic event handlers and managers are used for event-driven programming, where events are received as messages and one or more actions (called handlers) are applied to them. Typical examples of handler functionality include logging, metrics gathering, and alarming.

You can view handlers as a publish-subscribe communication layer, where publishers are processes sending events of a specific type and subscribers are consumers who do something with the events.

A *supervisor* is a behaviour whose only task is to monitor its children, which can be workers as well as other supervisors. Allowing supervisors to monitor other supervisors

results in a process structures we call *supervision trees*. We will be covering supervision trees in the upcoming chapters. Supervisors restart children based on configuration parameters defined in the callback functions. Supervision trees are packaged in a behaviour we call an application. The application starts the top-level supervisor, encapsulating processes that depend on each other into the main building blocks of an Erlang node.

Generic servers, finite state machine, and event handlers are examples of workers: processes who perform the bulk of the computations. They are held together by supervisors and application behaviours. If you need other behaviours not included as part of the standard library, you can implement them following a set of specific rules and directives explained in [Chapter 10](#). We call them *special processes*.

Now you might be wondering: what is the point of adding a layer of complexity to our software? The reasons are many. Using behaviours, we are reducing the code base while creating a standardized programming style needed when developing software in the large. By encapsulating all of the generic design patterns in library modules, we reuse code while reducing the development effort. The behaviour libraries we use consist of a solid, well-tested base that has been used in production systems since the mid-nineties. They cover all of the tricky aspects of concurrency, hiding them from the programmer. As a result, the final system will have fewer bugs¹ while being built on a fault tolerant base. The behaviours will have built-in functionality such as logs, tracing, and statistics, and be extensible in a generic way across all processes using that behaviour.

Another important advantage of using behaviours is that they promote a common programming style. Anyone reading the code in a callback module will immediately know that the process state is initialized in the `init` function, and that `terminate` contains the clean-up code executed whenever the process is stopped. They will know how the communication protocol will work, how processes are restarted in case of failure, and how supervision trees are packaged. Especially when programming in the large, this approach allows anyone reading the code to focus on the project specifics while using their existing knowledge of the generics. This common programming style also brings a component-based terminology to the table, giving potentially distributed teams a way to package their deliverables and use a standard vocabulary to communicate among each other. At the end of the day, much more time is spent reading and maintaining code than writing it. Making code easy to understand is imperative when dealing with complex systems that never fail.

So, with lots of advantages, what are the disadvantages? Learning to use behaviours properly and proficiently can be difficult. It takes time to learn how to properly create

1. Bug-free systems exist only in the dreams of the bureaucrats. When using Erlang/OTP, equal focus should be placed on correctness and error recovery, as the bugs will manifest themselves in production systems whether you like it or not.

systems using OTP design principles, but as documentation has improved, training courses and books have become available and tools have been written, this has become less of an issue. Just the fact that you are reading a book dedicated solely to OTP says it all.

Behaviours add a few layers to the call chain and slightly more data will be sent with every message and reply. While this might affect performance and memory usage, in most cases, it will be negligible, especially considering the improvement in quality and free functionality. What is the point of writing code which is fast but buggy? The small increase in memory usage and reduction in performance is a small price to pay for reliability and fault tolerance. The rule of thumb is to always start with behaviours, and optimize when bottlenecks occur. You will find that optimizations as a result of inefficient behaviour code are rarely if ever needed.

Extracting Generic Behaviours

Having introduced behaviours, let's look at a familiar client/server example written in pure Erlang without using behaviours. We will use the frequency server featured in the *Erlang Programming* book and implemented in the `frequency` module. No worries if you have not read the book and are not familiar with it, we will explain what it does as we go along. The server is a frequency allocator for cell phones. When a phone connects a call, it needs to be allocated a frequency to use as a communication channel for that conversation. The client holds this frequency until the call is terminated, after which the frequency is deallocated, allowing other subscribers to reuse it.

As this is the first major Erlang example in the book, we will step through it in more detail than usual. In the subsequent chapters, we speed up the pace, so if your Erlang is a bit rusty, take this opportunity to get up to speed. We will take the code from the frequency server example, find the generic parts embedded in the module and extract them into a library module. The outcome will be two modules, one containing generic reusable code and the second, specific code with the frequency server's business logic.

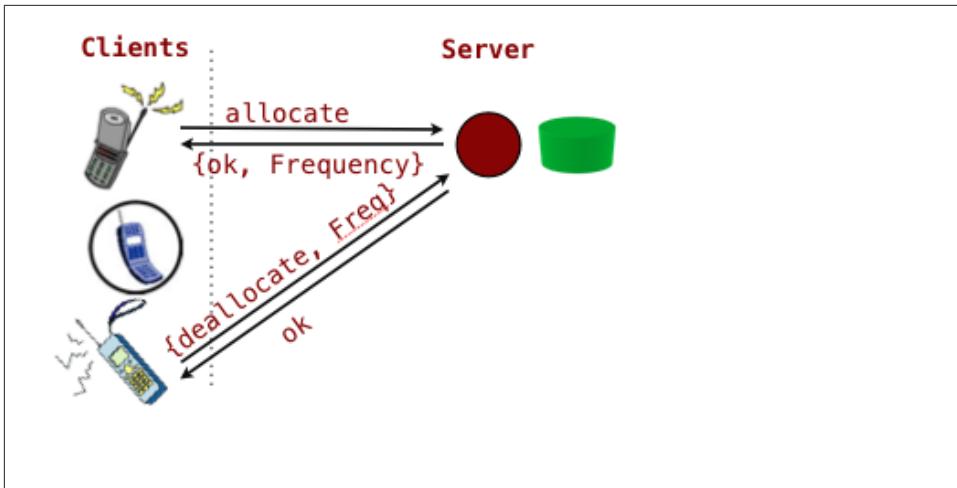


Figure 3-4. The Frequency Server

The clients and server will be represented as Erlang processes, and the exchange of information between them will be done through message passing hidden behind a functional interface. The functional interface used by the clients contains the functions `allocate/0` and `deallocate/1`:

```
allocate() -> {ok, Frequency} | {error, no_frequency}
deallocate(Frequency) -> ok
```

The `allocate/0` function returns the result `{ok, Frequency}` if there is at least one available frequency. But if all frequencies are in use, the tuple `{error, no_frequency}` is returned instead. When the client is done with a phone call, the frequency being used can be released by making a function call to `deallocate/1`, passing the Frequency used as an argument.

```
start()-> true
stop() -> ok
```

We start the server with the `start/0` call, terminating it with `stop/0`. The server is registered statically with the alias `frequency`, so no Pids need to be saved and used for message passing.

A trial run of the frequency module from the shell might look like this. We start the server, allocate all six available frequencies, and fail to allocate a seventh one. Only by deallocating frequency 11 are we then able to allocate a new one. We terminate the trial run by stopping the server:

```
1> frequency:start().
true
```

```

2> frequency:allocate(), frequency:allocate(), frequency:allocate(),
   frequency:allocate(),frequency:allocate(), frequency:allocate().
{ok,15}
3> frequency:allocate().
{error,no_frequency}
4> frequency:deallocate(11).
ok
5> frequency:allocate().
{ok,11}
6> frequency:stop().
ok

```

If you need a deeper understanding of the code, feel free to download the module from the book site and run the example. Next we'll go through the code in detail, explain what it does, and separate out the generic and the specific parts.

Starting The Server

Let's begin with the functions used to create and initialize the server. The `start/0` function spawns a process that calls the `frequency:init/0` function, registering it with the `frequency` alias. The `init` function initializes the process state with a tuple containing the list of available frequencies, conveniently hard coded in the `get_frequencies/0` function, and the list of allocated frequencies, represented by the empty list. We bind the frequency tuple, referred to in the rest of the example as the *process state* or *loop data*, to the `Frequencies` variable. The process state variable changes with every iteration of the loop when available frequencies are moved between list of allocated and available ones.

Note how we exported the `init/0` function, because it is passed as an argument to the spawn BIF, and how we registered the server process with the same name as the module. The latter, while not mandatory, is considered a good Erlang programming practice as it facilitates debugging and troubleshooting live systems.

```

-module(frequency).
-export([start/0, stop/0, allocate/0, deallocate/1]).
-export([init/0]).

start() ->
    register(frequency, spawn(frequency, init, [])).

init() ->
    Frequencies = {get_frequencies(), []},
    loop(Frequencies).

get_frequencies() -> [10,11,12,13,14,15].

```

Have a look at the above code and try to spot the generic expressions. Which expressions will not change from one client server implementation to another?

Starting with the export directives, you have to always start and stop servers, irrespective of what they do. So we consider these functions to be generic. What is also generic is the spawning, registering, and calling of an initialization function containing the expressions used to initialize the process state. The process state will be bound to a variable and passed to the process loop. Note, however, that while the functions and BIFs might be considered generic, expressions in the functions and arguments passed to them aren't. They will differ between different client/server implementations. We've highlighted all the parts we consider generic in the following code example.

```
-module(frequency).
-export([start/0, stop/0, allocate/0, deallocate/1]).
-export([init/0]).

start() ->
    register(frequency, spawn(frequency, init, [])).

init() ->
    Frequencies = {get_frequencies(), []},
    loop(Frequencies).

get_frequencies() -> [10,11,12,13,14,15].
```

From the generic, let's move on to the specific, which is the unhighlighted code in the earlier example. The first server-specific detail that stands out in the example is the module name `frequency`. Module names obviously differ from one server implementation to another. The client functions `allocate/0` and `deallocate/1` are also specific to this particular client/server application, as you will probably not find them in a windows manager or a key-value store (and if they did happen to share the same name, the functions would be doing something completely different). Although starting the server, spawning the server process and registering it are generic, the registered name and module containing the `init` function are considered specific.

The arguments passed to the `init` function are also specific. In our example, we are not passing any arguments (hence, the arity 0), but that could change in other client/server implementations. The expressions in the `init/0` function are used to initialize the process state. Initialising the state changes from one implementation to another. Various applications might initialize window settings and display the window, create an empty key-value store and upload a persistent backup, or in this example, generate a tuple containing the list of available frequencies.

When the process state has been initialized, it is bound to a variable. Storing the process state is considered generic, but the contents of the state itself are specific. In the code example that follows, we have highlighted the `Frequency` variable as specific. This means that the content of the variable is specific, whereas the mechanism of passing it to the process loop is generic. Finally, the `get_frequencies/0` call used in `init/0` is also specific. In a real world implementation, we would probably read the frequencies from a

configuration file, a persistent database, or through a query to the base stations. For the sake of this example, we've been lazy and hardcoded them in the module.

Let's highlight the specific code:

```
-module(frequency).
-export([start/0, stop/0, allocate/0, deallocate/1]).
-export([init/0]).  
  
start() ->
    register(frequency, spawn(frequency, init, [])).  
  
init() ->
    Frequencies = {get_frequencies(), []},
    loop(Frequencies).  
  
get_frequencies() -> [10,11,12,13,14,15].
```

Are you seeing the pattern and line of thought we are trying to get you into? Let's continue doing the same with the rest of the module, starting with the client functions.

The Client Functions

We refer to the functions called by client processes to control and access the services of a server process as the *client API*. It is always good practice, for readability and maintainability, to hide message passing and protocol in a functional interface. The client functions in the running example do exactly this. In fact, we've taken it a step further here, encapsulating the sending of requests and receiving of replies in the `call/1` and `reply/2` functions. They contain code that otherwise would have to be cloned for every message sent and received.

```
stop()      -> call(stop).
allocate()   -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).  
  
call(Message) ->
    frequency ! {request, self(), Message},
    receive
        {reply, Reply} -> Reply
    end.  
  
reply(Pid, Reply) ->
    Pid ! {reply, Reply}.
```

The `stop/0` function sends the atom `stop` to the server. The server, upon receiving it in its receive/evaluate loop, interprets it and takes appropriate action. For readability and maintainability reasons, it is good practice to use keywords that describe what we are trying to do, but for all it matters, we could have used the atom `foobar`, as it is not the name of the atom but the meaning we give it in our program that is important. In our

case, `stop` ensures a normal termination of the process. We will see how it is handled later on in the example.

The client functions `allocate/0` and `deallocate/1` are called and executed in the scope of the client process. The client sends a message to the server by executing one of the client functions in the `frequency` module. The message is passed as an argument to the `call/1` function and bound to the `Message` variable. The `Message` is in turn inserted in a tuple of the form `{request, Pid, Message}`, where the `Pid` is the client process identifier, retrieved by calling the `self()` BIF, and used by the server as the destination for a response in the format `{reply, Reply}`. We refer to this extra padding as the “protocol” between the client and the server.

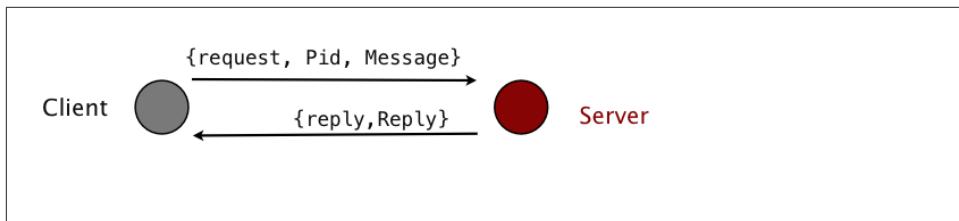


Figure 3-5. The Message Protocol

The server receives the request, handles it, and sends a reply using the `reply/2` call. It passes the `Pid` sent in the client request as the first argument and the message it has to send back as the second. This message is pattern matched in the receive clause of the `call/1` function, returning the contents of the variable `Reply` as a result. This will be the result returned by the client functions. A sequence diagram with the exchange of messages between the cell phones and the frequency server is shown in [Figure 3-6](#).

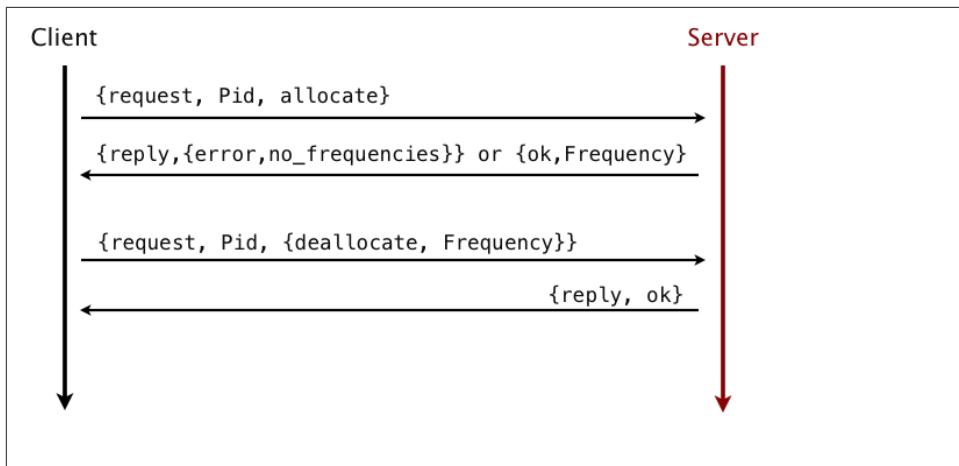


Figure 3-6. The Frequency Server Messages

So, which parts of the code are generic? Which will not change from one client/server implementation to another? First in line is the `stop/0` function, used whenever we want to inform the server that it has to terminate. This code can be reused, as it is universal in what it does. Every time we want to send a message, we use `call/1`. There is a catch, however, as this function is not completely generic. Have a look at the code and try to spot the anomaly.

```

stop()      -> call(stop).
allocate()   -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).

call(Message) ->
  frequency ! {request, self(), Message},
  receive
    {reply, Reply} -> Reply
  end.

reply(Pid, Reply) ->
  Pid ! {reply, Reply}.

```

We are sending a message to a registered process `frequency`. This name will change from one server implementation to the next. However, everything else in the call is generic. The function `reply/2`, called by the server process, is also completely generic. So what remains specific in the client functions are the client functions themselves, their message content to the server, and the name of the server itself:

```

stop()      -> call(stop).
allocate()   -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).

call(Message) ->

```

```

frequency ! {request, self(), Message},
receive
    {reply, Reply} -> Reply
end.

reply(Pid, Reply) ->
    Pid ! {reply, Reply}.

```

By hiding the message protocol in a functional interface and abstracting it, we are able to change it without affecting the code outside of the `frequency` module, client calls included. We will see how this comes in handy later on in the chapter, when we start dealing with some of the common error patterns that occur when working with concurrent programming.

The Server Loop

Server processes will iterate in a receive/evaluate loop. They will wait for client requests, handle them, return a result, and loop again, waiting for the next message to arrive. With every iteration, the process state is updated and side-effects may be generated.

```

loop(Frequencies) ->
    receive
        {request, Pid, allocate} ->
            {NewFrequencies, Reply} = allocate(Frequencies, Pid),
            reply(Pid, Reply),
    end.

```

In our frequency server example, the `loop/1` function receives the `allocate`, `{deallocate, Frequency}`, and `stop` commands. Allocating a frequency is done through the helper function `allocate/2`, which, given the loop data and the Pid of the client, moves a frequency from the list of available ones to the list of allocated ones. Deallocating a frequency invokes the `deallocate/2` call to do the opposite, moving the frequency from the list of allocated frequencies to the list of available ones.

Both calls return the pair of updated frequency lists that make up the process state; this new state is bound to the variable `NewFrequencies` and passed to the tail-recursive `loop/1` call. In both cases, a reply is sent back to the clients. When allocating a frequency, the contents of the variable `Reply` are either `{error, no_frequency}` or `{ok, Frequency}`. When deallocating a frequency, the server sends back the atom `ok`.

When stopping the server, we acknowledge having received the message through the `ok` response and, by the lack of a call to `loop/1`, we make the process terminate normally, as opposed to an abnormal termination that results from a runtime error. In this example, there is nothing to clean up, so we don't do anything other than acknowledge the `stop` message. Had this server handled some resource such as a key-value store, we could have ensured that the data was safely backed up on a persistent medium. Or in the case of a window server, we'd close the window and release any allocated objects associated with it.

With all of this in mind, what functionality do you think is generic?

For starters, looping is generic, as is receiving messages and sending replies. The protocol used to send and receive messages is generic, but the messages and replies themselves aren't. Finally, stopping the server is generic, as is acknowledging the stop message.

```
loop(Frequencies) ->
receive
    {request, Pid, allocate} ->
        {NewFrequencies, Reply} = allocate(Frequencies, Pid),
        reply(Pid, Reply),
        frequency. This name will change from one server implementation to the
        next. However, everything else in the call is generic. The function reply/2,
        called by the server process, is also completely generic.
        NewFrequencies = deallocate(Frequencies, Pid),
end.
```

We have not highlighted the variables `Frequencies` and `NewFrequencies` used to store the process state. Although storing the process state is generic, the state itself is specific. That is, the type of the state and the particular value that this variable has are specific, but not the generic task of storing the variable itself.

With the generic contents out of the way, the specifics include the loop data, the client messages, how we handle the messages, and the responses we send back as a result.

```
loop(Frequencies) ->
receive
    {request, Pid, allocate} ->
        {NewFrequencies, Reply} = allocate(Frequencies, Pid),
        reply(Pid, Reply),
end.
```

Had there been specific code to be executed when stopping the server, it would also have been marked as specific. This code is usually placed in a function called `terminate`, which, given the reason for termination and the loop data, handles all of the cleaning up.

Functions Internal to the Server

The functions that actually perform the work of allocating or deallocating a frequency within the server are not “visible” outside the server module itself, and so we call them *internal* to the server. The `allocate/1` call returns a tuple with the new frequencies and the reply to sent back to the client. If there are no available frequencies, the first function clause will pattern match because the list is empty. The frequencies are not changed, and `{error, no_frequency}` is returned to the client. If there is at least one frequency, the second function clause will match.

The available frequency list is split into a head and a tail, where the head contains the available frequency, and the tail (a possibly empty list) contains the remaining available frequencies. The frequency with the client pid is added to the allocated list, and the response `{ok, Freq}` is sent back to the client.

When deallocating a frequency in the `deallocate/2` function, we delete it from the allocated list and add it to the available one. Have a look at both functions and try to figure out what is generic and what is specific.²

```
allocate({[], Allocated}, _Pid) ->
    {[[], Allocated], {error, no_frequency}};;
allocate({[Freq|Free], Allocated}, Pid) ->
    {[Free, [{Freq, Pid}|Allocated]], {ok, Freq}}.

deallocate({[Free, Allocated], Freq} ->
    NewAllocated=lists:keydelete(Freq, 1, Allocated),
    {[Freq|Free], NewAllocated}.
```

This should have been an easy to answer, as these internal functions are all specific to our frequency server. When did you last allocate and deallocate frequencies when working with a key-value store or a windows manager?

The Generic Server

Now that we've gone through this example and distinguished the generic from the specific code, let's get to the core of this chapter, namely the separation of the code into two separate modules. We will put all of the generic code into the `server` module and all of the specific code into `frequency`.

Despite these changes, we maintain the same functionality and interface. Calling the `frequency` module in our new implementation should be no different from the trial run we did in “[Extracting Generic Behaviours](#)” on page 54.

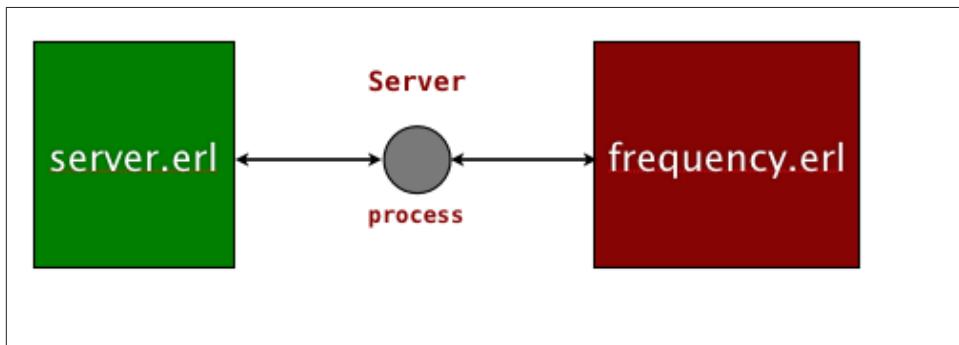


Figure 3-7. Frequency and Server Modules

The `server` module is in control, managing the process activities. Whenever it has to handle specific functionality it does not know how to execute, it hands over to the

2. Warning, this is a trick question.

callback functions in the `frequency` module. Let's start with the generic code in the `server` module which starts and initializes the server.

```
-module(server).
-export([start/2, stop/1, call/2]).
-export([init/2]).

start(Name, Args) ->
    register(Name, spawn(server, init, [Name, Args])).

init(Mod, Args) ->
    State = Mod:init(Args),
    loop(Mod, State).
```

Spawning a process, registering it, and calling the `init` function are all generic, whereas the alias we register the process with, the name of the callback module and the arguments we pass to the `init` function are all specific. We pass this specific information as parameters to the `server:start/2` function, using them where needed. `Name` is used both as an alias to register the frequency process, and as the name of the callback module. `Args` is passed to the `init` function and is used to initialize the process state.

We keep the client functions in the `frequency` module, using it as a wrapper around the `server`. By doing so, we are hiding implementation details, including the very use of the `server` module. Just like in our previous example, we start the `server` using `frequency:start/0`, resulting in a call to `server:start/2`. The newly spawned server, through the `Mod:init/1` call, invokes the callback function in the `frequency` module, initializing the process state by creating the tuple containing the available and allocated frequencies. `Mod` is bound to the callback module `frequency` and `Args` is bound to `[]`. The frequency tuple gets bound to the `State` variable, which alongside `Mod`, is passed as an argument to the loop in the `server` module.

```
-module(frequency).
-export([start/0, stop/0, allocate/0, deallocate/1]).
-export([init/1, terminate/1, handle/2]).

start() -> server:start(frequency, []).

init(_Args) ->
    {get_frequencies(), []}.

get_frequencies() -> [10,11,12,13,14,15].

terminate(_Frequencies) ->
    ok.
```

The `init/1` callback is required to return the initial process state, stored and used in the server receive/evaluate loop. In the `init/1` callback function, note that we are not using the value of the `_Args` parameter. Because `init/1` is a callback function, we have to follow the required protocol and functional interface for that callback API. In the

general case, `init/1` requires an argument because there might be server implementations that need data at startup. This particular example doesn't, so we pass the empty list and ignore it.

Let's jump back to the `server` module. When a client process wants to send a request to the server, it does so by calling `server:call(frequency, Msg)`. The server, when responding, does so using the `reply/2` call. We are, in effect, hiding all of the message passing behind a functional interface.

Another generic function is `server:stop/1`. We distinguish it from `call/2` because we want to fix its meaning and therefore differentiate it from `server:call(frequency, {stop, self()})`, which could be treated by the developer as a specific call. Instead, by calling `stop`, we invoke the `terminate/1` callback function, which is given the process state and will contain all of the specific code executed when shutting down the server. In our case, we have kept the example to a minimum. Note that we could, however, have chosen to terminate all of the client processes that had been allocated a frequency.

```
stop(Name) -> % server.erl
  Name ! {stop, self()},
  receive {reply, Reply} -> Reply end.

call(Name, Msg) ->
  Name ! {request, self(), Msg},
  receive {reply, Reply} -> Reply end.

reply(To, Reply) ->
  To ! {reply, Reply}.
```

To ensure that we maintain the same interface, we export exactly the same functions in our new implementation of the `frequency` module.

```
stop()          -> server:stop(frequency). % frequency.erl
allocate()      -> server:call(frequency, {allocate, self()}).
deallocate(Freq) -> server:call(frequency, {deallocate, Freq}).
```

These functions send requests and stop messages to the server. When the process receives the messages, the relevant callback functions in the `frequency` module are invoked. In the case of the stop message, it is the function `terminate/1`. It takes the process state as an argument and its return value is sent back to the client, becoming the return value of the `stop/1` call.

```
loop(Mod, State) -> % server.erl
  receive {request, From, Msg} -> {NewState, Reply} = Mod:handle(Msg, State),
    end.
```

In the case of a call request, the `handle/2` callback is invoked. The call takes two arguments, the first being the message bound to the variable `Msg` and the second the process state bound to the variable `State`. Pattern matching on the `Msg` picks the function clause

that handles the message. The callback has to return a tuple in the format {NewState, Reply}, where NewState contains the updated frequencies and Reply is the reply sent back to the client. Have a look at the implementation of `allocate/2`. It returns exactly that: a tuple where the first element is the updated process state and the second element either {ok, Frequency} or {error, no_frequency}.

The first clause of the `receive` in `loop/2` takes the return value from `handle/2`, sends back a reply to the client using `reply/2` and loops with the new state, awaiting to receive the next request.

```
handle({allocate, Pid}, Frequencies) -> %frequency.erl
    allocate(Frequencies, Pid);
handle({deallocate, Freq}, Frequencies) ->
    {deallocate(Frequencies, Freq), ok}.

allocate({[], Allocated}, _Pid) ->
    {[[], Allocated], {error, no_frequency}};
allocate({[Freq|Free], Allocated}, Pid) ->
    {[Free, [{Freq, Pid}|Allocated]], {ok, Freq}}.

deallocate({Free, Allocated}, Freq) ->
    NewAllocated=lists:keydelete(Freq, 1, Allocated),
    {[Freq|Free], NewAllocated}.
```

The same applies to the deallocate request. The frequency is deallocated, and the `handle/2` call returns a tuple with the new state, returned by the `deallocate/2` call, and the response is sent back to the client, namely the atom `ok`.

So what we now have is our frequency example split up into a library module we call `server` and a specific, callback module that we call `frequency`. This is all there is to understanding Erlang behaviours. It is all about splitting up the code into generic and specific parts, and packaging the generic parts into reusable libraries to hide as much of the complexity as possible from the developers. We've kept this example simple to show our point, and barely scratched the surface of the corner cases that are handled behind the scenes in the proper behaviour libraries. We will however start covering them in the next section, and introduce them as we talk about the individual behaviour library modules.

Message Passing: Under the hood

Concurrent programming is not easy. You need to deal with race conditions, deadlocks, and critical sections as well as many corner cases. Despite this, you rarely hear Erlang developers complain, let alone discuss these problems. The reason is simple, as most of these issues become non-issues as a result of the OTP framework. In this chapter, we extracted the generic code from a particular client/server system, but in doing so we kept our example as simple as possible. There are many error conditions in a scenario like this that are handled behind the scenes by the behaviour library modules we cover

in the next chapter. Just to emphasize the point, they are handled without the programmer having to be aware of them. Race conditions, especially with multicore architectures, have become more common, but they should be picked up with appropriate modeling and testing tools such as Concuerror, McErlang, PULSE and QuickCheck.

Having said that, let's look at an example of how behaviour libraries help us to hide a lot of the tricky cases an inexperienced developer might not think of when first implementing a concurrent system. We use the `call/2` function from the previous example, expanding it as we go along.

```
call(Name, Message) ->
    Name ! {request, self(), Message},
    receive
        {reply, Reply} -> Reply
    end.

reply(Pid, Reply) ->
    Pid ! {reply, Reply}.
```

We send a message to the server of the format `{request, Pid, Message}` and wait for a response of the format `{reply, Reply}`. When we receive the reply, as can be seen in [Figure 3-8](#), how can we be confident that the reply is actually a reply from the server, and not a message complying to the protocol sent by another process?

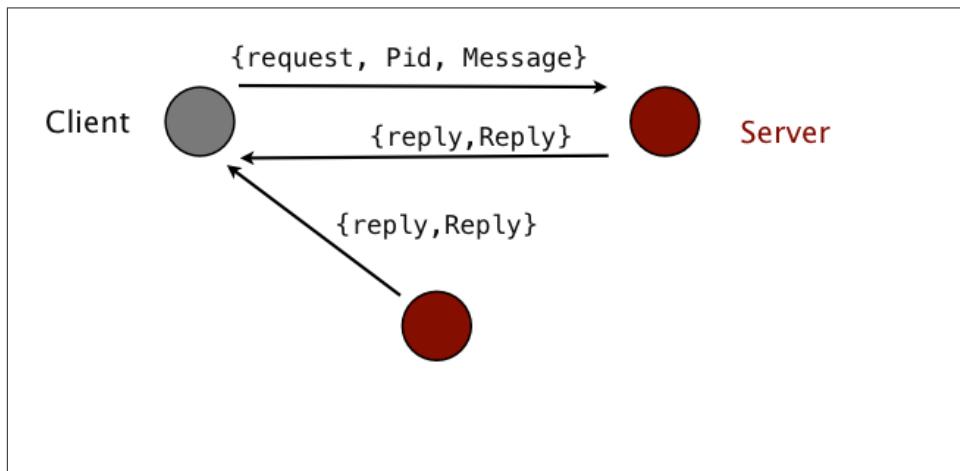


Figure 3-8. Message Race Conditions

We can't. The solution to this problem is to use references. By creating a unique reference with the `make_ref()` BIF, adding it to the message, and including it in the reply, we will be guaranteed that the response is actually the reply to our request, and not just a mes-

sage that happens to comply with our protocol. Adding references, our code would look like this:³

```
call(Name, Msg) ->
    Ref = make_ref(),
    Name ! {request, {Ref, self()}}, Msg,
    receive {reply, Ref, Reply} -> Reply end.

reply({Ref, To}, Reply) ->
    To ! {reply, Ref, Reply}.
```

Note how `Ref` is already bound when entering the receive clause, ensuring replies are the result of the original message. This solves the problem, but is this enough? What happens if the server crashes before we send a request? If `Name` is an alias, we are covered because the client process will terminate when trying to send a message to a nonexistent registered process. But if `Name` is a pid, the message will be lost and the client will hang in the receive clause of the call function. Or similarly, what happens if the server crashes between receiving the message and sending the reply? This could be as a result of our request, or as the result of another client request it might be handling. Having a registered process will not cover this case either, as the process is alive when the message is sent.

The solution is to monitor the server. In doing so, let's use the `monitor` BIF instead of a link, because links are bi-directional and might cause side effects on the server if the child process were to be killed during the request. While the client wants to monitor termination of the server, terminating the client should not affect the server. The `monitor` BIF returns a unique reference, so we can drop the `make_ref()` BIF and use the monitor reference to tag our messages.

```
call(Name, Msg) ->
    Ref = erlang:monitor(process, Name),
    Name ! {request, {Ref, self()}}, Msg,
    receive {reply, Ref, Reply} -> erlang:demonitor(Ref), Reply; {'DOWN', Ref, process, _}
end.
```

Have we covered everything that can go wrong? No, not really. By monitoring the process, we are now exposing ourselves to another race condition. Consider the following sequence of events:

1. The client monitors the server.
 2. The client sends a request to the server.
 3. The server receives the request and handles it.
 4. The server sends back a response to the client.
-
3. Minor changes are also needed to the code in order to get the stop call to work. We will skip them in this example.

5. The server crashes as the result of another request.
6. The client receives a DOWN message as a result of the monitor.
7. The client extracts the server response from its mailbox.
8. The client demonitors the (now defunct) server.

We are stuck with a DOWN message in the client mailbox containing a ref that will never match. Now, what are the chances of that happening? Do you really think someone would think of that particular test case where the server terminates right after it sent the client its reply, but before the client executes the `erlang:demonitor/2` call? While an extreme corner case, we still need to handle the DOWN message as it might cause a memory leak. We do this by passing the `[flush]` option to the second argument in the `demonitor/2` call, ensuring that any DOWN messages belonging to that monitor are not left lingering in the process mailbox.

Are we there yet? No, not really: what if `Name` is not an alias of a registered process? We need a `catch` to trap any exception raised as a result of the client sending a message to a nonexistent registered process. We don't really care about the result of the `catch` (if we did, we would have used `try-catch`), because if the server does not exist, `monitor/1` will send a DOWN message. Our new code now looks like this:

```
call(Name, Msg) ->
    Ref = erlang:monitor(process, Name),
    catch Name ! {request, {Ref, self()}}, Msg},
    receive {reply, Ref, Reply} ->    erlang:demonitor(Ref,[flush]),      Reply; {'DOWN', Ref, pr
```

Unfortunately, though, these changes are still not enough. What happens if process A does a synchronous call to B at the same time as process B calls A? By synchronous call, we mean an Erlang message exchange where the sending process expects a response, and the message and response are each sent as asynchronous messages. A enters the receive clause right after sending its request matching on a unique reference sent with the request, and B does the same. Back to answering our original question, if two processes synchronously call each other using this code, we get a deadlock. While deadlocks are a result of a design flaw, they might happen in live systems and a recovery mechanism (preferably a generic one) needs to be put in place. The easiest way to resolve deadlocks is through a timeout in your receive statement, terminating the process. We will go into more detail on deadlocks and timeouts and show you how OTP solves this problem in the next chapter.

Summing up

In this chapter, we've covered the principles behind concurrency design patterns, introducing the concept of behaviour libraries. We hope that we have made our point about the importance and power of behaviour libraries, as understanding them is fun-

damental to understanding the underlying principles of OTP. Decades of experience in process-oriented programming are reflected in them, removing the burden from the developers, reducing their code base, and ensuring that corner cases are handled in a consistent, efficient, and correct manner. Be honest: how many of the corner cases discussed in this example would you have handled in a first iteration? What about your colleagues? Imagine testing and maintaining a system where everyone has reinvented the wheel with their own representation of these concurrent conditions and corner cases!

If you have the time, pick a simple client/server example you might have written when learning Erlang. It could be a key-value store, a chat server or any other process which receives and handles requests. If you do not have any examples at hand, use the mobile subscriber database example from the ETS and Dets chapter of the *Erlang Programming* book. You can download the code from the authors' github repositories.

Another useful exercise is to extend the call function with an `after` clause, making the process exit with reason `timeout`. Create a new function called

```
call(Name, Message, Timeout)
```

which, given a `Timeout` integer value in milliseconds or the atom `infinity` allows the user to set their own timeouts. Keep the `call/2` call, setting the default to five seconds. If the server does not respond within the given timeout value, make the client process terminate abnormally with the reason `timeout`. Don't forget to clean up before exiting, as the exit signal might be caught in a try-catch expression in the code using the server library.

What's Next?

In the next chapters, we introduce the library modules that together give us the OTP behaviours. We start with the `gen_server` library, and then later use a similar approach to introduce finite state machines, event handlers, supervisors and applications. We have not yet covered deadlocks, timeouts, and error cases that can arise when dealing with distribution or messages that never match. These are all topics we discuss when covering the individual behaviour libraries.

CHAPTER 4

Generic Servers

Having broken up the radio frequency allocator into generic and specific modules and investigated some of the corner cases that can occur when dealing with concurrency, you will have figured out that there is no need to go through this process every time you have to implement a client-server behaviour. In this chapter, we are introducing the `gen_server` OTP behaviour, a library module that contains all of the generic client-server functionality whilst handling a large number of corner cases. Generic servers are the most commonly used behaviour pattern, setting the foundations for other behaviours, all of which can (and at some point were) implemented using this module.

Generic Servers

The `gen_server` module implements the client-server behaviour we extracted in the previous chapter. It is part of the standard library application and available whenever downloading the Erlang/OTP distribution. It contains the generic code that interfaces with the callback module through a set of *callback functions*. The *callback module*, in our example containing the code specific to the frequency server, is implemented by the programmer. The module has to export a series of functions that follow naming and typing conventions, so that their inputs and return values conform to the protocol required by the behaviour.

As seen in [Figure 4-1](#), both the behaviour and callback module are executed within the scope the same server process. In other words, it is a process looping in the generic server module from which it calls the callback functions in the callback module.

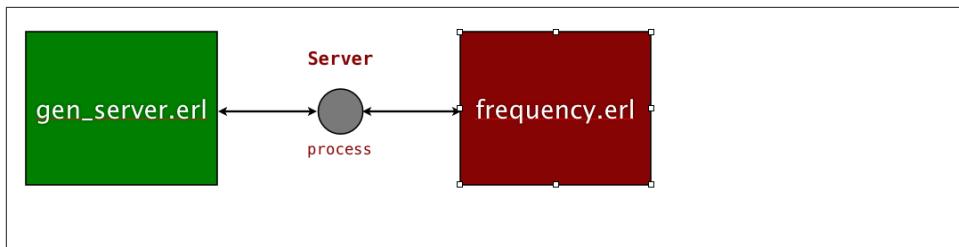


Figure 4-1. The Callback and Behaviour Modules

The `gen_server` library module provides functions to start and stop the server. You will supply callback code to initialise the system, and in case of either normal or abnormal process termination, it is possible to call a function from your callback module to clean up the state prior to termination. In particular, you no longer need to send messages to your process. Generic servers encapsulate all message passing in two functions, one for sending synchronous messages and one for sending asynchronous messages. These handle all of the borderline cases we discussed in the previous chapter, and many others we probably hadn't even realised could be an issue. There is also built-in functionality for software upgrades, where you are able to suspend your process and migrate data from one version of your system to the next. And finally we enable timeouts, both on the client side when sending requests, and on the server side should no messages have been received in a predetermined time interval.

We are now going to cover all of the callback functions required when using generic servers. They include:

- The `gen_server:start_link/4` function, which creates a server process and initialises it by calling the `init/1` callback function.
- The synchronous `gen_server:call/2` function sends a message to a server process and waits for a reply. The message is handled in the server by the `handle_call/3` callback function.
- The asynchronous `gen_server:cast/2` function, which sends a message to a server process and immediately returns. The message is handled in the server by the `handle_cast/2` callback function.
- Termination is handled when any of the server callback functions return a stop message, resulting in the `terminate/2` callback function being called.

We will look at these functions in more detail including all of their arguments, return values and associated callbacks as soon as we've covered the module directives.

Behaviour Directives

If we are implementing an OTP behaviour, we need to include a behaviour directives in our module declarations.

```
-module(frequency).  
-behaviour(gen_server).  
  
-export([start_link/1, init/1, ...]).  
  
start_link(...) -> ...
```

The `behaviour` directive is used by the compiler to issue warnings over callback functions which might not have been exported¹. Not all callbacks are mandatory, so if any are missing, a warning (which you can ignore) will be issued. An even more important use of the behaviour directive is for the poor souls² who have to support, maintain and debug your code long after you've moved on to other exciting and stimulating projects. They will see it and immediately know you have been using the generic server patterns. If they want to see the initialisation of the server, they go to the `init/1` function. If they want to see how the server cleans up after itself, they jump to `terminate/3`, and obviously, everything in between the two. This is a great improvement over a situation in which every company, project, or developer reinvents their own, possibly buggy, client/server implementations. No time is wasted understanding this framework, allowing whoever is reading the code to concentrate on the specifics.

The compiler warnings come as a result of the `-behaviour(gen_server).` directive, as we are omitting the `code_change/3` function, a callback we will cover in [Chapter to Come] when discussing release upgrades. There is a second, optional directive `-vsn(Version)` used to keep track of module versions during code upgrade (and down-grade). We will cover them in more detail in [Chapter to Come].

Starting a Server

With the knowledge of our module directives, let's start a server. Generic servers and other OTP behaviours are started not with the spawn BIFs, but with dedicated functions that do more behind the scenes than just spawn a process.

```
gen_server:start_link({local,Name},Mod,Args,Opts) -> {ok, Pid} |  
                                ignore |  
                                {error, Reason}
```

The `start_link/4` function takes four arguments. The first tells the `gen_server` module to register the process locally with the alias `Name`. `Mod` is the name of the callback module,

1. These declarations will also be used by Dialyzer for checking type discrepancies.
2. At the risk of sounding repetitious, be nice to them, as it might be you someday.

where the server-specific code and the callback functions will be found. `Args` is an Erlang term passed to the callback function that initialises the server state. `Opts` is a list of process and debugging options that we will cover at various points in this chapter. For the time being, let's pass the empty list to it. If a process is already registered with the `Name` alias, `{error, {already_started, Pid}}` is returned. Keep a vigilant eye over which process executes which functions. You note them in [Figure 4-2](#), where the server bound to the process `Pid` is started by the supervisor. The supervisor is denoted by a double ring as it is trapping exits.

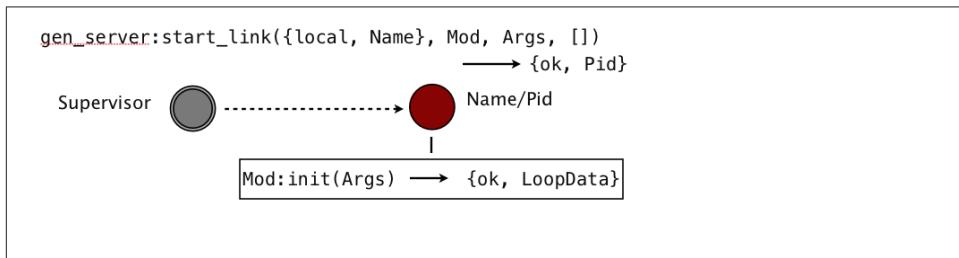


Figure 4-2. Starting A Generic Server

When the `gen_server` process has been spawned, it is registered with the alias `Name`, subsequently calling the `init/1` function in the callback module `Mod`. The `init/1` function takes `Args`, the third parameter to the `start_link` call, as an argument, irrespective of whether it is needed. If no arguments are needed, the `init/1` function can ignore it with the don't care variable. Keep in mind that `Args` can be any valid Erlang term; you are not bound to using lists.



If `Args` is a (possibly empty) list, the list will be passed to `init/1` as a list, and not result in init of a different arity being called. For example, if you pass `[foo, bar]`, `init([foo,bar])` will be called, not `init(foo, bar)`. This is a common mistake developers do when transitioning from Erlang to OTP, as they confuse the properties of `spawn` and `spawn_link` with those of the behaviour `start` and `start_link` functions.

The `init/1` callback function is responsible for initializing the server state. In our example, this would entail creating the variable containing the lists of available and allocated frequencies. If successful, `init/1` returns `{ok, LoopData}`. If the startup fails, but you do not want to affect other processes, return `ignore`. If you want to affect other processes, return `{stop, Reason}`. We will cover `ignore` in [Chapter 8](#) and `stop` in “Termination” on page 83.

```

start() ->
  gen_server:start_link({local, frequency}, frequency, [], []).
  
```

```

init(_Args) ->
    Frequencies = {get_frequencies(), []},
    {ok, Frequencies}.

get_frequencies() -> [10,11,12,13,14,15].

```

In our example, `start_link/4` passes the empty list `[]` to `init/1`, which in turn uses the `_Args` don't care variable to ignore it. We could have passed any other Erlang term, as long as we make it clear to anyone reading the code that no arguments are needed. The atom `null` or the empty tuple `{}` are other favourites.

By passing `{timeout, Ms}` as an option in the `Opts` list, we allow our generic server `Ms` milliseconds to start up. If it takes longer, `start_link/4` returns the tuple `{error, timeout}` and the behaviour process is not started. No exception is raised. We will cover options in more detail in [Chapter 5](#)

Starting a generic server behaviour process is a synchronous operation. Only when `init/1` returns `{ok, LoopData}` to the server loop does the `gen_server:start_link/4` function return `{ok, Pid}`. It's important to understand the synchronous nature of `start_link` and its importance to a repeatable startup sequence. The ability to deterministically reproduce an error is important when troubleshooting issues which occur at startup. You could asynchronously start all of the processes, checking each afterwards to make sure they all started correctly. But as a result of changing scheduler implementations and configuration values running on multi-core architectures, deploying to different hardware or operating systems, or even the state of the network connectivity, the processes would not necessarily always initialise their state and complete the startup sequence in the same order. If all goes well, you won't have an issue with the variability inherent in an asynchronous startup approach, but if race conditions manifest themselves, trying to figure out what went wrong and when, especially in production environments, is not for the faint of heart. The synchronous startup approach implemented in `start_link` clearly ensures through its simplicity in that each process has started correctly before moving on to the next one, providing determinism and reproducible startup errors on a single node. If startup errors are influenced by external factors such as networks, external databases or the state of the underlying hardware or OS, try to contain them. In the cases where determinism does not help, a controlled startup procedure removes any element of doubt as to where the issue might be.

Message Passing

Having started our generic server and initialised its loop data, we are now going to look at how communication works. As you might have understood from the previous chapter, sending messages using `!` is out of fashion. OTP uses functional interfaces which provides a higher level of abstraction. The `gen_server` module exports functions that

allow us to send both synchronous and asynchronous messages, hiding the complexity of concurrent programming and error handling from the programmer.

Synchronous Message Passing

While Erlang has built-in asynchronous message passing as part of the language, there is nothing stopping us from implementing synchronous calls using existing primitives. This is what the `gen_server:call/2` function does. It sends a synchronous Message to the server, allows the server to handle it in a callback function and waits for a Reply. The Reply is passed as the return value to the call. The message and reply follows a specific protocol and contain a unique tag (or reference), matching the message and the response. Let's have a look at the `gen_server:call/2` function in more detail:

```
gen_server:call(Name, Message) -> Reply
```

`Name` is either the server pid or the registered name of the server process. The `Message` is an Erlang term that gets forwarded as part of the request to the server. Requests are received as Erlang messages, stored in the mailbox and handled sequentially. Upon receiving a synchronous request, the `handle_call(Message, _From, LoopData)` callback function is invoked in the callback module. The first argument is the `Message` passed to `gen_server:call/2`. The second argument `_From` contains a unique request identifier and information about the client. We will ignore it for the time being, binding it to a don't care variable. The third argument is the `LoopData`, originally returned by the `init/1` callback function. You should be able to follow the call flow in [Figure 4-3](#).

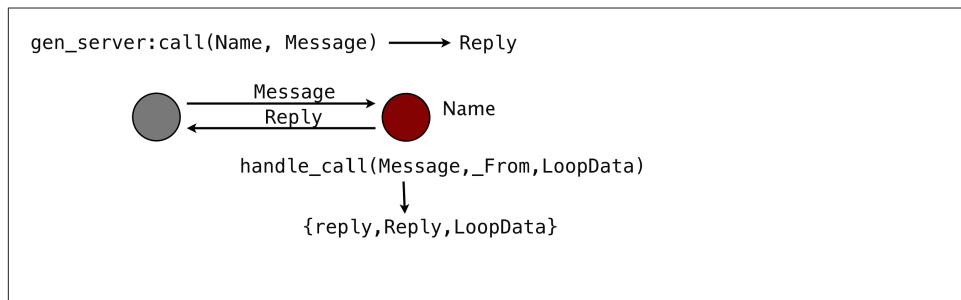


Figure 4-3. Synchronous Message Passing

The `handle_call/3` callback function contains all the code required to handle the request. It is good practice to have a separate `handle_call/3` clause for every request and to use pattern matching to pick the right one, instead of using a case statement to single out the individual messages. In the function clause, we would execute all of the code for that particular request, and when done, return a tuple of the format `{reply, Reply, NewLoopData}`. A callback module uses the atom `reply` to tell the `gen_server` that the second element `Reply` has to be sent back to the client process, becoming the return

value of the `gen_server:call/2` request. The third element `NewLoopData` is the callback module's new state, which the `gen_server` passes into the next iteration of its tail-recursive receive-evaluate loop. If `LoopData` does not change in the body of the function, we just return the original value in the reply tuple. The `gen_server` merely stores it without inspecting it or manipulating its contents. Once it sends back the reply tuple to the client, the server is then ready to handle the next request. If none are queued up in the process mailbox, the server is suspended waiting for a new request to arrive.

In our frequency server example, allocating a frequency would need a synchronous call, because the reply to the call will contain the allocated frequency. To handle the request, we call the internal function `allocate/2`, which you might recall returns `{NewFrequencies, Reply}`. `NewFrequencies` is the tuple containing the lists of allocated and available frequencies, while the `Reply` is the tuple `{ok, Frequency}` or `{error, no_frequency}`.

```
allocate() -> gen_server:call(frequency, {allocate, self()}).  
  
handle_call({allocate, Pid}, _From, Frequencies) ->  
    {NewFrequencies, Reply} = allocate(Frequencies, Pid),  
    {reply, Reply, NewFrequencies}.
```

Once completed, the `allocate/0` function called by the client process returns `{ok, Frequency}` or `{error, no_frequency}`. The updated loop data containing available and allocated frequencies is stored in the generic server receive-evaluate loop awaiting the next request.

Asynchronous Message Passing

If the client needs to send a message to the server but does not expect a reply, it can use asynchronous requests. This is done using the `gen_server:cast/2` library function:

```
gen_server:cast(Name, Message) -> ok
```

`Name` is the pid or the locally registered alias of the server process. `Message` is the term the client wants to send to the server. As soon as the `cast/2` call has sent its request, it returns the atom `ok`. On the server side, the request is stored in the process mailbox and handled sequentially. When it is received, `Message` is passed on to the `handle_cast/2` callback function, implemented by the developer in the callback module.

The `handle_cast/2` callback function take two arguments. The first is the `Message` sent by the client, while the second is the `LoopData` previously returned by the `init/1`, `handle_call/3` or `handle_cast/2` callbacks. This can be seen in [Figure 4-4](#).

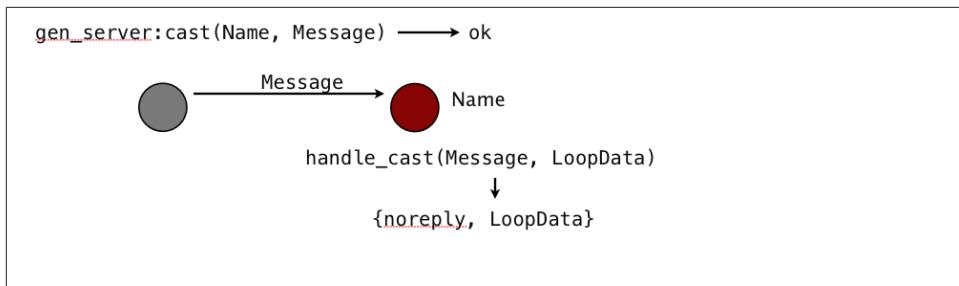


Figure 4-4. Asynchronous Message Passing

The `handle_cast/2` callback function has to return a tuple of the format `{noreply, NewLoopData}`. The `NewLoopData` will be passed as an argument to the next call or cast request.

In some applications, client functions return a hard-coded value, often the atom `ok`, relying on side effects executed in the callback module. Such functions could be implemented as asynchronous calls. In our frequency example, did you notice that `frequency:deallocate(Freq)` always returns the atom `ok`? We don't really care if handling the request is delayed because the server is busy with other calls, making it a perfect candidate for an example using a generic server cast:

```

deallocate(Frequency) ->
    gen_server:cast(frequency, {deallocate, Frequency}).

handle_cast({deallocate, Freq}, Frequencies) ->
    NewFrequencies = deallocate(Frequencies, Freq),
    {noreply, NewFrequencies};

```

The client function `deallocate/1` sends an asynchronous request to the generic server and immediately returns the atom `ok`. This request is picked up by the `handle_cast/2` function, which pattern matches the `{deallocate, Frequency}` message in the first argument and binds the loop data to `Frequencies` in the second. In the function body, it calls the helper function `deallocate/2`, moving `Frequency` from the list of allocated frequencies to the list of available ones. The return value of `deallocate/2` is bound to the variable `NewFrequencies`, returned as the new loop data in the `noreply` control tuple.

Note that we said that only in some applications do client functions ignore return values from server functions with side-effects. Pinging a server to make sure it is alive, for example, would rely on `gen_server:call/2` raising an exception if the server had terminated or if there were a delay, possibly as a result of heavy load, in handling the request and sending the response. Another example where synchronous calls are used is when there is a need to throttle requests and control the rate at which messages are sent to the server. We will discuss the need for this in [Chapter 5](#).

As with pure Erlang, call and casts should be abstracted in a functional API if used from outside the module. This gives you greater flexibility to change your protocol, debugging capabilities and hides information from the caller of the function. Place the client functions in the same module as the process, as it makes it easier to follow the message flow without jumping between modules.

Other Messages

OTP behaviours are implemented as Erlang processes. So while communication should ideally be through the protocols defined in the `gen_server:call/2` and `gen_server:cast/2` functions, that is not always the case. As long as the pid or registered name is known, there is nothing stopping a user from sending it a message using the `Name ! Message` construct. In some cases, Erlang messages are the only way to get information across to the generic server. For example, if the server is linked to other processes or ports but has called the `process_flag(trap_exit, true)` BIF to trap exits from those processes or ports, it might receive EXIT signal messages. Also, communication between processes and ports or sockets is based on message passing. Or finally, what if we are using a process monitor, monitoring distributed nodes or communicating with legacy, non-OTP compliant code?

These examples all result in our server receiving Erlang messages that do not comply with the internal OTP messaging protocol of the server. Compliant or not, if you are using features that can generate messages to your server, then your server code has to be capable of handling them. Generic servers provide a callback function that takes care of all of these messages. It is the `handle_info(Message, LoopData)` callback. When called, it has to return either the tuple `{noreply, NewLoopData}` or when stopping, `{stop, Reason, NewLoopData}`. It is common practice, even if you are not expecting any messages, to include this callback function. Not doing so and sending the server a non-OTP compliant message (they arrive when you least expect them!) would result in a runtime error and the server terminating, as the `handle_info/2` function would be called in the callback module, resulting in an undefined function error.

```
handle_info(_Msg, LoopData) ->
    {noreply, LoopData}.
```

We've kept our frequency server example simple. We ignore any message coming in, returning the unchanged `LoopData` in the `noreply` tuple. If you are certain you should not be receiving non-OTP messages, you could log such messages as errors. If we wanted to print an error message every time a process the server is linked to terminated abnormally, the code would look like this. We are assuming that the server in question is trapping exits:

```
handle_info({'EXIT', _Pid, normal}, LoopData) ->
    {noreply, LoopData};
handle_info({'EXIT', Pid, Reason}, LoopData) ->
    io:format("Process: ~p exited with reason: ~p~n",[Pid, Reason]),
```

```
{noreply, LoopData};  
handle_info(_Msg, LoopData) ->  
{noreply, LoopData}.
```



One of the downsides of OTP is the overhead resulting from the layering of the various behaviour modules and the data overhead required by the communication protocol. Both will affect performance. In the attempt to save a few microseconds from their calls, developers have been known to bypass the `gen_server` cast function and use the `Pid ! Msg` construct instead. Or even worse, embed `receive` statements in their callback functions to receive these messages. Don't do this! You will make your code hard to debug, support and maintain, lose many of the advantages OTP brings to the table, and get the authors of this book to stop liking you. If you need to shave off microseconds, optimize only when you know from actual performance measurements that your program is not fast enough.

Using From

What happens in a situation where two clients send a synchronous request to a server, but instead of immediately responding to each individually, the server has to wait for both requests before responding to the first? We demonstrate this in [Figure 4-5](#). This could be done for synchronization purposes or because the server needs the data from both requests.

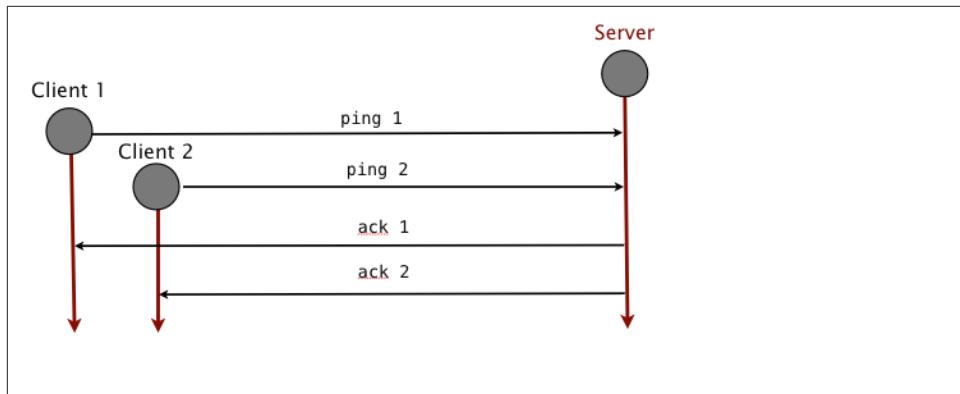


Figure 4-5. Rendezvous with Generic Servers

The solution to this problem is simple. Remember the `From` field in the `handle_call(Message, From, State)` callback function? Instead of returning a reply back

to the behaviour loop, we return `{noreply, NewState}`. We then use the `From` attribute and the function

```
gen_server:reply(From, Reply)
```

to send back the reply to the client when it suits us. In the case of having to synchronize two clients, it could be in the second `handle_call/3` callback, where `From` value for the first client is stored between the calls either as part of the `NewState` or in a table or database.

You can also use `reply/2` if a synchronous request triggers a time-consuming computation, and the only response the client is interested in is an acknowledgement that the request has been received and is in the process of being fulfilled, without having to wait for the whole computation to be completed. To send an immediate acknowledgment, the `gen_server:reply/2` can be used in the callback itself:

```
handle_call({compute, Data}, From, State) ->
    gen_server:reply(From, {reply, Data}),
    io:format("From: ~p~n",[From]),
    {noreply, compute(Data, State)};
```

Let's add this code in the ping server (assuming `compute/2` is implemented as a dummy function) and see what we get as a result:

```
1> gen_server:start({local, frequency}, frequency, [], []).
{ok,0.66.0}
2> gen_server:call(frequency, {compute, 123}).
From: {<0.31.0>,#Ref<0.0.0.261>}
{reply,123}
```

Note the value and format of the `From` argument we are printing in the shell. It is a tuple containing the client pid and a unique reference. This reference is used in a tag with the reply sent back to the client, ensuring that it is in fact the intended reply, and not a message confirming to the protocol sent from another process. Always use `From` as an opaque data type; don't assume it is a tuple, as its representation might change in future releases.

Unhandled Messages

Erlang uses selective receives when retrieving messages from the process mailbox. Allowing us to extract certain messages while leaving others unhandled comes with the risk of memory leakages. What happens if a message type is never read? Using Erlang without OTP, the message queue would get longer and longer, increasing the number of messages to be traversed before one is successfully pattern matched. This will manifest itself in the Erlang VM through high CPU usage as a result of the traversal of the mailbox, eventually causing the VM to run out of memory and possibly be restarted through `heart`, covered in [Chapter 11](#)

All of this is valid if we are using pure Erlang. OTP behaviours take a different approach. Messages are handled in the same order in which they are received. Start your frequency server, and try sending yourself a message you are not handling:

```
1> frequency:start().
{ok,<0.33.0>}
2> gen_server:call(frequency, foobar).

=ERROR REPORT==== 25-Jun-2012::10:05:36 ===
** Generic server frequency terminating
** Last message in was foobar
** When Server state == {data,[{"State",
    [{"available,[10,11,12,13,14,15]},
     {"allocated,[]"}]}]}
** Reason for termination ==
** {function_clause,[{frequency,handle_call,
    [foobar,
     {<0.31.0>,#Ref<0.0.0.31>},
     {[10,11,12,13,14,15],[]}],
    {gen_server,handle_msg,5},
    {proc_lib,init_p_do_apply,3}]}]
```

Probably not what you were expecting. The frequency server terminated with a `function_clause` runtime error, printing an error report.³ When you call a function, one of the clauses has to always match. Failure to do so results in a runtime error. When doing a `gen_server` call or cast, the message is always retrieved from the mailbox in the generic server loop, and the `handle_call/3` or `handle_cast/2` callback function is invoked. In our example, `handle_call(foobar, _From, LoopData)` doesn't match any of the clauses, causing the function clause error we've just viewed. The same would happen with a cast.

How do we avoid such errors? One option is to have a catch-all, where unknown messages are pattern matched to a don't care variable and ignored. This is specific to the application, and might or might not be the answer. A catch-all might be the norm with the `handle_info/2` callback when dealing with ports, sockets, links, monitors, and monitoring of distributed nodes where there is a risk of forgetting to handle a particular message not needed by the application. When dealing with call and cast, however, all requests should originate from the behaviour callback module and any unknown messages should be caught in early stages of testing.

If in doubt, don't be defensive, and instead make your server terminate when receiving unknown messages. Treat these terminations as bugs, and either handle the message or correct it at the source. If you do decide to ignore unknown messages, don't forget to log them.

3. If you run this example in the shell, you will also get an error report from the shell itself terminating as a result of the exit signal propagating through the link.

Termination

What if we want to stop a generic server? So far, we've seen the callback functions `init/1`, `handle_call/3`, and `handle_cast/2` return `{ok, LoopData}`, `{reply, Reply, LoopData}` and `{noreply, LoopData}` respectively. If instead of returning these tuples, let:

- `init/1` return `{stop, Reason}`
- `handle_call/3` return `{stop, Reason, Reply, LoopData}`
- `handle_cast/2` return `{stop, Reason, LoopData}`

These return values terminate with the same behaviour as if `exit(Reason)` was called. In the case of calls and casts, before exiting, the callback function `terminate(Reason, LoopData)` is called. It allows the server to clean up after itself before being shut down. Any value returned by `terminate/2` is ignored. In the case of `init`, `stop` should be returned if something fails when initialising the state. As a result, `terminate/2` will not be called. If we return `{stop, Reason}` in the `init/1` callback, the `start_link` function returns `{error, Reason}`.

In our frequency server example, the `stop/0` client function sends an asynchronous message to the server. Upon receiving it, the `handle_cast/2` callback returns the tuple with the `stop` control atom, which in turn results in the `terminate/2` call being invoked. Have a look at the code:

```
stop() -> gen_server:cast(frequency, stop).

handle_cast(stop, LoopData) ->
    {stop, normal, LoopData}.

terminate(_Reason, _LoopData) ->
    ok.
```

To keep the example simple, we've left `terminate` empty. In an ideal world, we would probably have killed all of the client processes that were allocated a frequency, thereby terminating the calls and ensuring that upon a restart, all frequencies are available.

Look at the message `gen_server:cast/2` sends to the frequency server. You'll notice it is the atom `stop`, pattern matched in the first argument of the `handle_cast/2` call. The message has no meaning other than the one we give to it in our code. We could have sent any atom; `gen_server:cast(frequency, donald_duck)`. Pattern matching `donald_duck` in the `handle_cast/2` would have given us the same result. The only `stop` that has a special meaning is the one that occurs in the first element of the tuple returned by `handle_cast/2`, as it is interpreted in the receive-evaluate loop of the generic server.

If you are shutting down your server as part of your normal work flow (e.g. the socket it is handling has been closed, or the hardware it controls and monitors is shutting down), it is good practice to set your `Reason` to `normal`. A non-normal reason, while perfectly

acceptable, will result in error reports being logged by the SASL logger. These entries might overshadow those of real crashes. The SASL logger is another freebie you get when using OTP. We will cover it in [Chapter 9](#).

Although servers can be stopped normally by returning the `stop` tuple, there might be cases when they terminate as the result of a run-time error. In these cases, if the generic server is trapping exits (by having called the `process_flag(trap_exit, true)` BIF), `terminate/2` will also be called. If you are not trapping exits, the process will just terminate *without* calling `terminate/2`.

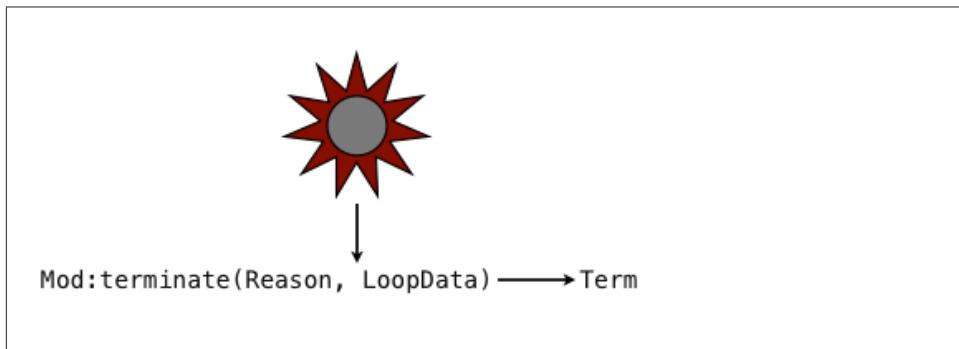


Figure 4-6. Abnormal Server Termination

If you want the `terminate/2` function to execute after abnormal terminations, you have to set the `trap_exit` flag. If it is not set, a supervisor or linked process might bring the server down without allowing it to clean up. Having said this, always check the context for termination. If a run-time error has occurred, clean up the server state with extreme care, as you might end up corrupting your data, and so setting up for more run-time errors after the server has been restarted. When restarting, you should aim to recreate the server state from correct (and unique) sources of data, not a copy you stored right before the crash as it might have been corrupted by the same fault that caused the crash.

Call Timeouts

When sending synchronous messages to your server using a `gen_server` call, you should expect a response within milliseconds. But what if there is a delay in sending the response? Your server might be extremely busy handling thousands of requests, or there might be bottlenecks in external dependencies such as databases, authentication servers, TCP/IP networks, or any other resource or API taking its time to respond. OTP behaviours have a built-in timeout in their synchronous `gen_server:call/2` APIs which is set to five seconds. This should be enough to cater to most queries in any soft real-time system, but there are borderline cases that need to be handled differently. If you are sending a synchronous request using OTP behaviours and have not received a re-

sponse within five seconds, the client process will raise an exception. Let's try it out in the shell with the following callback module:

```
-module(timeout).
-behaviour(gen_server).

-export([init/1, handle_call/3]).

init(_Args) ->
{ok, null}.

handle_call({sleep, Ms}, _From, LoopData) ->
timer:sleep(Ms),
{reply, ok, LoopData}.
```

In the `gen_server:call/2` function, we send a message of the format `{sleep, Ms}` where `Ms` is a value used in the `timer:sleep/1` call executed in the `handle_call/3` callback. Sending a value larger than 5000 milliseconds should cause the `gen_server:call/2` function to raise an exception, as the default timeout is set to that value. Let's try it out in the shell. We are assuming that the `timeout` module is already compiled, so as to avoid the compiler warnings from the callback functions we have omitted:

```
1> gen_server:start_link({local, timeout}, timeout, [], []).
{ok,<0.66.0>}
2> gen_server:call(timeout, {sleep, 1000}).
ok
3> catch gen_server:call(timeout, {sleep, 5001}).
{'EXIT',{timeout,[{gen_server,call,[{timeout,{sleep,5001}}]}]}
4> flush().
Shell got {#Ref<0.0.0.300>,ok}
5> gen_server:call(timeout, {sleep, 5001}).
** exception exit: {timeout,[{gen_server,call,[{timeout,{sleep,5001}}]}]
   in function  gen_server:call/2
6> catch gen_server:call(timeout, {sleep, 1000}).
{'EXIT',{noproc,[{gen_server,call,[{timeout,{sleep,1000}}]}]}
```

We start the server, and in the shell command 2, send a synchronous message telling the server to sleep for 1000 milliseconds before replying with the atom `ok`. As this is within the five seconds default timeout, we get our response back. But in shell command 3, we raise the timeout to 5001 milliseconds, causing the `gen_server:call/2` function to raise an exception. In our example, shell command 3 catches the exception, allowing the client function to handle any special cases that might arise as the result of the timeout.

If you decide to catch exceptions arising as the result of a timeout, be warned: if the server is alive but busy, it will send back a response after the timeout exception has been raised; the response has to be handled. If the client is an OTP behaviour, the exception will result in the `handle_info/2` call being invoked. If this call has not been implemented, the client process will crash. Worse things might happen if you did not remember to handle this specific message, mistaking it for something else.

If the call is from a pure Erlang client, the exception will be stored in the client mailbox and never handled. Having unread messages in your mailbox will consume memory and slow down the process when new messages are received, as the littering messages need to be traversed before new ones will be pattern matched. Not only that, but sending a message to a process with a large number of unread messages will slow down the sender, because the send operation will consume more reductions. This will have a knock-on effect, potentially triggering more timeouts and further growing the number of littering messages in the client mailbox.

The performance penalty when sending messages to a process with a long message queue does not apply to behaviours synchronously responding to the process where the request originated. If the client process has a long message queue, thanks to compiler and virtual machine optimizations, the receive clause will match the reply without having to traverse the whole message queue.

We have proof of this memory leak in shell command 4, where unread messages are flushed. Had we not flushed the message, it would have remained in the shell's mailbox. Throughout this book, we keep reminding you not to handle corner cases and unexpected errors in your code, as you run the risk of introducing more bugs and errors than you actually solve. This is a typical example where side effects resulting from these timeouts will probably manifest themselves only under extreme load in a live system.

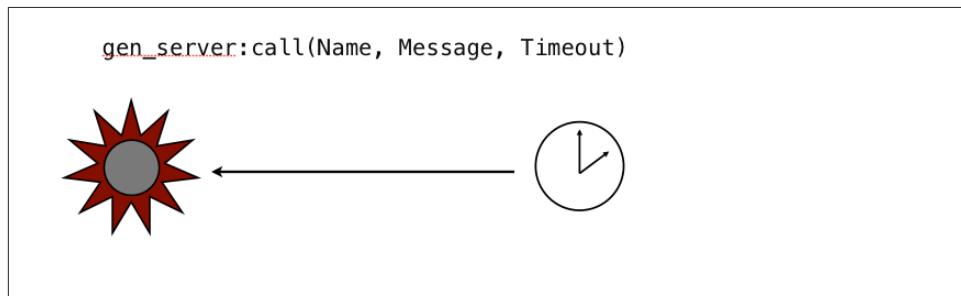


Figure 4-7. Server Timeouts

Now have a look at shell command 5 and [Figure 4-7](#). We have a call that causes the client process to crash, because it is executed outside the scope of a try-catch statement. In a majority of cases, if your server is not responding for any (possibly unknown) reason, making the client process terminate and letting the supervisor deal with it is probably the best approach. In this example, the shell process terminates and is immediately restarted. The timeout server sends a response to the old client (and shell) Pid after 5001 milliseconds. As this process does not exist anymore, the message is discarded. So why is shell command 6 failing with reason *noproc*? Have a look at the sequence of shell commands and see if you can figure it out before reading on.

When we started the server, we linked it to the shell, making the shell process act as both the client and the parent. The timeout server terminated after we executed a `gen_server:call/2` outside of the scope of a try-catch in shell command 5. Because the server is not trapping exits, when the shell terminated, the EXIT signal propagated to the server, causing it to also terminate. In normal circumstances, the client and the parent of server that links to it would not be the same process, so this would not occur. These issues tend to show up when testing behaviours from the shell, so keep them in mind when working on your exercises.

So, how do we overcome the five-second default value in behaviours? Easy: we set our own timeout. In generic servers, we do this using the

```
gen_server:call(Server, Message, TimeOut) -> Reply
```

function call, where `TimeOut` is the value in milliseconds or the atom `infinity`.

A client call will often consist of a chain of synchronous requests to several, potentially distributed, behaviour processes. They might in turn send requests to external resources. More often than not, choosing timeout values becomes tricky, as these processes are accessing services and APIs provided by third parties, completely out of your control. Systems that have been known to respond in milliseconds to the majority of the requests can take seconds or even minutes under extreme loads. The throughput of your system counted in operations per second might still be the same, but when there is a higher load—possibly many orders of magnitude higher—going through it, the latency of the individual requests will be higher.

The only answer to the question of what `TimeOut` you should set is to start with your external requirements. If a client specifies a 30-second timeout, start with it and work your way through the chain of requests. What are the guaranteed response times of your external dependencies? How will disk access and I/O respond under extreme load? What about network latency? Spend lots of time stress testing your system on the target hardware and fine-tune your values accordingly. And when unsure, start with the 5000 milliseconds default value. Use the value `infinity` with extreme care, avoiding it altogether where possible.

Deadlocks

Picture two generic servers in a badly designed system. Server A does a synchronous call to server B. Server B receives the request, and through a series of calls in other modules ends up (possibly unknowingly) executing a synchronous callback to server A. Observing [Figure 4-8](#), this problem is resolved not through complex deadlock prevention algorithms, but through timeouts.

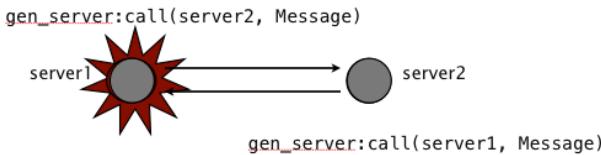


Figure 4-8. Generic Server Deadlocks

If server1 has not received a response within 5000 milliseconds, it terminates, causing server2 to terminate as well. Depending on who gets there first, the termination is triggered either through the monitor signal or through a timeout of its own. If more processes are involved in the deadlock, the termination will propagate to them as well. The supervisor will receive the EXIT signals and restart the servers accordingly. The termination is stored in a log file where it is hopefully detected, resulting in the bug leading to the deadlock being fixed.

Strategies for Avoiding Deadlocks

Despite the ease of creating deadlocks, they are extremely rare, no matter how complex the program might be. This has to do with how the systems are architected, the concurrency is modeled, and dependencies among processes and applications are handled. A standard practice when dealing with static processes who are not started and terminated dynamically is to allow synchronous calls to be made only to processes that were started before the process making the call. Calls from older processes to younger ones may only be asynchronous. If a reply is required from the younger process, it sends it back (through a possibly asynchronous) callback function. The start order of static processes is defined in supervision trees, which also happens to be the order used with dynamic processes. This will become clear when we cover supervision trees and restart orders in [Chapter 8](#). You need to keep it in mind when processes are grouped into supervision trees, when supervision trees are grouped into applications, and when application start orders are defined. The lack of shared memory and critical sections further remove the danger of deadlocks. Experienced Erlang programmers will by default ensure that their programs are designed to avoid deadlocks, often without having to think about it. Newbies, however, need to find a suitable strategy in the initial design phase of the system and stick to it.

In 17 years of working with Erlang,⁴ I've come across only one deadlock. Process A synchronously called Process B, which in turn, did an RPC to another node that resulted in a synchronous call to process C. Process C synchronously called process D, which did another RPC back to the first node. This RPC resulted in a synchronous callback to process A, who was still waiting for a response back from B. We discovered this deadlock when integrating the two nodes for the first time, and it took us five minutes to solve. Process A should have called B asynchronously, and process B should have responded back to A with an asynchronous callback. So while there is a risk of deadlocks, if you approach the problem right, it is minimal, as the largest cause of deadlocks occurs when controlling execution and failure in critical sections, something the shared nothing approach in Erlang provides plenty of alternatives to.

Generic Server Timeouts

Picture a generic server whose task is to monitor and communicate with a particular hardware device. If the server has not received a message from the device within a predefined timeout, it should send a ping request and ensure the device is alive. These ping requests can be triggered by internal timeouts, created by adding a timeout value in the control tuples sent back as a result of the behaviour callback functions:

```
init/1      -> {ok, LoopData, Timeout}
handle_call/3 -> {reply, Reply, LoopData, Timeout}
handle_cast/2 -> {noreply, LoopData, Timeout}
handle_info/2 -> {noreply, LoopData, Timeout}
```

The value `Timeout` is either an integer in milliseconds or the atom `infinity`. If the server does not receive a message in `Timeout` milliseconds, it receives a `timeout` message in its `handle_info/2` callback function. Returning `infinity` is the same as not setting a timeout value. Let's try it with a simple example where every 5000 milliseconds, we generate a timeout that retrieves the current time and prints the seconds. We can pause the timer and restart it by sending the synchronous messages `start` and `pause`:

```
-module(ping).
-behaviour(gen_server).

-export([init/1, handle_call/3, handle_info/2]).
-define(TIMEOUT, 5000).

init(_Args) ->
{ok, null, ?TIMEOUT}.

handle_call(start, _From, LoopData) ->
{reply, started, LoopData, ?TIMEOUT};
```

4. I'm the author who in the previous book caused the nationwide data outage in a mobile network.

```

handle_call(pause, _From, LoopData) ->
{reply, paused, LoopData}.

handle_info(timeout, LoopData) ->
{_Hour, _Min, Sec} = time(),
io:format("~2.w~n",[Sec]),
{noreply, LoopData, ?TIMEOUT}.

```

Assuming the ping module is compiled, we start it and generate a timeout every 5 seconds. We can suspend the timeout by sending it the pause message, which when handled in the second clause of the handle_call/3 function does not include a timeout in its return tuple. We turn it back on with the start message.

```

1> gen_server:start({local, ping}, ping, [], []).
{ok,<0.38.0>}
22
27
2> gen_server:call(ping, pause).
paused
3> gen_server:call(ping, start).
started
51
56
4> gen_server:call(ping, start).
started
4

```

Because we set a relatively high timeout, we do not generate a timeout message at 5000 millisecond intervals. We send a timeout message *only* if a message has not been received by the behaviour. If a message is received, as is happening with shell command 4 in our example, the timer is reset. If you need timers that may not be reset or have to run at regular intervals irrespective of incoming messages, use functions such as erlang:send_after/3 or those provided by the timer module including apply_after/3, send_after/2, apply_interval/4, and send_interval/2.

Hibernating Behaviours

If instead of a timeout value or the atom infinity we return the atom hibernate, the server will reduce its memory footprint and enter a wait state. Hibernate will discard the call stack and run a full sweep garbage collection, placing everything in one continuous heap. The allocated memory is then shrunk to the size of the data on the heap. The server will remain in this state until it receives a new message. Use hibernation only if you do not expect the behaviour to receive any messages in the foreseeable future and need to economize on memory, not for servers receiving frequent bursts of messages.



There is a cost associated with hibernating processes, as it will involve a full sweep garbage collection prior to hibernating, and one soon after the process wakes up. Using it as a preemptive measure is dangerous, especially if your process is busy, as it might (and probably will) cost more to hibernate the process than to just leave it as is. The only way to know for sure is to benchmark your system under stress and demonstrate a gain in performance alongside a substantial reduction in memory usage. Add it as an afterthought only if you know what you are doing. If in doubt, don't do it!

Going Global

Behaviour processes can be registered locally or globally. In our examples, they have all been registered locally using a tuple of the format `{local, ServerName}`, where `ServerName` is an atom denoting the alias. This is equivalent to registering the process using `register(ServerName, Pid)` BIF. But what if we want location transparency in a distributed cluster?

Globally registered processes piggyback on the global name server, which makes them transparently accessible in a cluster of (possibly partitioned) distributed nodes. The name server stores local replicas of the names on every node and monitors node health and changes in connectivity, ensuring there is no central point of failure. You register a server globally by using the `{global, Name}` tuple as an argument to the server name field. It is equivalent to registering the process using `global:register_name(Name, Pid)`. Use the same tuple in your synchronous and asynchronous calls:

```
gen_server:start_link({global, Name}, Mod, Args, Opts) -> {ok, Pid} |  
                                         ignore |  
                                         {error, Reason}  
  
gen_server:call({global, Name}, Message) -> Reply  
gen_server:cast({global, Name}, Message) -> ok
```

There is an API that allows you to replace the global process registry with one you have implemented yourself. You can create your own when the functionality provided by the `global` module is not enough, or when you want a different behaviour that caters for different network topologies. You need to provide a callback module, `Module` say, that exports the same functions and return values defined in the `global` module, namely `register_name/2`, `unregister_name/1`, `whereis_name/1`, and `send/2`. Name registration then used the tuple `{via, Module, Name}`, and starting your process using `{via, global, Name}` is the same as registering it globally using `{global, Name}`. For globally registered processes, the `Name` does not have to be an atom; rather, any Erlang term is valid. Once you have your callback module, you can start your process and send messages using:

```
gen_server:start_link({via, Module, Name}, Mod, Args, Opts) -> {ok, Pid}
gen_server:call({via, Module, Name}, Message) -> Reply
gen_server:cast({via, Module, Name}, Message) -> ok
```

In the remainder of the book, we will aggregate `{via, Module, Name}`, `{local, Name}` and `{global, Name}` using `NameScope`. Most servers are registered locally, but depending on the complexity of the system and clustering strategies, `global` and `via` are used as well.

When communicating with behaviours, you can use their pid instead of their registered aliases. Registering behaviours is not mandatory, allowing many instances of the same behaviour to run in parallel. When starting the behaviours, just omit the name field:

```
gen_server:start_link(Mod, Args, Opts) -> {ok, Pid} |
                                             ignore |
                                             {error, Reason}
```

If you broadcast a request to all servers within a cluster of nodes, you can use the generic server `multi_call/3` call if you need results back and `abcast/3` if you don't. On the servers of the individual nodes, requests are handled in the `handle_call/3` and `handle_cast/2` callbacks respectively. When broadcasting asynchronously with `abcast`, no checks are made to see whether or not the nodes are connected and still alive. Requests to nodes that can not be reached are simply thrown away.

```
gen_server:multi_call(Nodes, Name, Request [, Timeout]) -> {[{Node, Reply}], BadNodes}
gen_server:abcast(Nodes, Name, Request) -> abcast
```

Linking Behaviours

When you start behaviours in the shell, you link the shell process to them. If the shell process terminated abnormally, its `EXIT` signal will propagate to the behaviours it started and cause them to terminate. Generic servers can be started without linking them to their parent by calling `gen_server:start/3` and `gen_server:start/4`. Use these functions with care, and preferably only for development and testing purposes, because behaviours should always be linked to their parent.

```
gen_server:start(NameScope, Mod, Args, Opts) -> {ok, Pid} |
gen_server:start(Mod, Args, Opts)           -> {error, {already_started, Pid}}
```

Erlang systems will operate for years without rebooting the computers they run on. They can continue even during software upgrades for bug fixes, feature enhancements and new functionality, and behaviours terminating abnormally and being restarted. When shutting down a subsystem, you need to be 100% certain that all processes associated with that subsystem are terminated, and avoid leaving any orphan processes lingering. The only way to do so with certainty is using links. We will go into more detail when we cover supervisor behaviours in [Chapter 8](#).

Summing Up

In this chapter, we have introduced the most important concepts and functionality in generic server behaviours, the behaviour behind all behaviours. You should by now have a good understanding of the advantages of using the `gen_server` behaviour instead of rolling your own. We have covered the majority of functions and associated callbacks needed when using the `gen_server` behaviour. Although you do not need to understand everything that goes on behind the scenes, we hope you now have an idea and appreciation that there is more than meets the eye. The most important functions we have covered include:

Table 4-1. gen_server callbacks

gen_server function or action	gen_server callback function
<code>gen_server:start/3</code> , <code>gen_server:start/4</code> , <code>gen_server:start_link/3</code> , <code>gen_server:start_link/4</code>	<code>Module:init/1</code>
<code>gen_server:call/2</code> , <code>gen_server:call/3</code> , <code>gen_server:multi_call/2</code> , <code>gen_server:multi_call/3</code>	<code>Module:handle_call/3</code>
<code>gen_server:cast/2</code> , <code>gen_server:abcast/2</code> , <code>gen_server:abcast/3</code>	<code>Module:handle_cast/2</code>
Pid ! Msg, monitors, exit messages, messages from ports and sockets, node monitors, and other non-OTP messages	<code>Module:handle_info/2</code>
Triggered by returning {stop, ...} or when terminating abnormally while trapping exits	<code>Module:terminate/2</code>

When compiling behaviour modules, you will have seen a warning over the missing `code_change/3` callback. We will cover it in [Chapter 11](#) when looking at release handling and software upgrades. In the next chapter, while using the generic server behaviour as an example, we look at advanced topics and behaviour-specific functionality that comes with OTP.

At this point, you will want to make sure you review the manual pages for the `gen_server` module. If you are feeling brave, read the code in the `gen_server.erl` source file, and the source for the `gen` helper module. Having read chapters 3 and 4 and understood the corner cases, you will discover the code is not as cryptic as might first appear.

What's Next?

The next chapter contains odds and ends that allow you to dig deeper into behaviours. We start investigating the built-in tracing and logging functionality we get from using them. We finally introduce you to the `Opts` flags in the start functions that until now you've always set to the empty list. The flags allow you to fine tune performance and memory usage. Finally, we cover the most common source of bottlenecks, helping you determine if you should be using synchronous or asynchronous calls. So read on, as interesting things are in store in the next page.

Controlling OTP Behaviours

In the previous chapter, we covered the highlights of the gen_server behaviour. You should by now have implemented your first client-server application and started to build an idea of how OTP behaviours allow you to reduce your code base by allowing you to focus on the specifics of what your system has to do. This chapter digs deeper into behaviours, exploring some of the advanced topics intermixed with built-in functionality. While we are focusing on generic servers, most of what we write will apply to many of the other behaviours, including behaviours you will implement yourself. Read with care, as we will be referencing this chapter often in the remainder of this book.

The sys Module

In the previous chapters, we've mentioned the built-in functionality you get as a result of using OTP behaviours and the ease with which you can add your own features. Most of the functionality we cover is accessed through the `sys` module, allowing you to generate trace events and inspect the behaviour status. There are also options you can include when starting the behaviour to optimise memory management.

Tracing and Logging

Let's find out how built-in tracing works by running a little example. Start your frequency server in the shell, and using the `sys` module, try the following:

```
1> frequency:start().
{ok,<0.38.0>}
2> sys:trace(frequency, true).
ok
3> frequency:allocate().
*DBG* frequency got call {allocate,<0.31.0>} from <0.31.0>
*DBG* frequency sent {ok,10} to <0.31.0>, new state {[11,12,13,14,15],[{10,<0.31.0>}]}
{ok,10}
4> frequency:deallocate(10).
```

```

*DBG* frequency got cast {deallocate,10}
ok
*DBG* frequency new state {[10,11,12,13,14,15],[]}
5> sys:trace(frequency, false).
ok

```

By turning on the trace flags for our frequency allocator, we are able to generate print-outs of system events, including messages and state changes. Our example pipes the messages out to the shell. If we instead use the `sys:log/2` call, we store them in the server loop. We can display them (through a `print` flag) at any time or retrieve them (through a `get` flag) as an Erlang data structure:

```

6> sys:log(frequency, true).
ok
7> {ok, Freq} = frequency:allocate(), frequency:deallocate(Freq).
ok
8> sys:log(frequency, print).
*DBG* frequency got call {allocate,<0.31.0>} from <0.31.0>
*DBG* frequency sent {ok,10} to <0.31.0>, new state {[11,12,13,14,15],[{10,<0.31.0>}]}
*DBG* frequency got cast {deallocate,10}
*DBG* frequency new state {[10,11,12,13,14,15],[]}
ok
9> sys:log(frequency, get).
{ok,[{{in,['$gen_call',{<0.31.0>,#Ref<0.0.0.97>}],
       {allocate,<0.31.0>}}},
      {frequency,#Fun<gen_server.0.81308364>},
      {{out,{ok,10},<0.31.0>,[{10,11,12,13,14,15},[]]},
       {frequency,#Fun<gen_server.6.5805831>},
       {{in,['$gen_cast',{deallocate,10}]}}},
      {frequency,#Fun<gen_server.0.81308364>},
      {{noreply,[{10,11,12,13,14,15},[]]},
       {frequency,#Fun<gen_server.4.63595447>}}]}
10> sys:log(frequency, false).
ok

```

When you store trace events in the server loop by using the `sys:log/2` call, the default number of events stored is 10. You can override this number by passing the `{true, Int}` flag when enabling the logging. `Int` is an integer denoting the new default number of events you want to store. When you plan to deal with large volumes of debug messages or leave debugging turned on for a long time, use `sys:log_to_file/2` to pipe the messages to a text file.

System Messages

Have a look at the return value of shell command 9 in the previous example. If we pass the `get` flag to `sys:log/2`, we get back a list of system events. The forms of the events in the log depend on the process producing them, but generally each event contains a system message with one the following forms:

{in, Msg}

This system message is triggered when a message (including a timeout) is sent to the gen_server. Msg includes any construct that is part of the OTP message protocol, for example {'\$gen_cast', Msg} for casts and {'\$gen_call',{Pid, Ref}, Msg} for calls. For any regular Erlang term sent as a message to a gen_server process, Msg will simply be that term.

{out, Msg, To, State}

This is generated when replying to the client using the {reply, Reply, NewState} control tuple, but is not generated for replies sent via gen_server:reply/2. Msg is the reply sent to the client, and To is the pid of the client. State is the same as NewState specified in the reply tuple.

term()

System messages of any format are allowed. For example, the return value of shell command 9 includes the message {noreply, {[10,11,12,13,14,15],[]}} which is the result of handle_cast/2 after handling the deallocate cast. The second element of the noreply tuple is the new state of the gen_server.

Note that the R16B03 documentation for the sys module also specifies {in, Msg, From} and {out, Msg, To} as valid system messages, but these are not used by any standard behaviours.

Your Own Trace Functions

You can implement your own trace functions by implementing your own Fun that gets triggered when a system event takes place. You can pattern match on the events, taking any course of action you like. Trace functions can generate your own debug printouts, turn on low-level traces using dbg or the trace BIFs, enable logging of particular information, run diagnostic functions, or execute any other code you might need, or none at all.

The following example keeps a counter for every time a client is refused a frequency and prints a warning message¹. Note how we achieve this without touching the original frequency code:

```
11> F = fun(Count,{out, {error, no_frequency}, Pid, _LoopData}, ProcData) ->
    io:format("/*DBG* Warning, Client ~p refused frequency! Count:~w~n", [Pid, Count]),
    Count + 1;
    (Count, _, _) ->
    Count
end.
#Fun<erl_eval.18.105910772>
```

1. Note that the io:format/2 executed in the fun attaches itself to the group leader of the traced behaviour, causing warnings to be printed in the local shell. If you connect from a remote shell, you will not be able to see them.

```

12> sys:install(frequency, {F, 1}).
ok
13> frequency:allocate(), frequency:allocate(), frequency:allocate(),
    frequency:allocate(), frequency:allocate(), frequency:allocate().
{ok,15}
14> frequency:allocate().
*DBG* Warning, Client <0.31.0> refused frequency! Count:1
{error,no_frequency}
15> frequency:allocate().
*DBG* Warning, Client <0.31.0> refused frequency! Count:2
{error,no_frequency}
16> sys:remove(frequency, F).
false
17> frequency:allocate().
{error,no_frequency}

```

Let's look at this example in more detail. We create a fun `F` which takes three arguments. The first, `Count`, is the state of the debug function, passed between calls. The second argument is the system message, in which we pattern match on outbound messages of the format `{error, no_frequency}`. The third argument `ProcData` is specific to the behaviour being traced; for example, for a `gen_server` it's either the registered name of the process or its pid, whereas for a `gen_fsm` it is a tuple of the process name or pid and the current statename of the FSM. All other system messages are ignored by the second clause of the `F` function. We set the state of the debug function `Count` to the integer 1 in the second element of the tuple of the `sys:install/2` call in shell command 12. In this command, we also pass on the fun `F` to the frequency server, enabling the debug printout. We continue by calling `frequency:allocate/0` enough times to run out of frequencies, triggering the debug printout twice and increasing the counter. Every time it is executed, `F` returns the `Count` state variable, incremented by 1 if the first clause pattern matches or unchanged if the second clause matches. Returning the atom `done` in the debug function is equivalent to disabling the function by calling `sys:remove/2`, as shown in command line 16.

Statistics, Status, and State

The `sys` module also lets you collect general statistics on behaviours as well as retrieve internal information on their internal state, including loop data, without having to re-invent the wheel or implement anything new. Let's have a look:

```

28> sys:statistics(frequency, true).
ok
29> frequency:allocate().
{ok,10}
30> sys:statistics(frequency,get).
{ok,[{start_time,{{2012,6,23},{18,50,0}}},  

     {current_time,{{2012,6,23},{18,51,3}}},  

     {reductions,31},  

     {messages_in,1},

```

```

  {messages_out,0}}]}
31> sys:statistics(frequency, false).
ok
32> sys:get_status(frequency).
{status,<0.38.0>,
  {module,gen_server},
  [{{'$ancestors',[<0.31.0>]},{'$initial_call',{frequency,init,1}}},
   {running,<0.31.0>,[]},
   [{header,"Status for generic server frequency"},{data,[{"Status",running},{ "Parent",<0.31.0>},{ "Logged events",[]}]}],
   {data,[{"State",[[11,12,13,14,15],[{10,<0.31.0>}]]}]}]}

```

While `sys:statistics/2` returns a list of self-explanatory tagged values, the tuple returned by `sys:get_status/1` is not as obvious. It returns a tuple of the format:

```
{status, Pid, {module,Mod}, [ProcessDictionary, SysState, Parent, Dbg, Misc]}
```

where `Pid` and `Mod` are the behaviour's process identifier and module respectively. The `ProcessDictionary` is a list of key-value tuples. Note that while we do not use the process dictionary in our frequency server example, the `gen_server` library module and other behaviours we have yet to cover all do.

`SysState` tells us whether the behaviour's state is running or suspended. By calling `sys:suspend/1` and `sys:resume/1`, we can stop the behaviour from handling normal messages, ensuring only system messages to be handled. Usually you suspend a process when upgrading software using the OTP-specified upgrade capabilities. You might also suspend the process when defining your own behaviours, but most probably not when using standard behaviours. The only way you should suspend Erlang processes in your programs is by using receive clauses when none of the messages in the mailbox match. Using the `sys:suspend/1` call in your code is a no no!

`Parent` is the parent `Pid`, needed by behaviour processes that trap exits. If the parent terminates, the behaviour processes have to terminate as well. In this example, `Parent` is the shell process ID. `DbgFlag` holds the trace and statistics flags, which at the time we retrieved the status had all been turned off, hence the empty list.

Finally, `Misc` is a list of tagged tuples that contain behaviour-specific information. The contained items vary among behaviours, and you are able to override them yourself by providing an optional callback function in your behaviour callback module. When working with generic servers, the most important information in `Misc` is the loop data. This will, however, vary depending on the behaviour. You can influence the contents of the `Misc` value yourself by providing an optional callback function in your behaviour callback module, using the function to format the `{data, [{"State", ...}]} field to a value the end user might find simpler, more meaningful, or more helpful:`

```

...
-export([format_status/2]).
...

format_status(Opt, [ProcDict, {Available, Allocated}]) ->
    {data, [{"State", [{available, Available}, {allocated, Allocated}]}]}.

```

If Opt is the atom `normal`, it tells us the status is being retrieved as a result of the `sys:get_status/1` call. If the behaviour is terminating abnormally and the status is being retrieved to incorporate it in an error report, Opt is set to `terminate`.

`ProcDict` is a list of key-value tuples containing the process dictionary. In the earlier example, the new state would be:

```
 {data,[{"State", [{available, [11,12,13,14,15]}, {allocated, [10,<0.31.0>]}]}]}
```

While it is not mandatory to return a tuple of the format `{data, [{"State", State}]}}, it is recommended in order to stay consistent with what is currently in use.`

To examine just the loop data stored in the behaviour process by the callback module, use `sys:get_state/1`:

```

34> sys:get_state(frequency).
{[11,12,13,14,15],[{10,<0.31.0>}]}

```

This handy method allows you to avoid having to extract the loop data from the results of `sys:get_status/1`, something that's often difficult to do while debugging interactively in the shell. The `sys:get_state/1` call is intended only for debugging, in fact, as is the corresponding function `sys:replace_state/2`, which allows you to replace the loop state of a running behaviour process. For example, imagine you are debugging in the shell and you want to quickly check the result of the frequency server running out of frequencies. You could either invoke `frequency:allocate/0` repeatedly to exhaust the available frequencies—something simple to do when there are only a few available as in our example but much more difficult if thousands or simply modify the loop data to artificially exhaust them:

```

35> sys:replace_state(frequency, fun(_) -> {},[]).
{[],[]}
36> frequency:allocate().
{error,no_frequency}

```

Replacing the loop data requires passing a function that receives the current value of the loop data and returns a new value. This allows you to easily modify only the necessary portions of a complex loop data value. In this example, though, we just ignore the old state and replace it wholesale with a tuple of two empty lists. The `sys:replace_state/2` function returns the new loop data. Since the new value in our example specifies an empty list of available frequencies, the next call to `frequency:allocate/0` results in `{error, no_frequency}` as expected. Since the replacement loop data effectively dis-

ables all frequency allocation and deallocation, it is best to immediately either replace the loop data with something more practical, or restart the frequency server.

The sys module Recap

Summing up, here are the functions in the sys module we have covered. Note the notation we are using for [,Timeout] in our function descriptions. It means an optional argument to the call, defining functions of arity 2 and 3. Because these functions are nothing other than synchronous calls to our behaviour, using Timeout allows us to override the 5-second default timeout time with a value more suited for our application.

```
sys:trace(Name,TraceFlag [,Timeout]) -> ok

sys:log(Name,LogFlag [,Timeout]) -> ok | {ok, EventList}
sys:log_to_file(Name,FileFlag [,Timeout]) -> ok | {error, open_file}

sys:install(Name,{Func,FuncState} [,Timeout]) -> ok
sys:remove(Name,Func [,Timeout])

sys:statistics(Name,Flag [,Timeout]) -> ok | {ok, Statistics}.

sys:get_status(Name [,Timeout]) -> {status, Pid, {module, Mod}, Status}

sys:get_state(Name [,Timeout]) -> State

sys:replace_state(Name,ReplaceFun [,Timeout]) -> State

sys:suspend(Name [,Timeout]) -> ok
sys:resume(Name [,Timeout]) -> ok
```

To print our trace events to the shell, use `trace/2`. When logging the events for later retrieval, use `log/2`. Turn on and off the logging by setting the `LogFlag` to `true` or `false`. By default, the last 10 events are stored, a value you can override by turning on the logging using `{true, Int}`, where `Int` is a non negative integer.

Events can be retrieved using the `print` and `get` flags. When using `log_to_file/2`, events are stored in textual format. The `FileFlag` is a string denoting the absolute or relative filename or the atom `false` to turn it off. Use `sys:install/2` to write your own triggers and trace functions in conjunction with system events and `sys:remove/2` to recall them.

Turn the gathering of the `statistics/2` on and off by setting the `Flag` to `true` or `false` respectively. Use `get_state/1` to examine loop data and `replace_state/2` to replace it. And finally, `get_status/1` returns all the available data relative to the internal behaviour state. The `get_state/1`, `replace_state/2`, and `get_status/1` functions are incredibly helpful when debugging and troubleshooting live systems.

You can enable tracing, logging, and statistics when starting your behaviour by using the `Opts` field. If you pass `[{debug, DbgList}]`, where `DbgList` contains one or more

of the entries `trace`, `log`, `statistics`, and `{log_to_file, FileName}`, these flags are enabled as soon as the behaviour is created.

Performance Tuning Options

If benchmarks and profiling show you are having problems with execution time, there are some useful performance tuning options that might help. They are the same options taken by the `spawn_opt/4` BIF, but passed as an argument of the format `[{spawn_opts, OptsList}]` in the behaviour `Opts` field. They help with performance tuning and controlling the memory usage of your behaviours and can be included together with debug and timeout options.



The only way to be sure you have performance issues and bottlenecks related to memory management is by profiling and benchmarking your systems. Start fine-tuning only if you have benchmarks to guide you, and retain modified settings only if you are seeing benefits. Optimising memory management often has the opposite effect and makes your programs slower. The vast majority of cases do not call for performance tuning.

Memory Usage

The Erlang runtime system uses a *generational garbage collector*. This means that data in the heap which survives its first garbage collection is copied to an area referred to as the *old heap*. Data is collected in the old heap, because having survived one garbage collection, it will most likely survive future ones. We call this a generational garbage collection. The garbage collector always starts by freeing data on the heap first. If it has been unable to free enough memory, it will garbage collect the old heap as well, stopping as soon as it has reclaimed what is needed. After a predefined number of generational garbage collections, a *fullsweep garbage collection* is triggered. This will inspect and free all data no longer referenced, both in the heap and the old heap. There might be cases where, because of periods of little activity and a large allocated heap, long-lived processes might be holding onto data which is no longer needed. This data will be freed only when a full sweep happens.

If you suspect that the performance issues are related to memory management, benchmark your system while manipulating the heap and the garbage collector settings. In the following example, we start the frequency server and trace events related to the garbage collector. We notice that when allocating five frequencies, we spend ten microseconds (447804 - 447794) garbage collecting:

```
1> dbg:tracer().
{ok,<0.33.0>}
2> {ok, Pid} = frequency:start().
```

```

{ok,<0.36.0>}
3> dbg:p(Pid, [garbage_collection, timestamp]).
{ok,[{matched,nonode@nohost,1}]}
4> frequency:allocate(), frequency:allocate(), frequency:allocate(),
   frequency:allocate(), frequency:allocate().
{ok,14}
(<0.36.0>) gc_start [{old_heap_block_size,0},
 {heap_block_size,233},
 {mbuf_size,0},
 {recent_size,0},
 {stack_size,11},
 {old_heap_size,0},
 {heap_size,215},
 {bin_vheap_size,0},
 {bin_vheap_block_size,46368},
 {bin_old_vheap_size,0},
 {bin_old_vheap_block_size,46368}] (Timestamp: {1341,351874,447794})
(<0.36.0>) gc_end [{old_heap_block_size,0},
 {heap_block_size,233},
 {mbuf_size,0},
 {recent_size,45},
 {stack_size,11},
 {old_heap_size,0},
 {heap_size,45},
 {bin_vheap_size,0},
 {bin_vheap_block_size,46368},
 {bin_old_vheap_size,0},
 {bin_old_vheap_block_size,46368}] (Timestamp: {1341,351874,447804})

```

If we now spawn the frequency server, setting the minimum heap size to 1024 words (a smaller size would have been enough), we have enough memory to allocate the frequencies without triggering the garbage collector:

```

1> dbg:tracer().
{ok,<0.33.0>}
2> {ok, Pid} = gen_server:start_link({local, frequency}, frequency, [],
   [{spawn_opt, [{min_heap_size, 1024}]}]).
{ok,<0.36.0>}
3> dbg:p(Pid, [garbage_collection, timestamp]).
{ok,[{matched,nonode@nohost,1}]}
4> frequency:allocate(), frequency:allocate(), frequency:allocate(),
   frequency:allocate(), frequency:allocate().
{ok,14}

```

Memory-related options that can be passed when starting a behaviour include

`min_heap_size`

It sets the size the process heap will grow to before the garbage collector is triggered. There is something fishy about this name, is it is in fact the maximum the heap is allowed to grow to before triggering the gc.

`min_bin_vheap_size`

Sets the initial and minimal value of the space this process is allowed to use in the shared binary heap before triggering a garbage collection on the binaries.

`fullsweep_after`

Determines the number of generational garbage collections that have to be executed before a complete garbage collection pass.

Process heap

By increasing the `{min_heap_size, Size}` to an appropriate value in a short-lived process, you can allow the process to execute without having to allocate more memory to further increase the heap size. It will also stop the garbage collector from running when more memory is needed. Upon termination, all the memory is released at once without the need to trigger the garbage collector. The minimum heap size value has to be calculated and derived through proper benchmarks. Trial and error will not give you enough information to choose the right size. Picking too large a size might slow down your program.

`Size` is measured in words, a unit size of data used by a particular processor architecture. In a 32 bit architecture, a word is 4 bytes (32 bits) and in a 64 bit architecture, 8 bytes (64 bits). You could set the minimum heap size for all processes using the `+hms` flag when you start the Erlang runtime system using `erl`. Using the `+hms` flag is advisable only if you have relatively few processes running in your system and, of course, only if benchmarks show an increase in performance. As a rule of thumb, it is always better to set the minimum heap size on a per process basis, and only if benchmarks show benefits. Heap sizes increases are based on the Fibonacci series, so the minimum heap size set will be the next value in the Fibonacci sequence larger than or equal to `Size`.

Virtual Binary Heap

Binaries larger than 64 bytes are stored in a shared binary heap used by all processes. Binaries smaller than 64 bytes are stored in the respective process heaps. Binaries over 64 bytes are accessed by a reference which, through message passing, can be shared among processes. Using a reference making message passing of large binaries efficient, because they do not have to be copied. A reference counter increments for every reference pointing to the binary, and decrements when the reference is removed. When this counter reaches 0, the binary can be garbage collected. The garbage collection is triggered when the virtual binary heap size is exceeded. The virtual binary heap is local to every process, and is not shared globally.

A spawn option, related to garbage collection and useful for performance tuning, is the minimum binary virtual heap size, configured using the `{min_bin_vheap_size, VSize}` value. The virtual binary heap size is the space a process is allowed to use before triggering the garbage collector and freeing the space taken up by binaries which are no

longer referenced. This size refers to binaries larger than 64 bytes in size. These are accessed through binary references, which can be used by all processes. Binaries smaller than 64 bytes are stored on the normal heap and are copied when sent as a part of a message to other processes or during garbage collection. As with the heap size, pick your VSize after having executed proper benchmarks. You can set the virtual binary heap size for all processes using the `+hmbs` flag when you start your system with `erl`, but just like the regular heap, use this option with restraint, and preferably only on specific processes, not on all of them.

Full sweep of the heap

By setting the `{fullsweep_after, Number}` spawn option, you can specify the Number of generational garbage collections that take place before executing a fullsweep. Setting Number to 0 will disable the garbage collection mechanism, and free all unused data in both the heap and the old heap every time it is triggered (NOTE: Documentation states *Setting the number to zero effectively disables the general collection algorithm, meaning that all live data is copied at every garbage collection.* Do they mean from heap to old heap?). This will help in environments with little RAM where memory has to be strictly managed, or in cases where we want to immediately garbage collect (potentially large) binaries which are no longer referenced. Setting a small value will be suitable if your data is short-lived and benchmarks demonstrate is cluttering up your heap. The Erlang documentation suggests 10 or 20, but you should pick your own as a result of your benchmarks. The default value is much larger!

A fullsweep garbage collection is triggered every time you hibernate your process. This might help reduce the memory footprint when working with processes who have memory intensive computations but little overall activity. You can set the fullsweep value globally for all processes using the `erlang:system_flag/2` call, but we recommend you don't. You can use the `process_info/2` BIF to get information on the settings you change:

```
5> process_info(Pid, garbage_collection).
{garbage_collection,[{min_bin_vheap_size,46368}, {min_heap_size,1597},
 {fullsweep_after,65535}, {minor_gcs,0}]}]
```

Note the default value settings of the `fullsweep_after` option. Note how we set the minimum heap size to 1024, but it appears to be 1597. We requested 1024 words, but 1597 is the first value greater than 1024 (after 987) in the Fibonacci sequence, and is thus picked.



If you start playing with the heap size and garbage collection settings, keep in mind that memory is freed only when the garbage collector is triggered. There might be cases where the process heap contains binary references to potentially large binaries in the shared heap. Each reference to the binary is relatively small, so despite the process not referencing these binaries any more, potentially huge amounts of memory can be consumed without the garbage collector being triggered, as there is still plenty of space on the process heap. That is why the binary virtual heap is there, calculating the total amount of memory used up by the binaries in the heap. Under these circumstances, hibernating the process or triggering the garbage collection using the `erlang:garbage_collect()` BIF might prove itself more useful. Other issues which can occur include running out of memory. As an example, having a large `min_heap_size` and using the dangerously high default `fullsweep_after` value of 65535 might result in the old heap growing as garbage collections are so far apart, resulting in your system running out of memory before the first full sweeps were triggered. Always stress test your systems, and let soak tests runs span days, if not weeks.

Options to Avoid

Although `monitor` can be passed as an option when using the `spawn_opt/3` BIF, it is disallowed in `gen_servers` and will result in the process terminating with a `badarg`. And although you are allowed to use `link` as an option, starting the behaviours with `start_link` is preferred. Finally, process priorities [should never] be set using the `{priority, Level}` option, where `Level` is the atom `low`, `normal` or `high`. Changing process priorities is even more dangerous than meddling with memory and garbage collection, as it can upset the VM's balance. It can backfire causing the schedulers to behave strangely and unfairly; processes with a higher priority have been known to starve when the ratio between them and those with a lower priority reached certain limits. Furthermore, processes with a lower priority have caused the runtime system to run out of memory when, under heavy load, messages were not consumed as fast as they were produced. You obviously never notice these issues when testing your system, they tend come back and bite you when the live system comes under heavy load. Let the runtime system decide on your behalf, especially when dealing with hundreds of thousands of processes. You have been warned!

Summing Up

There are many options to control and monitor your behaviours. Alongside your built in tracing and logging functionality, you can during runtime and without the need to recompile your code dynamically add generic trace and debug triggers or change your

process state using funs and the sys module. This is a priceless feature, as you can use it on systems you have never seen which have been running for years on end without the need for them to be restarted.

Optimizing processes through the use of the memory flags in their options is trickier, as it requires you to stress test your system and base your optimisations on the information you extract as a result of your benchmarks. It is rare you will have to manipulate the default garbage collector settings or play with your heap sizes. But if and when you are having performance problems, you will be grateful you have read this far in this chapter.

What's Next?

We will now park performance tuning until we reach [Chapter to Come]. In the next chapters, we will focus on the remaining behaviours, starting with finite state machines, followed by event managers, supervisors and applications. Remember that they are all built on the same foundations, so the sys module and all of the options we have discussed in this chapter will be valid.

CHAPTER 6

Finite State Machines

Now that we've become experts at writing generic servers, the time has come to master our next behaviour. When prototyping systems with what eventually became Erlang, its inventors Joe Armstrong, Mike Williams and Robert Virding were implementing a soft telephony switch allowing them to phone each other and say hello.¹ Each phone accessing the switch was prototyped as a process acting as a finite state machine (FSM). At any one time, the function would represent the state the phone was in (on hook, off hook, dialing, ringing, etc) and receive events associated with that state (incoming call, dial, off hook, on hook, etc).

One of the outcomes of this prototyping activity was to ensure that Erlang became a language suited for and optimized for building nontrivial and scalable finite state machines, a key component in many complex systems. Developers use FSMs to program protocols stacks, connectors, proxies, workflow systems, and simulations, to mention but a few examples. So it was no surprise that when OTP behaviours came along, they included generic finite state machines.

In this chapter, we introduce finite state machines implemented in pure Erlang. We break an example up into generic and specific code, migrating it to the `gen_fsm` behaviour. The good news is that all the borderline cases relating to concurrency and error handling that apply to generic servers also apply to finite state machines. So while we might mention some of them, there will be no need for us to go into the same level of detail. After all, a finite state machine implementation is essentially a special variant of a generic server.

1. Movie fans will have seen this switch in the blockbuster production of *Erlang The Movie*. It was filmed when the language was still evolving, so observant fans will have noticed the old syntax in some of the examples. If you have not viewed it, look for it on YouTube. It is a must see!

Finite State Machines the Erlang Way

Before diving into our examples, let's get a bit of theory out of the way. A *finite state machine* is an abstract model consisting of a finite number of states and incoming events. When the program is in each state, it can receive certain events from the environment—and only those events. When an event arrives and the finite state machine is in a certain state, the program executes some pre-determined actions associated with that state and transitions to a new state. In it, it waits for a new event.

For instance, in the FSM shown in [Figure 6-1](#), a program in state 1 can handle events 1 and 2. Event 1 keeps the FSM in its current state while event 2 causes a transition to state 2. In state 2, event 3 causes a transition back to state 1. Any other events coming out of sequence (such as event 1 or 2 when in state 2) are handled only after a transition to a state where they can be dealt with.

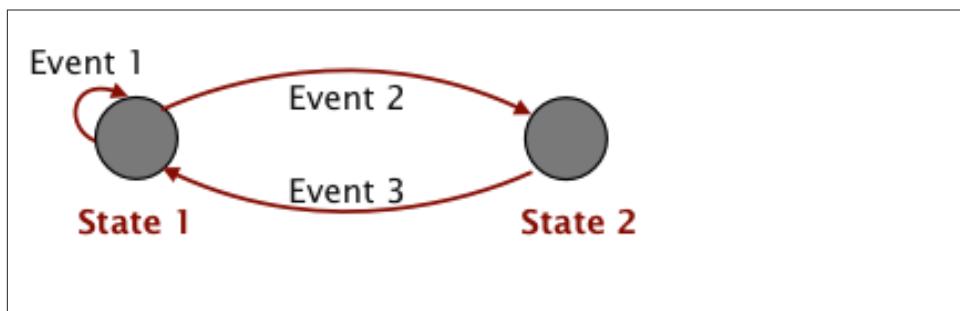


Figure 6-1. Finite State Machines

Picture an FSM consisting of two states, **day** and **night**, depicted in [Figure 6-2](#). If we are in state **day** and the **sunset** event occurs, the FSM transitions to state **night**. At **sunrise**, it transitions back into state **day**.

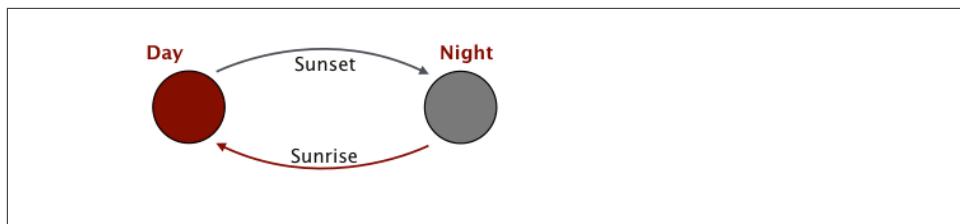


Figure 6-2. Erlang FSMs

In Erlang, each state is represented within a tail-recursive function and events are represented as messages. So for [Figure 6-2](#), the code for state **day** would look as follows:

```
day() ->
receive
    sunset -> night()
end.
```

Upon receiving an incoming event, the FSM executes one or more actions before transitioning to its next state. The state transition is achieved by calling the next function, determined by the combination of the current state and inbound event. In the following example, the combination of the event **sunrise** in state **night** will result in the action defined in the function `make_roosters_crow/0`, followed by a transition to state **day**.

```
night() ->
receive
    sunrise ->
        make_roosters_crow(),
        day()
end.
```

When you start a finite state machine, you need to give it a starting state and initialize it. As in our code example, we could initialize the FSM by spawning the `init/0` function and create the earth there² before moving on to state **day**.

```
start() ->
spawn(?MODULE, init, []).

init() ->
create_earth(),
day().
```

This is how we do FSMs in Erlang. The keys to keeping FSMs simple are selective receives, tail recursive functions, and the ability to initialize the FSM when spawning the process.

You should completely design your FSM, perhaps by drawing out a diagram like the ones in this chapter, before you start coding. You want to know what your states, events, actions and state transitions are. If they get complex, see whether your FSM can be split up into smaller FSMs which, during execution, pass the flow between each other. They will be easier to both implement and maintain.

Beware of the common beginner error where instead of using a generic FSM, you use a generic server and unknowingly store the FSM state in the loop data. Ask yourself when designing the system whether you need a FSM or a client-server behaviour. The answer is usually obvious if you consider the question in the design phase of the project.

2. This would be an interesting function to benchmark.

Coffee Finite State Machine

To keep our Java fans happy, let's use a coffee vending machine as an FSM example. The implementation we are about to study will have three states:

- **select**, allowing the customer to select the desired coffee brew
- **payment**, allowing the customer to insert coins and pay for the selected item
- **remove**, a state where the FSM waits for the user to remove the drink from the machine

These states are linked by four events that trigger actions and transitions to next states. Events triggered by the customer include:

- Making a coffee **selection**
- Dropping a coin of any value in the slot to **pay** for the selection
- Pressing the **cancel** button
- A successful **remove** of the cup of coffee from the machine

Note that most of these events can be triggered in most states. If the FSM is in the payment state, there is nothing stopping a user from pressing the coffee selection buttons, or if we are in state selection, the user can always insert a coin. If the events can be triggered, they have to be managed regardless of the state. When events are received in a particular state, actions can be executed before transitioning to the next state. The ones in our example include:

- **Display** text in the coffee machine LED display.
- **Return change** or inserted coins to the client.
- **Drop** the **cup** in the machine.
- **Prepare** the selected drink.
- **Reboot** the coffee machine (not user-initiated).

A simplified version of the FSM can be seen in [Figure 6-3](#). Note that it does not depict a complete set of events and actions. Coins can be inserted in states other than payment, the cancel button can be pressed in the selection or remove states, or the hardware could be reset when starting the FSM. The figure does, however, provide an overview of all the state transitions and events that trigger them.

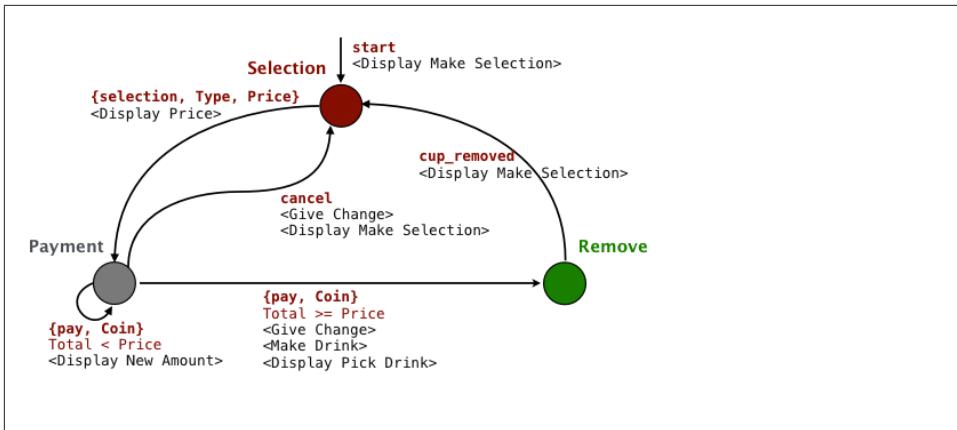


Figure 6-3. Coffee machine FSM

With this model in mind, let's start by stepping through a pure Erlang implementation of the FSM. After that, we will migrate the implementation to the generic finite state machine behaviour module.

The Hardware Stub

Embedded systems that require sensors and hardware interactions include device drivers written in C interfacing to the Erlang code. To keep the example simple, we have stubbed this interaction in the *hw.erl* module. We use this module in both the Erlang implementation and the generic FSM behaviour implementation.

```

-module(hw).
-compile(export_all).

display(Str, Arg)      -> io:format("Display:" ++ Str ++ "~n", Arg).
return_change(Payment) -> io:format("Machine:Returned ~w in change~n",[Payment]).
drop_cup()              -> io:format("Machine:Dropped Cup.~n").
prepare(Type)           -> io:format("Machine:Preparing ~p.~n",[Type]).
reboot()                -> io:format("Machine:Rebooted Hardware~n").

```

You will see calls to this module in the FSM implementations. Functions being called as a result of the sensors in the coffee machine will call the client functions in *coffee.erl* module directly. For testing purposes, we instead call them from the shell. With this out of the way, let's start looking at the implementation itself.

The Erlang Coffee Machine

In this section we create the Erlang part of the application, keeping in mind throughout how the FSM in this example can be generalized and made into a reusable behaviour in OTP.

Starting

We start the FSM using the `start_link/0` function. It spawns a new process that starts executing in the `init/0` function and registers itself using the name `coffee`, the same name as the module. It reboots the machine and shows *Make Your Selection* in the display. We then enter into our first state by calling the tail recursive function `selection/0`. Have a look at it and try to split it up into generic and specific code.

```
-module(coffee).
-export([tea/0, espresso/0, americano/0, cappuccino/0, pay/1, cup_removed/0, cancel/0]).
-export([start_link/0, init/0]).

start_link() ->
    {ok, spawn_link(?MODULE, init, [])}.

init() ->
    register(?MODULE, self()),
    hw:reboot(),
    hw:display("Make Your Selection", []),
    selection().
```

The generic code, highlighted in the above example, includes spawning the process that runs in the `init/0` function, registering it and transitioning to the first state. The code specific to the coffee machine is the process name, the callback module, and the hardware specific operations executed in `init/0`, alongside any arguments we pass on to that call. The first state is also specific, as is any loop data we might pass on to that state. In our example, there is no state needed at startup.

The events

Two sets of client functions generate events, which are passed on to the coffee FSM as asynchronous calls. The first four inform the FSM of the drink selection the user made, together with the price. The `cup_removed` event is triggered by hardware sensors when a cup is removed. If a coin is inserted, `pay/1` is called, with the value of the coin passed as an argument. Finally, `cancel` is called when the cancel button is pressed.

As we mentioned earlier, these events can be triggered in any state. There is nothing stopping a user from pressing the cancel button when the drink is being prepared, or inserting a coin without having made a selection.

```
%% Client Functions for Drink Selections

tea()      -> ?MODULE ! {selection, tea,      100}.
espresso()  -> ?MODULE ! {selection, espresso, 150}.
americano() -> ?MODULE ! {selection, americano, 100}.
cappuccino() -> ?MODULE ! {selection, cappuccino, 150}.

%% Client Functions for Actions

cup_removed() -> ?MODULE ! cup_removed.
```

```

pay(Coin)      -> ?MODULE ! {pay, Coin}.
cancel()       -> ?MODULE ! cancel.

```

In these client functions, the tags and any data (such as the price) associated with the events are specific. What is generic are the sending of the events to the FSM and the possibility of having synchronous and asynchronous calls. In our example, the calls are all asynchronous. Had some of them been synchronous, the return value would also have been specific, but the protocol and receive statement receiving the reply would be generic.

The selection state

In the `init/0` function, after having initialised the coffee machine, we make the transition to our first state. This is the selection state, where the customer picks their drink. Upon receiving the event `{selection, Type, Price}`, we display the price of the drink and move to the next state, payment. In this state, we pass the arguments `Type`, `Price` and amount `Paid`, initially set to 0. These three arguments are the loop data needed in the payment state.

If a customer inserts a coin without having made a selection, we have to return it. If they press the cancel button, we need to remove the event from the process mailbox, ensuring that it is not accidentally received in a later state.

```

%% State: drink selection

selection() ->
  receive  {selection, Type, Price} ->      hw:display("Please pay:~w",[Price]),
  end.                                              payment(Type)

```

Every combination of state and event will result in a specific set of actions and a transition to the next state. The generic code consists of the sections receiving events, state transitions, and storing the loop data. The specific code relates to handling the events, namely updating the display, returning the coins and deciding on the next state.

The payment state

When the customer has picked their drink, it is time to either pay for it or cancel the selection. Every coin inserted will result in the event `{pay, Coin}` being generated, where `Coin` is the amount that has been inserted. This amount is added to the total. If the total is greater than or equal to the price of the drink, the code will trigger actions terminating with the transition to the remove state. If not enough money has been inserted, the remaining amount to be paid is updated and the FSM remains in the payment state. If the Cancel button is pressed, any payment made is returned to the user and the FSM returns to the selection state. Any other event (more specifically, pressing any of the selection buttons) is ignored. The way we ignore an event is to re-invoke the current state.

```

%% State: payment

payment(Type, Price, Paid) ->
    receive {pay, Coin} -> if Coin + Paid >= Price ->
        hw:display("Preparing Drink.", []),
        hw:return_change(Coin + Paid - Price),
        hw:drop_cup(), hw:prepare(Type),
        hw:display("Remove Drink.", []),
        remove(); true ->
        ToPay = Price - (Coin + Paid),
        hw:display("Please pay:~w", [ToPay]),
        payment(Type, Price, Coin + Paid) end; cancel -> hw:display("Make Your Selection")
    _Other -> %selection payment(Type, Price, Paid)
end.

```

As in the selection state, the generic code includes receiving events, state transitions and storing the loop data. Specific code includes the events themselves, the actions executed as a result, and the next state. Storing the loop data could have been done in one variable containing a record, but as different states need a different number of arguments, this solution is cleaner for this particular example.

The remove state

The FSM enters state *remove* when the coffee is paid for and has been brewed. It is a state of its own because the machine cannot be used to brew other beverages until the user removes the cup. When that happens, sensors will trigger the *cup_removed* event and reset the display. This allows us to transit to the selection state, where the activity can start all over again. There is nothing stopping the customer from inserting coins, so if they do, they have to be returned. The same applies to them pressing the cancel or selection buttons, events that have to be ignored.

```

%% State: remove cup

remove() ->
    receive cup_removed -> hw:display("Make Your Selection", []), selection();
end.

```

Before starting the next section about the FSM behaviour, download the code and stub modules and try it out. When doing so, take a moment to think of other possible implementations of an Erlang-based FSM. What parts of them are specific and what parts are generic? Of the generic parts, how would you package the generics into a callback-based library module?

Generic Finite State Machines

To separate the generic from the specific functionality in a finite state machine, we'll take the same course we took with generic servers. **Table 6-1** lists the major generic and specific parts of the FSM.

Table 6-1. Client-Server Generic and Specific Code

Generic	Specific
<ul style="list-style-type: none">• Spawning the FSM• Storing the loop data• Sending events to the FSM• Sending synchronous requests• Receiving replies• Timeouts• Stopping the FSM	<ul style="list-style-type: none">• Initialising the FSM state• The loop data• The events• Handling events/requests• The FSM states• State transitions• Cleaning up

Spawning the FSM, ensuring it has started correctly, and registering it does not change from one implementation to another. What does change is the local or global registered name of the process (if registered at all), debugging options, and arguments needed for the initialization. Initializing the FSM is specific, including determining the initial state and binding the loop data. Both are returned to the generic FSM receive/evaluate loop, which generically stores the data and state.

Sending both synchronous and asynchronous events and requests to the FSM is generic, as is receiving replies. What is specific are the contents of the events and requests and how they are handled based on the the FSM state.

The states are all specific, as are the actions that have to be executed, choosing the next state to which to transition the FSM, and updating loop data. Handling of timeouts, both within the client and the FSM itself, is generic. What happens when the timeout is triggered, on the other hand, is specific. Finally, stopping the FSM is generic, whilst cleaning up prior to termination is specific.

We can view the FSM as an extension to the generic server, with state handling added on top. Messages become events and callback functions that receive the messages become states. All of the generic code is placed in a library module called `gen_fsm`, while all of the specifics are placed in a callback module. The architecture is illustrated in **Figure 6-4**, which you can compare to **Figure 4-1**.

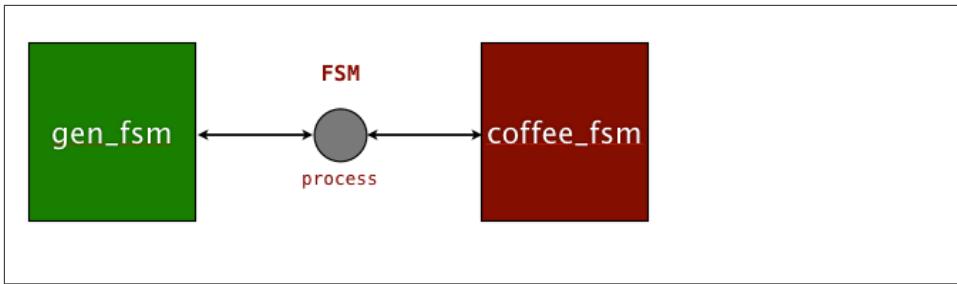


Figure 6-4. The *FSM* callback module

A Behaviour Example

Using the coffee machine example, let's have a look at all the library APIs and associated callback functions of the *gen_fsm* behaviour module. We explore starting and stopping the generic FSM, as well as synchronous and asynchronous events. When stepping through the code, compare the *gen_fsm* behaviour with that of *gen_server*. If you want to take it for a practice run, download the code from the book website.

Starting the FSM

Every behaviour callback module starts with module, version, behaviour, and export directives. It also contains all of the state callback functions. While not mandatory, it is good practice to also include all of the client functions that generate the events in one place.

```
-module(coffee_fsm).
-behaviour(gen_fsm).

-export([start_link/0, stop/0]).
-export([init/1, terminate/3]).
-export([americano/0, cappuccino/0, ...]). %% Client Functions for events
```

The behaviour directive specifies the atom *gen_fsm*, used for compile-time warnings if callback functions are not exported. Exported functions include the start and stop functions with their respective callbacks, the client functions and state callback functions. The *version* directive contains a valid term, usually an atom, integer, or real number denoting the module version used during software upgrade.

The coffee machine is started using the *gen_fsm:start_link/4* call, which spawns the FSM and links it to the parent. It returns the tuple {ok, Pid}, where Pid identifies the spawned process, or {error, Reason} if something goes wrong. We will cover possible error reasons in a moment. For now, let's focus on the example.

As with all OTP behaviours, we tend to wrap the *start_link/4* call in a client function, located in the callback module. In our example, we've called it *coffee_fsm:start_link/*

0, but it could take on any name you like. What is important is that it eventually calls `gen_fsm:start_link` and returns whatever this call returns, most commonly `{ok, Pid}` or `{error, Reason}` as shown in (Figure 6-5), or the atom `ignore`. These values become relevant when we look at supervisors in Chapter 8

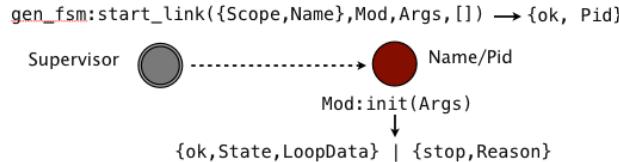


Figure 6-5. Starting a `gen_fsm`

As soon as the generic FSM process has been spawned, the `init/1` function in the callback module is invoked. Just as with generic servers, this function contains all the specific initialization code. In our example, it would reboot the hardware, reset the display, and return a tuple of the format `{ok, StartState, LoopData}`, where `Start State` denotes the state the FSM will be in when it receives its first event. `LoopData` contains the data passed to the state callback functions. We are also trapping exits in this example, for reasons that will become obvious when we look at termination.

```

start_link() ->
    gen_fsm:start_link({local, ?MODULE}, ?MODULE, [], []).

init([]) ->
    hw:reboot(),
    hw:display("Make Your Selection", []),
    process_flag(trap_exit, true),
    {ok, selection, []}.

```

In our example, the `StartState` is `selection` and the `LoopData` is not used, so we simply return the empty list value `[]`. When `init/1` callback returns control to the generic module, the synchronous `gen_fsm:start_link` call returns.

We register the process locally and set the callback module using the `?MODULE` macro, which at compile time is replaced with the atom `coffee_fsm`. We pass `[]` as an argument to the `init/1` callback function and set no options.

The following functions, identical to the ones exported by the generic server module, start a finite state machine:

```

gen_fsm:start_link(NameScope,Mod,Args,Opts)
gen_fsm:start(NameScope,Mod,Args,Opts)
gen_fsm:start_link(Mod, Args, Opts)
gen_fsm:start(Mod, Args, Opts) -> {ok, Pid}

```

```

{error, Error}
ignore

Mod:init/1 -> {ok, NextState, LoopData}
{stop, Reason}
ignore

```

NameScope defines how we register our behaviour. Just like with gen_servers and can be set to {local, Name}, {global, Name}, or {via, Module, ViaName}, where the via tuple points to a user-defined process registry exporting the same API as the global module, all previously covered in “[Going Global](#)” on page 91. We can use the start functions to avoid linking the FSM process to its parent, and we can also decide not to register it. Options covered in [Chapter 5](#) can also be passed. They include timeouts, debug, and spawn options.

If something goes wrong in the init/1 callback, you can either terminate *abnormally* or return the tuple {stop, Reason}. It will propagate the error to the parent process calling the gen_fsm start function (typically via one of the callback module’s start functions), causing it to terminate as well. If the parent process happens to be a supervisor, it will in turn terminate all of its children and abort the start-up procedure. Although things can go wrong when the system is running, by default, the system cannot recover from a fault in the init callback function.

The most common failure reason you will encounter when testing your FSM from the shell is {error, {already_started, Pid}}. It occurs if another process with the same registered name already exists:

```

1> coffee_fsm:start_link().
Machine:Rebooted Hardware
Display:Make Your Selection
{ok,<0.38.0>}
2> coffee_fsm:start_link().
{error,{already_started,<0.38.0>}}

```

If want to let the supervisor continue to start workers when init/1 fails, return the atom ignore. Instead of aborting the start-up procedure, the supervisor will store the child specification and continue starting other behaviours. We cover the ignore and stop options in more detail in chapter [Chapter 8](#) when we look at supervisors.

Until then, the following example should however give you an overview of the different behaviours. Pay particular attention to what causes the process calling the start and start_link functions to terminate. We’ve omitted the module headers from this example. If you want to view them, download the *fsm_test.erl* module from the book code repository.

```

start_link(TimerMs, Options) ->
    gen_fsm:start_link(?MODULE, TimerMs, Options).
start(TimerMs, Options) ->
    gen_fsm:start(?MODULE, TimerMs, Options).

```

```

init(0) ->
  {stop, stopped};
init(1) ->
  {next_state, selection, []};
init(TimerMs) ->
  timer:sleep(TimerMs),
  ignore.

```

Let's run the code. In the first set of tests, we stop the FSM by returning {stop, Reason}.

```

1> test_fsm:start_link(0, []).
** exception exit: stopped
2> test_fsm:start(0, []).
{error,stopped}

```

Note the difference when the shell is linked to the behaviour and when it is not.

In the shell prompts 3 and 4, we the FSM is initialized with the `test_fsm:init(1)` call, which accidentally specifies `next_state` instead of `ok` as the first element of the return tuple in the callback function. This results in an invalid return value not recognised by the FSM backend module, a mistake one of the authors has made many times.

```

3> test_fsm:start_link(1, []).
** exception exit: {bad_return_value,{next_state,selection,[]}}
4> test_fsm:start(1, []).
{error,{bad_return_value,{next_state,selection,[]}}}

```

A behaviour module will terminate with the reason `bad_return_value` whenever you return a control tuple that does not follow the predefined format.

When reading through this example, make sure you understand the effect of the EXIT signal propagation when the shell process is linked to the finite state machine and when it is not. In shell prompt 5, we pass a 1000 millisecond argument to `init/1` to cause it to sleep for that long, but set the `timeout` option to 100 milliseconds; this triggers a timeout in the startup process that results in the `{error, timeout}` tuple. This will be returned whether or not the process is linked to the shell process.

```

5> test_fsm:start_link(1000, [{timeout, 100}]). 
{error,timeout}

```

In our last set of tests, at shell prompts 6 and 7, our `init/1` function returns `ignore`. This does not result in the behaviour terminating abnormally, and as a result, does not propagate further.

```

6> test_fsm:start_link(2, []).
ignore
7> test_fsm:start(2, []).
ignore

```

Although these examples specific use the `gen_fsm` behaviour, they are valid for all OTP workers.

Enough on starting and initializing our FSMs. Let's move on to important things in life and figure out how to get this coffee brewed.

Sending Events

Having started our coffee FSM, we need to be able to define the states and send both *synchronous* and *asynchronous* events. When handled, they trigger state transitions. Events are usually sent in client functions defined in the callback module. Let's start looking at asynchronous events in our FSM and see how they are handled in the different states.

Asynchronous events

Asynchronous events are sent using the `gen_fsm:send_event(Name, Event)` library function. This sends the Event to the FSM, which handles it in the callback function `State(Event, LoopData)` in the callback module. After handling the request, the `State/2` function returns the new loop data with the `next_state` or the `stop` reason (Figure 6-6).

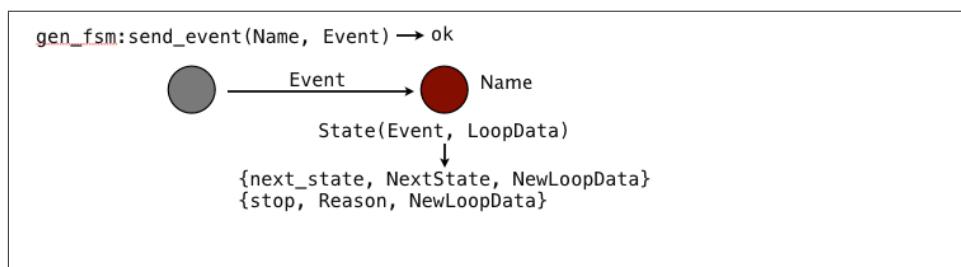


Figure 6-6. Sending events

Our FSM event functions are split into two categories. The first are customer drink selections. These send events of the format `{selection, Type, Price}`, where `Type` is one of the atoms `tea`, `espresso`, `americano`,³ or `cappuccino`. `Price` is either 100 or 150 units.

```
tea()      -> gen_fsm:send_event(?MODULE,{selection,tea,100}).  
espresso() -> gen_fsm:send_event(?MODULE,{selection,espresso,100}).  
americano() -> gen_fsm:send_event(?MODULE,{selection,americano,150}).  
cappuccino() -> gen_fsm:send_event(?MODULE,{selection,cappuccino,150}).
```

The second set of events include actions where the user inserts a coin, presses the cancel button or removes a cup. There are no rules stating that events must comprise only static

3. An americano coffee is an espresso topped up with water—it could not be omitted as it is the favourite of one of the authors.

values. Note how in the `pay/1` function we pass a variable as part of our event; the value of the inserted coin is bound to `Coin` and passed through the event `{pay, Coin}`.

```
pay(Coin)      -> gen_fsm:send_event(?MODULE,{pay, Coin}).  
cancel()       -> gen_fsm:send_event(?MODULE, cancel).  
cup_removed() -> gen_fsm:send_event(?MODULE,cup_removed).
```

States in FSMs are defined in callback functions, where the name of the function is the name of the state, Event is the first argument, and the LoopData is the second one. Remember that state callback functions are defined in the callback module and have to be exported. The first state we look at is selection, where the customer is prompted to choose their drink. It was the start state returned by the `init/1` function when we started the FSM.

```
selection({selection,Type,Price}, _LoopData) ->  
    hw:display("Please pay:~w",[Price]),  
    {next_state, payment, {Type, Price, 0}};  
selection({pay, Coin}, LoopData) ->  
    hw:return_change(Coin),  
    {next_state, selection, LoopData};  
selection(_Other, LoopData) ->  
    {next_state, selection, LoopData}.
```

Upon choosing a drink, one of the functions `tea/0`, `espresso/0`, `americano/0` or `capuccino/0` is called. This sends the asynchronous event of the format `{selection, Type, Price}` to the FSM. Regardless of which drink or price the customer chooses, the selection gets handled generically. This event is pattern matched in the first clause of the state callback function, displaying the price the customer has to pay. By returning the tuple `{next_state, NextState, NewLoopData}`, we return the control to the `gen_fsm` module and wait for the next event. In this case, `NextState` is bound to the `payment` state and the `LoopData` to a tuple denoting the selection (`Type`), the price, and the amount paid so far which is initially set to 0. Note how we ignore the incoming loop data, set to the empty list in the `init/1` callback function, but create it for the next state.

What happens if you walk up to a coffee machine when it is in the selection state and insert a coin? In our example, we programmed the FSM to return the coin using the `hw:return_change/1` call, remaining in the selection state and not changing the loop data (which is set to the empty list anyhow). If you prefer to keep the coin, just delete that line of code. Or if you are implementing a deluxe variant of a coffee machine, add functionality to block the coin insert facility until the selection has been entered.

When in the selection state, clients can generate events that do not require any actions or state changes. They include pressing the cancel button or setting off the cup removed sensors, events which need to be handled but can be ignored in the sense that they change neither the current state nor the loop data. Had we not included the third function, a customer pressing the cancel button would have triggered the `selection(cancel)` event.

`cel, []])` call, causing a runtime error, because none of the function clauses would have matched.

Having selected an americano coffee, the FSM displays the amount owed and moves to the state payment, eagerly awaiting the next event.

```
payment({pay, Coin}, {Type, Price, Paid}) when Coin+Paid < Price ->
    NewPaid = Coin + Paid,
    hw:display("Please pay:~w", [Price - NewPaid]),
    {next_state, payment, {Type, Price, NewPaid}};
payment({pay, Coin}, {Type, Price, Paid}) when Coin+Paid >= Price ->
    NewPaid = Coin + Paid,
    hw:display("Preparing Drink.", []),
    hw:return_change(NewPaid - Price),
    hw:drop_cup(), hw:prepare(Type),
    hw:display("Remove Drink.", []),
    {next_state, remove, null};
payment(cancel, {_Type, _Price, Paid}) ->
    hw:display("Make Your Selection", []),
    hw:return_change(Paid),
    {next_state, selection, null};
payment(_Other, LoopData) ->
    {next_state, payment, LoopData}.
```

The customer has to now pay for the coffee. Every time a coin is inserted, the `{pay, Coin}` event is generated. We add the value in `Coin` to the amount `Paid`, and, if the sum is less than the price of the drink, we display the remaining amount to pay. By recursively calling the `payment` function again, we keep the FSM in the payment state, changing the loop data to reflect the amount paid so far.

If the customer has inserted enough change to pay for the drink, we trigger a chain of actions that start by changing the display telling them we are preparing the drink. We return any change and drop the cup. We brew the drink, returning from the synchronous `hw:prepare(Type)` call only when the drink is finished. At this point, we tell the customer to remove their drink and return the control to the `gen_fsm` control loop, indicating that the next state is `remove`.

The customer, while paying for their coffee, could change their mind and press the cancel button. If they do, we change the display to “Make Your Selection”, return any coins they might have paid and indicate that the next state is `selection`. Finally, if a customer triggers the cup removed sensors or presses any of the drink selection buttons, we ignore the event and remain in state `payment`.

Let’s assume the customer has paid for their coffee, they have received their change and the drink has been brewed. The FSM would at this stage be in state `remove`.

```
remove(cup_removed, LoopData) ->
    hw:display("Make Your Selection", []),
    {next_state, selection, LoopData};
```

```

remove({pay, Coin}, LoopData) ->
    hw:return_change(Coin),
    {next_state, remove, LoopData};
remove(_Other, LoopData) ->
    {next_state, remove, LoopData}.

```

Sensors in the coffee machine will be triggered when they remove their cup. This will trigger the `coffee_fsm:cup_removed()` call, resulting in the `cup_removed` event being handled in the first clause. The coffee machine updates its display to “Make Your Selection” and the function returns, setting the next state to *selection*. In the *remove* state, customers can also insert coins, which we return in the second function clause, or they can press the cancel or drink selection buttons, which we ignore in the third clause.

The moment of truth has arrived. Will we get our coffee? Let’s test our program and see if it works. When compiling your behaviour, as we saw in “[Generic Servers](#) on page 71 with generic servers, you get a warning over the missing `code_change/3` callback when compiling the code in this chapter. We will cover it in [Chapter to Come] when looking at software upgrades.

To better understand what is going on, we’ll use the debug options built into OTP and described in “[Tracing and Logging](#) on page 95”. We start the FSM, select tea, change our mind to coffee and insert two 100 unit coins. We get our change, and while waiting to remove the cup, we insert a 50 unit coin just for the sake of testing out the FSM. As we step through the example, you can distinguish the code you input by the prompts (such as `1>`), and debugger print-outs by the `*DBG*` prefix. Output from `io:format/2` in the `hw.erl` module starts with a hint of what parts of the system they represent (`Display:` or `Machine:`), and the rest of the output is actual return values from the function calls.

```

1> {ok, Pid} = coffee_fsm:start_link().
Display:Make Your Selection
{ok,<0.68.0>}
2> sys:trace(Pid, true).
ok
3> coffee_fsm:cancel().
*DBG* coffee_fsm got event cancel in state selection
ok
*DBG* coffee_fsm switched to state selection
4> coffee_fsm:tea().
*DBG* coffee_fsm got event {selection,tea,100} in state selection
ok
Display:Please pay:100
*DBG* coffee_fsm switched to state payment
5> coffee_fsm:cancel().
*DBG* coffee_fsm got event cancel in state payment
ok
Display:Make Your Selection
Machine:Returned 0 in change

```

```

*DBG* coffee_fsm switched to state selection
6> coffee_fsm:americano().
*DBG* coffee_fsm got event {selection,americano,150} in state selection
ok
Display:Please pay:150
*DBG* coffee_fsm switched to state payment
7> coffee_fsm:pay(100).
*DBG* coffee_fsm got event {pay,100} in state payment
ok
Display:Please pay:50
*DBG* coffee_fsm switched to state payment
8> coffee_fsm:pay(100).
*DBG* coffee_fsm got event {pay,100} in state payment
ok
Display:Preparing Drink.
Machine:Returned 50 in change
Machine:Dropped Cup.
Machine:Preparing americano.
Display:Remove Drink.
*DBG* coffee_fsm switched to state remove
9> coffee_fsm:pay(50).
*DBG* coffee_fsm got event {pay,50} in state remove
ok
Machine:Returned 50 in change
*DBG* coffee_fsm switched to state remove
10> coffee_fsm:cup_removed().
*DBG* coffee_fsm got event cup_removed in state remove
ok
Display:Make Your Selection
*DBG* coffee_fsm switched to state selection
11> sys:trace(Pid, false).
ok

```

It seems to work, time for a break!

Timeouts

We are not sure if it has ever happened to you, but imagine you're standing patiently in line to buy your coffee. While doing so, you decide what you want and prepare the exact change and are ready to go. But the person in front of you is apparently not in the same rush. After spending ages reading through all the options, they make their selection and get shown the price. Only then do they dip into their purse or pocket and start looking not for change, but for the exact change. They insert a penny and go back in looking for another one, until they find no more. After which they start looking for nickels and dimes. It can be aggravating, not only for impatient authors. Luckily, we control the coffee machine now, so we can take advantage of that to implement punishment and revenge to discourage this type of behaviour.

Timeouts can be specified within the FSM as an integer in milliseconds or as the atom `infinity`. We can include them in the `init/1` and `State` callback functions. When a

timeout is triggered, the event is sent to the state the FSM is currently in. As we are controlling the code for coffee machine, let's put a bit of stress into the lives of those who do not have any by triggering a timeout if a user spends more than 10 seconds between one coin insertion and another. First, let's refactor the *payment* state by adding a timeout.

```
-define(TIMEOUT, 10000).
...

selection({selection,Type,Price}, _LoopData) ->
...
{next_state, payment, {Type, Price, 0}, ?TIMEOUT};

payment({pay, Coin}, {Type,Price,Paid}) when Coin+Paid >= Price ->
...
{next_state, remove, []};
payment({pay, Coin}, {Type,Price,Paid})
when Coin+Paid < Price ->
...
{next_state, payment, {Type, Price, NewPaid}, ?TIMEOUT};
payment(timeout, {Type, Price, Paid}) ->
hw:display("Make Your Selection", []),
hw:return_change(Paid),
{next_state, selection, []};
payment(_Other, LoopData) ->
{next_state, payment, LoopData, ?TIMEOUT}.
```

Customers inserting coins will now have to hurry. If they take longer than 10 seconds to insert a coin, their selection will be canceled and their money returned. There is a risk that they figure out that by pressing one of the drink selection buttons they will get an extra ten seconds, but let's assume for now that they are too wrapped up looking for their next penny to figure this one out.

In place of a timeout value we can alternatively return `hibernate` if we want to reduce the generic FSM's memory footprint. Use hibernate only of you are not expecting the FSM to receive events for a while, with benchmarks showing you have memory issues. We can also stop the FSM, something we cover later in this chapter.

```
gen_fsm:send_event(NameScope ,Event) -> ok

Mod:State/2 -> {next_state, NextState,NewLoopData}
                  {next_state ,NextState,NewLoopData, Timeout}
                  {next_state, NextState,NewLoopData, hibernate}
                  {stop, Reason, NewLoopData}
```

Asynchronous events to all states

If you want to send an asynchronous event but are not concerned about the state in which it is received, you can use the `send_all_state_event/2` call. This could be useful if you want to execute actions, such as printing the loop data or stopping the FSM, that

are not dependent on a particular state. Events are passed as the first argument to the `handle_event/3` callback function, which executes the actions and then returns the `{next_state, NextState, NewLoopData}` tuple back to the `gen_fsm` control loop (Figure 6-7).

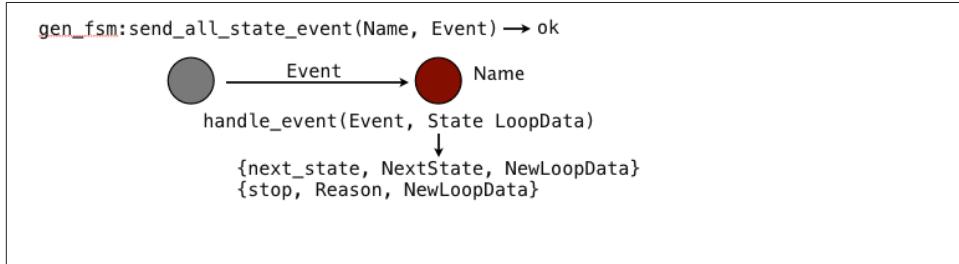


Figure 6-7. Sending events to all states

As with generic servers, the `handle_info/3` callback function takes care of all non OTP-compliant messages such as exit signals, monitors, and messages sent using the `Pid! Message` construct. The `handle_info/3` callback returns the same range of control tuples as `handle_event/3` and `State/2`.

```

gen_fsm:send_all_state_event(NameScope ,Event) -> ok

Mod:handle_info/3,
Mod:handle_event/3 -> {next_state, NextState,NewLoopData}
                         [{next_state ,NextState,NewLoopData, Timeout}]
                         [{next_state, NextState,NewLoopData, hibernate}]
                         [{stop, Reason, NewLoopData}]
  
```

Let's use the `send_all_state_event/2` function to trigger the actions for normal termination of our coffee machine. After all, it doesn't really matter what state it is in, as long as it stops.

Termination

Our coffee machine can terminate for two reasons. It is either stopped *normally* or the process terminates *abnormally* if the exit BIFs are used or a runtime error occurs. For abnormal termination, if the FSM is trapping exits as a result of the `process_flag(trap_exit, true)` call, `terminate/3` (Figure 6-8) is invoked in the callback module. If the FSM is not trapping exits, the FSM terminates and its exit signal propagates to other processes linked to it.

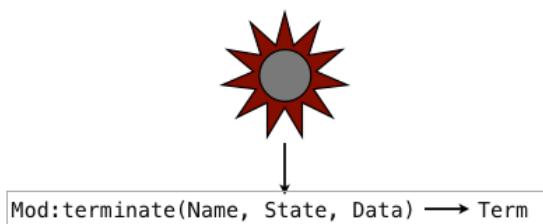


Figure 6-8. Termination

If a stop event is sent using `sync_send_event/2`, the event is handled in `handle_event/3`. Note that unlike the `stop` atom returned in the tuple, the `stop` we pass through `sync_send_event/2` call has no meaning other than one given to it in the program. This also contrasts with the `stop` parameter in `{stop, Reason, LoopData}`, which is interpreted and used by the `gen_fsm` module to terminate the FSM. This is exactly the same principle we discussed when we looked at generic server termination in “Termination” on page 83.

```

stop() -> gen_fsm:sync_send_event(?MODULE, stop).

handle_event(stop, State, LoopData) ->
    {stop, normal, LoopData}.

terminate(_Reason, payment, {_Type,_Price,Paid}) ->
    hw:return_change(Paid);
terminate(_Reason, _StateName, _LoopData) ->
    ok.

```

Note how, in the `terminate` function, we handle the cleanup for the states individually. If a customer has started paying for their drink, they should receive a refund. By doing this in `terminate/3`, we are also able to refund users after an abnormal termination. Here’s an example of what happens.

```

1> {ok, Pid} = coffee_fsm:start_link().
Display:Make Your Selection
{ok,<0.38.0>}
2> coffee_fsm:americano().
Display:Please pay:150
ok
3> coffee_fsm:pay(100).
Display:Please pay:50
ok
4> exit(Pid, crash).
Display:Shutting Down
true

```

```

Machine:Returned 100 in change

=ERROR REPORT==== 3-Mar-2013::12:01:25 ===
** State machine coffee_fsm terminating
** Last message in was {'EXIT',<0.31.0>,crash}
** When State == payment
**     Data == {americano,150,100}
** Reason for termination =
** crash
** exception exit: crash

```

Synchronous events

Although all events sent to our FSM examples were asynchronous, there are cases where we want to ensure the producers can't generate a new event until their previous one is handled. Or you might want to retrieve particular values of the loop data, query the state, or simply synchronize the clients with the FSM. This is when we use the `sync_send_event/2` and `sync_send_all_state_event/2` calls.

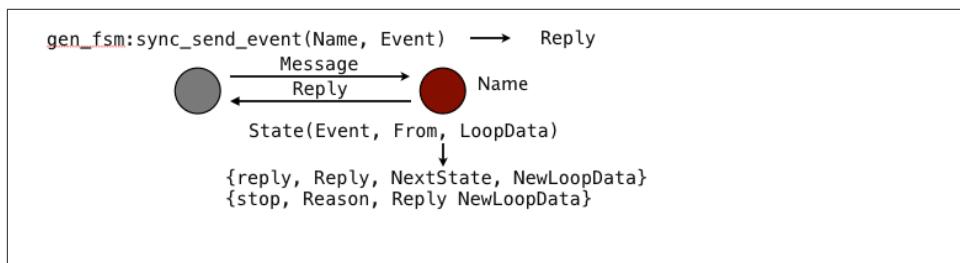


Figure 6-9. Synchronous events

This call and its callback are a middle ground between the `call/2` and `handle_call/3` functions in the generic server, and asynchronous events and event handling in FSMs. Events are handled in the `State(Event, From, LoopData)` callback, where `From` is a tuple denoting the client and the request reference. Instead of returning the `next_state` tuple, the callback returns a tuple of the format `{reply, Reply, NextState, NewLoopData}`. `Reply` is sent back to the client and becomes the return value of the `gen_fsm:sync_send_event/2` call.

Just like with generic servers, we can use the `From` in a `gen_fsm:reply(From, Reply)` call to send the reply, returning `{next_state, NextState, NewLoopData}` in the `State/3` callback function itself.

The `gen_fsm:sync_send_all_state_event/2` function (Figure 6-10) sends synchronous requests to the FMS regardless of its current state. The event is handled in the `handle_sync_event/4` callback function, which returns a `Reply` sent back to the original

caller, either through the use of `From` or in the control tuple sent back to the `gen_fsm` module.

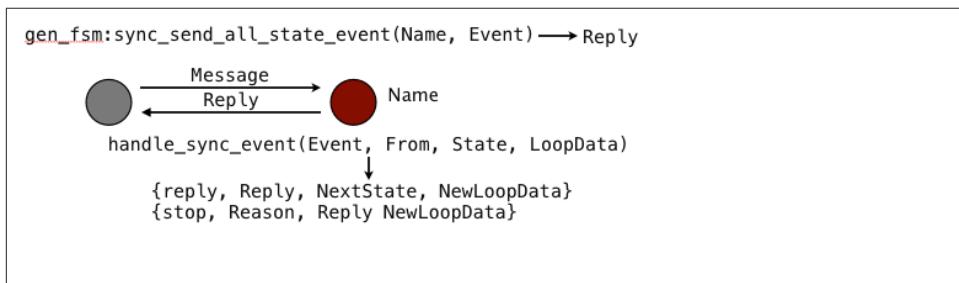


Figure 6-10. Synchronous all state events

```
gen_fsm:sync_send_event(NameScope, Event) -> Reply
gen_fsm:sync_send_event(NameScope, Event, Timeout) -> Reply

gen_fsm:sync_send_all_state_event(NameScope, Event) -> Reply
gen_fsm:sync_send_all_state_event(NameScope, Event, Timeout) -> Reply

Mod:State/3,
Mod:handle_sync_event/4 -> {reply,Reply,NextState,NewLoopData}
                                {reply,Reply,NextState,NewLoopData,Timeout}
                                {reply,Reply,NextState,NewLoopData,hibernate}
                                {next_state,NextState,NewLoopData}
                                {next_state,NextState,NewLoopData,Timeout}
                                {next_state,NextState,NewLoopData,hibernate}
                                {stop,Reason,Reply,NewLoopData}
                                {stop,Reason,NewLoopData}
```



Selective receives are one thing the OTP `gen_fsm` behaviour module does not provide. In complex finite state machines running across unreliable distributed networks, events occasionally arrive out of sequence. Imagine receiving a sunset event when you are in state night! You can either buffer these events in your loop data and handle them when you reach a state which knows how to deal with them, or add an extra state, turning the out of sequence events into a valid one. Both solutions cause unnecessary complexity when compared to the simplicity of using a selective receive, leaving the events in the process mailbox until they are matched in a state in which can actually handle them.

This lack of functionality arises from a conscious design decision in behaviours, where messages are handled in the order they arrive, ensuring no memory leaks occur as a result of any message not being matched. Events in the `gen_fsm` behaviour are handled on a first in, first out basis, and removed from the receiving process' mailbox when read.

There are two approaches if you want to avoid the increase in complexity resulting from messages arriving out of sequence. You could implement your own selective FSM behaviour, which we will explain how to do in [Chapter 10](#). Or you can use a selective FSM behaviour someone else has already implemented. At the time of writing, the most commonly used implementation is `plain_fsm` by Ulf Wiger. It follows all OTP principles and can be included in supervision trees. The `plain_fsm` source code and examples are available on GitHub⁴.

Summing Up

We've now introduced the principles behind the generic finite state machine behaviour. Although it might not be the most commonly used behaviour, when it fits your application it will greatly simplify your task, making your code more readable and easy to maintain. The most important functions we have covered include:

Table 6-2. gen_server callbacks

gen_fsm function or action	gen_fsm callback function
<code>gen_fsm:start/3</code> , <code>gen_fsm:start/4</code> , <code>gen_fsm:start_link/3</code> , <code>gen_fsm:start_link/4</code>	<code>Module:init/1</code>
<code>gen_fsm:send_event/2</code>	<code>Module:State/2</code>
<code>gen_fsm:send_all_state_event/2</code>	<code>Module:handle_event/3</code>
<code>gen_fsm:sync_send_event/2</code> , <code>gen_fsm:sync_send_event/3</code>	<code>Module:StateName/3</code>

4. TODO: Add Link

gen_fsm function or action	gen_fsm callback function
gen_fsm:sync_send_all_state_event/2, gen_fsm:sync_send_all_state_event/3	Module:handle_sync_event/4
Pid ! Msg, monitors, exit messages, messages from ports and socket, node monitors, and other non OTP messages	Module:handle_info/2
Triggered by returning {stop, ...} or when terminating abnormally while trapping exits	Module:terminate/3

Get Your Hands Dirty

Before moving on to the next chapter, why not have a go at implementing a finite state machine to get a feel over the process in designing, coding and testing it? If you are not up to coding, download the code from the book code repository, read through it, and take it for a trial run. What makes this example interesting is that different instances of the behaviours, each representing a cell phone, will speak to each other. They will use the home location register, the database that maps users registered on the network to a unique phone number we implemented in “[ETS: Erlang Term Storage](#)” on page 40.

The Phone Controllers

In our cellular system, there is no central switch. Instead, for every phone attached to the network, we create a phone controller that interact with other controllers. Each controller is a process implemented as a Finite State Machine holding the state of a single phone. All communication between the phone controllers must be asynchronous so as to prevent blocking of the system. Carry out the following API to implement the phone controllers in the `phone_fsm.erl` module:

`start_link(PhoneNumber) -> {ok, FsmPid}.`

Starts a new phone controller FSM process for the phone number linked to the calling process. This should also attach the phone controller process to its phone number in the HLR.

`stop(FsmPid) -> ok.`

Stop a phone controller FSM at `FsmPid`. This should also detach it from its phone number in the HLR.

`connect(FsmPid) -> ok. disconnect(FsmPid) -> ok.`

This function is called by a phone to attach itself to a phone controller FSM process. This must be done so that the phone controller knows where to send the phone replies. This is usually done when a phone is started, when it terminates, or when it is connecting to another FSM process. Note that we connect to an FSM process by its Pid and not its number.

`action(FsmPid, Action) -> ok.`

Sends an action from the phone to the phone controller at `FsmPid`. The legal actions are:

`{outbound,PhoneNumber}`

Try to connect to another phone.

`accept`

Accept a call request.

`reject`

Reject a call request.

`hangup`

Hang up an ongoing call.

The following calls send events between the phone controllers inside the switch.

`busy(FsmPid) -> ok.`

Sends a busy event to `FsmPid`, generally as a reply to an inbound request indicating that this phone is busy and can't accept the call.

`reject(FsmPid) -> ok.`

Send a reject event to `FsmPid`, generally as a reply to an inbound request indicating that we refuse the call.

`accept(FsmPid) -> ok.`

Send an accept event to `FsmPid`, generally as a reply to an inbound request indicating that we accept the call.

`hangup(FsmPid) -> ok.`

Sends a hangup event to `FsmPid` to terminate an ongoing call.

`inbound(FsmPid) -> ok.`

Sends an inbound event to `FsmPid` requesting a call to be set up.

Given this API, [Figure 6-11](#) show what the controller FSM might look like. Note that the FSM is not complete. Before coding, make sure you have reviewed it and added the missing events and state transitions. You'll figure out what they are when reviewing the interfaces.

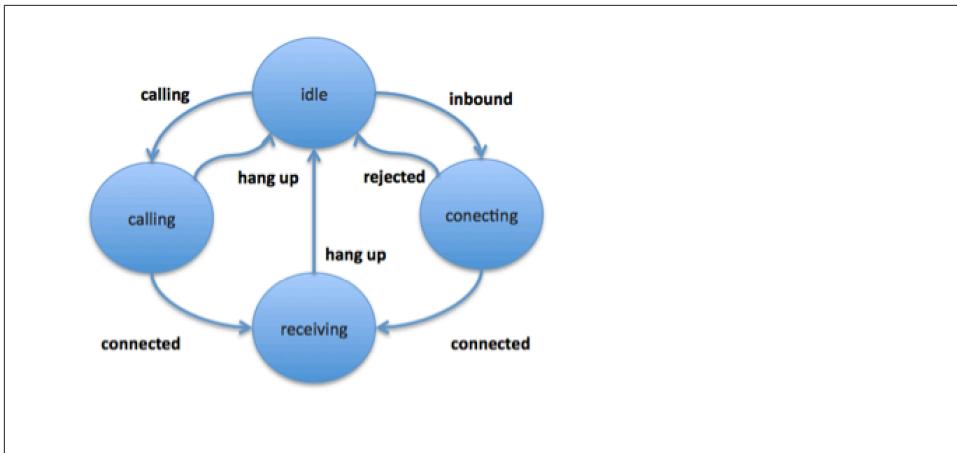


Figure 6-11. Phone Controller FSM

Let's Test It

Each phone controller is connected to a mobile phone. You do not have to write the code for the phone. It is provided in the module *phone.erl* and has the following API:

`start_link(PhoneNumber) -> {ok, PhonePid}.`

Starts a new phone for number *PhoneNumber*, which is linked to the calling process.

`stop(PhonePid) -> ok.`

Stops the phone at *PhonePid*.

`action(PhonePid, Action) -> ok.`

Perms an action requested by the phone user to the phone at *PhonePid*. The legal actions are:

`{call,PhoneNumber}`

Start a call to *PhoneNumber*.

`accept`

Accept a call request.

`reject`

Reject a call request.

`hangup`

Hang up an ongoing call.

Calling an action will result in events being sent to the phone's phone controller using the API defined for the phone controller which we defined in “[The Phone Controllers](#)” on page 133.

```
reply(PhonePid, Reply) -> ok.
```

Send a reply event from the phone controller to the phone. The legal reply events are:

```
{inbound,PhoneNumber}
```

An inbound call has arrived from PhoneNumber.

```
accept
```

An outbound call has been accepted.

```
invalid
```

An outbound call was attempted to an invalid number.

```
reject
```

An outbound call has been rejected.

```
busy
```

An outbound call was attempted to a busy phone.

```
hangup
```

An outbound call has hung up.

These reply events will result in the phone process printing information on the console of the format *PhonePid: PhoneNumber: Event*. For example:

```
<0,459,0>: 103618: hangup
```

You should start your phones in a different node from those running the hlr and the phone controllers. The ultimate test is for a phone to call itself and return a busy signal. Here is a trial test run with three phones:

```
1> hlr:start_link().
{ok,<0.34.0>}
2> phone_fsm:start_link("123").
{ok,<0.36.0>}
3> phone_fsm:start_link("124").
{ok,<0.38.0>}
4> phone_fsm:start_link("125").
{ok,<0.40.0>}
5> {ok,P123}=phone:start_link("123").
{ok,<0.42.0>}
6> {ok,P124}=phone:start_link("124").
{ok,<0.44.0>}
7> {ok,P125}=phone:start_link("125").
{ok,<0.46.0>}
8> phone:action(P123, {call,"124"}).
<0.44.0>: 124: inbound call from 123
ok
9> phone:action(P124, accept).
<0.42.0>: 123: call accepted
ok
10> phone:action(P125, {call,"123"}).
```

```
<0.46.0>: 125: busy
ok
11> phone:action(P125, {call,"124"}).
<0.46.0>: 125: busy
ok
```

What's Next?

In the next chapter, we will look at another worker behaviour, the generic event handler. Before moving on to it, review the manual pages for the `gen_fsm` module. You can find the code implementing the behaviour library in the `gen_fsm.erl` source file. If you previously looked at the `gen_server.erl` code, pay particular attention to how they both interact with the `gen` helper module.

CHAPTER 7

Event Handlers

The mobile frequency server your company produces hits the market and has shown to be extremely popular. Having no visibility of its performance and uptime, you have been asked to implement monitoring software that not only collects statistics and log important things that happen, but also warns you when things go wrong. And that is where the problem begins, because when you are in the office, you want a widget to start flashing on your screen. When you leave your desk, you might want to keep the widget, but also have the system send you an email. And if you leave the office, you want an SMS or pager message but no emails. Your other colleagues on call might prefer a phone call, as an SMS or pager message would not wake them up in the middle of the night. So the same event types must trigger different actions at different times, all dependent on external factors. This is where the *event handler* behaviour comes to the rescue.

Events

An *event* represents a state change in the system. It could be a high CPU load, a hardware failure, or a trace event resulting from the activity in a port. An *event manager* is an Erlang process that receives a specific type of event, which could be alarms, warnings, equipment state changes, debug traces, or issues related to network connectivity. When generated, events are sent to the manager in the form of a message, as shown in **Figure 7-1**. For every event generated, the system might want to take a specific set of actions, as discussed earlier: generate SNMP traps; send emails, SMSes or pager messages; collect statistics; print messages to a console; or log the event to file.

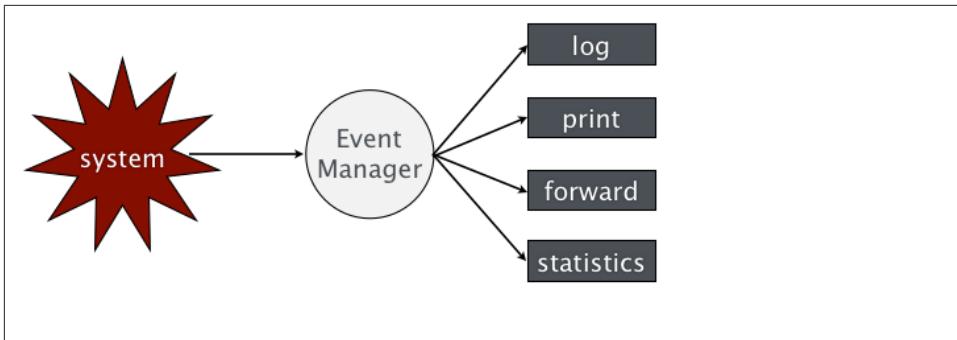


Figure 7-1. Event managers and handlers

Event handlers are behaviour callback modules that handle these types of actions. They subscribe to events sent to a manager, allowing different handlers to subscribe to the same events. Different managers handling different event types can use the same event handler. If a handler allows you to log an event to a file, another allows you to print them to a console, and a third collects statistics, they could be all be used both by the event manager dealing with debug traces and the event manager handling equipment state changes. Functionality to add, remove, query, and upgrade handlers during runtime is provided in the code implementing the event manager. If you were to implement the code managing events and handlers, what would be generic to all Erlang systems and what would be specific to your application?

Table 7-1. Event Handler and Manager Generic and Specific Code

Generic	Specific
<ul style="list-style-type: none"> Starting/stopping the event manager Sending events Sending synchronous requests Forwarding events/requests to handlers Adding/deleting handlers Upgrading handlers 	<ul style="list-style-type: none"> The events The event handlers Initialising event handlers Event handler loop data Handling events/requests Cleaning up

Starting and stopping the event manager processes is generic, as is registering them with an alias. The process name and events sent to the manager are specific, but the producer sending them, the manager receiving them, and the act of calling a handler is generic. The event handlers themselves are specific, as well as what we do to initialise them, alongside the cleaning up when they are removed (or when the event manager is stopped). How the handlers deal with the events is specific, as is their loop data. And finally, upgrading the handlers is generic, but what the individual handlers have to do to hand over their state is specific.

Let's have a look at the behaviour module. While the generic server still acts as its foundation, it is very different from the behaviours we've looked at so far.

Generic Event Managers and Handlers

Generic event handlers and managers are part of the standard library application, and like all other behaviours, are split up into generic and specific code. The `gen_event` module contains all of the generic code. The process running this code is often referred to as the event manager. The callback modules subscribing to the events and handling them through a set of callback functions are called the event handlers. Each handler solves a specific event-driven task and is part of the specific code. Unlike other behaviours, which allow only one callback module per instance, an event manager can take care of zero or more event handlers. But despite the possibility of there being multiple handlers, they will all be executed in a single event manager process.

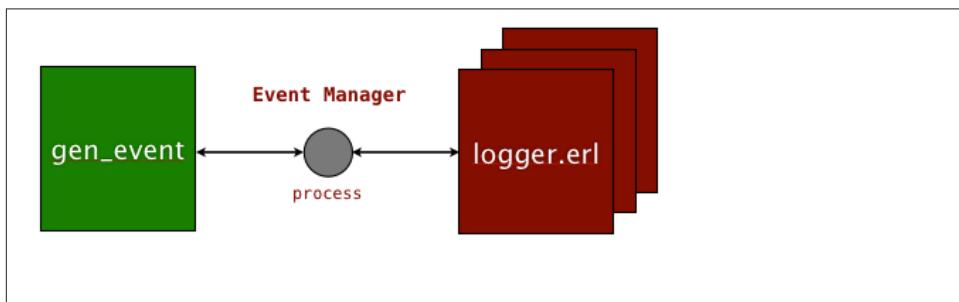


Figure 7-2. Handler callback module

Starting and Stopping Event Managers

The `gen_event:start_link(NameScope)` function call starts a new event manager. `NameScope` specifies the *local* or *global* process name or the *via* module first explained in “[Going Global](#)” on page 91. Should you not want to register the process, use `start_link/0` and communicate with it using its `Pid`. Unlike other behaviours, `start_link/0` accepts no callback modules, arguments, or options. Nor does it invoke any callback functions. All the manager does is set its handler list to the empty list.

```
gen_event:start()  
gen_event:start(NameScope)  
gen_event:start_link()  
gen_event:start_link(NameScope) -> {ok,Pid}  
                                {error,{already_started,Pid}}  
gen_event:stop(NameScope) -> ok
```

Because you are not calling an `init/1` callback function that can return `stop` or `ignore`, or even generate a runtime error, not much can go wrong here unless an event manager or process with the same name is already registered.

Stop the event manager using the `gen_event:stop/1` call.

Adding Event Handlers

Now that we can start and stop our manager, let's implement a handler and add it. Event handlers are added and removed from the event manager process dynamically, at runtime. They are considered more generic than other behaviours because you can add a manager that can handle not only different events, but also different event types received (and pattern matched) by different event managers.

In our logger example, we implement an event handler that logs events and unexpected messages to standard I/O or a file, depending on which parameters are provided when it is added to the manager. As with our other generic behaviours, we start with the `behaviour` directive and export our callback functions.

```
-module(logger).
-behaviour(gen_event).
-export([init/1, terminate/2, handle_event/2, handle_info/2]).

init(standard_io) ->
    {ok, {standard_io, 1}};
init({file, File}) ->
    {ok, Fd} = file:open(File, write),
    {ok, {Fd, 1}};
init(Args) ->
    {error, {args, Args}}.
```

If we pass the `standard_io` atom as an argument, all events will be printed to the shell. Passing `{file, File}`, where `file` is an atom and `File` is a string containing the file name, will log all events to that file.

If we call the `gen_event:add_handler(Name, Mod, Args)` function, the handler implemented in the `Mod` module is added to the event manager. The event manager calls the `Mod:init(Args)` callback function, returning `{ok, LoopData}`, where the `LoopData` refers to that particular handler. In our example, our loop data contains a tuple with either the file descriptor or the atom `standard_io` and the integer 1, a counter incremented every time we receive an event.

To manage multiple events, the event manager stores its handlers and their loop data in a list. [Figure 7-3](#) shows our handler instance and its loop data getting added to the list of other handlers stored by the event manager.

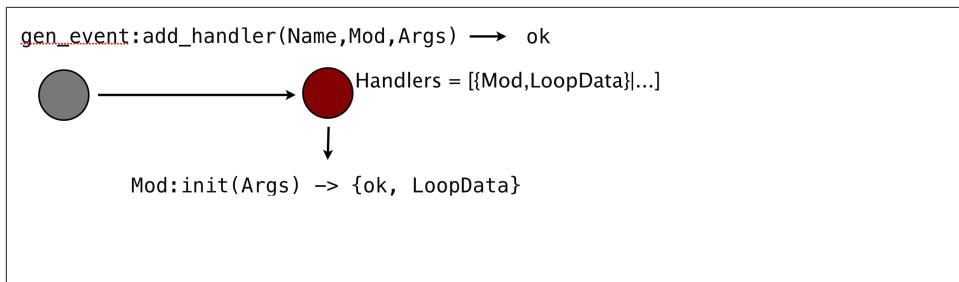


Figure 7-3. Adding Handlers

You can not only add many handlers to a manager, but also add the same handler many times, storing different instances of the loop data. In our case, we could add two logger handlers, one saving everything to file and the other printing the events in the shell. Alternatively, the Mod parameter can be specified as {Module, Id}, where Id can be any Erlang term. If Id is unique, it allows client functions to differentiate between multiple handlers using the same callback module in a particular manager.

```

gen_event:add_handler(NameScope, Mod, Args) -> {'EXIT',Reason}
                                                ok
                                                Term

Mod:init/2 -> {ok, LoopData}
                {ok, LoopData, hibernate}
                Term

```

Adding a non-existent event handler will result in the event manager calling `Mod:init/1` and returning `{'EXIT', Reason}`, where `Reason` is the `undef` runtime error (the undefined function). Should the evaluation of any expression in the `init/1` callback function result in a runtime error, `{'EXIT', Reason}` is returned. Keep in mind that `{'EXIT', Reason}` is a tuple caught within the scope of a try-catch expression, and not an exception. If the `init/1` callback returns a `Term` other than `{ok, LoopData}`, the `Term` itself is returned. This includes the case where the `Term` is the atom `ok` without the `LoopData`, a common beginner error. Whenever `init/1` does not return `{ok, LoopData}`, the event handler is not added to the manager. So just returning `ok` without `LoopData` will not work as you might at first think, as the handler is not added. In our example, if the handler is started with arguments that fail pattern matching in the first two clauses, `init/1` returns `{error, {args, Args}}` and the manager does not add it to its list of handlers. So while `init/1` can return any term, be careful and stick to return values of the format `{ok, LoopData}` and `{error, Reason}` to avoid confusion.

Just like other behaviours, you can make your event manager hibernate in-between events. It is enough for one handler to return `hibernate` for it to happen. Use hibernation with care, and only if events will be intermittent. Hibernating your process will trigger

a full sweep garbage collection before you hibernate and right after waking up. It is not a behaviour you want when receiving large number of events in short intervals.

Deleting an Event Handler

Now that we have added a handler, let's see what we need to do in order to delete it. The `logger` callback module exports the `terminate(Args, LoopData)` callback function. The function is invoked whenever `gen_event:delete_handler(Name, Mod, Args)` is called. `Name` identifies the specific event manager process where our handler is registered; it is either its `Pid`, its `{local, Name}` or `{global, Name}` when using the global name server, or `{via, Name, Module}` when using your own name server. `Mod` specifies the handler you want to delete and `Args` is any valid Erlang term passed as the first argument to `terminate/2`. `Args` could be the reason for termination or just a parameter with instructions needed in the clean up (Figure 7-4).

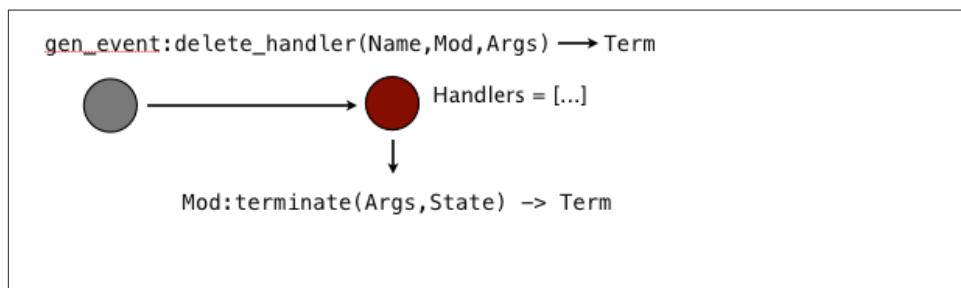


Figure 7-4. Deleting Handlers

So in our example, if we were to remove the logger handler, we would have to cater for the cases where we are printing the logs to standard I/O or to file.

```
terminate(_Reason, {standard_io, Count}) ->
    {count, Count};
terminate(_Reason, {Fd, Count}) ->
    file:close(Fd),
    {count, Count}.
```

When the `terminate/2` function returns, the handler is deleted from the list of handlers in the specific event manager process identified by the `Name` argument to `delete_handler/3`. Other managers using the same handler are not affected. If multiple handlers are registered using the same `Mod`, such as one for logging to `standard_io` and another for logging to a file, they are deleted in the reverse order of their addition. If you stop the manager using `gen_event:stop/1`, all handlers are deleted with reason `stop`.

Note how `terminate/2` returns `Term`. This is the return value of the `delete_handler/3` call. In our example, we return the log counter, `{count, Count}`, which lets the caller of `delete_handler/3` know how many events came through the handlers before they

were terminated. But if we were upgrading the handler, `Term` might be all of the loop data. We will cover upgrades later in this chapter.

Attempting to delete a handler that isn't registered results in `{error, module_not_found}`. Both adding and deleting a handler in a non-existent event manager, irrespective of whether the manager is referenced using a Pid or a registered alias, will result in the calling process terminating with reason `noproc`.

```
gen_event:delete_handler(NameScope, Mod, Args) -> {error,module_not_found}
                                                {'EXIT',Reason}
                                                Term

Mod:terminate/2 -> Term
```

Sending Synchronous and Asynchronous Events

Events can be sent to the manager and forwarded to the handlers synchronously or asynchronously depending on the need to control the rate at which producers generate events. Events are handled by the manager process, which invokes all added handlers sequentially, one at the time. If you send multiple events to the event manager and they need to be handled by several—potentially slow—event handlers, your message queue might grow and result in the reduction of throughput described in [Chapter to Come], so make sure your handler does not become a bottle neck.

Using `gen_event:notify/2`, you will send an asynchronous event to all handlers and immediately return `ok`. The callback function `Mod:handle_event/2` is called for every handler that has been added to the manager, one at a time. `gen_event:sync_notify/2` also invokes the `Mod:handle_event/2` callback function for all handlers. The difference from its asynchronous variant is that `ok` is returned only when all callbacks have been executed.

Let's consider how we might implement the `handle_event/2` callback function for our logger:

```
handle_event(Event, {Fd, Count}) ->
    print(Fd, Count, Event, "Event"),
    {ok, {Fd, Count+1}}.

print(Fd, Count, Event, Tag) ->
    io:format(Fd, "Id:~w Time:~w Date:~w~n"++Tag++":~w~n", [Count, time(), date(), Event]).
```

The `handle_event/2` callback receives an event together with either the atom `standard_io` or the file descriptor of the file opened in the `init/1` callback. The `print/4` function invokes `io:format/3` to output the counter value, current date and time, and the "Event" tag value followed by the event itself.

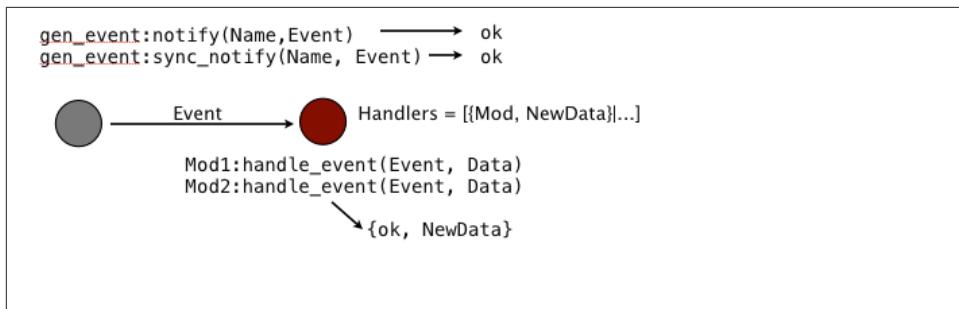


Figure 7-5. Notifications

If our event handler receives any non OTP compliant event originating from links, trapping exits, monitors, monitoring distributed Erlang nodes or a message resulting from `Pid!Msg`, it is handled in the `handle_info/2` callback function of the event handlers.

```

handle_info(Event, {Fd, Count}) ->
    print(Fd, Count, Event, "Unknown"),
    {ok, {Fd, Count+1}}.

```

The implementation of `handle_info/2` for the logger is almost identical to `handle_event/2`, except that it passes the tag value "Unknown" to the `print` function to indicate that it doesn't know the source of the event.

```

gen_event:notify(NameScope, Event)
gen_event:sync_notify(Name, Event) -> ok

Mod:handle_event(Event, Data)
Mod:handle_info(Event, Data) -> {ok, NewData}
                                         {ok, NewData, hibernate}
                                         remove_handler
                                         {swap_handler, Args1, NewData, Handler2, Args2}

```

If a handler returns `remove_handler` from its `handle_event/2` or `handle_info/2` function, `Mod:terminate(remove_handler, Data)` is called and the handler is deleted. We will look at swapping handlers later on in this chapter. Until then, let's make sure that the code in the event handler we have written so far work.

In shell command line 1, we start the event manager without registering or linking it to its parent. Should the shell process crash, the event manager process will not be affected. We proceed by adding a handler and send two notifications, a synchronous and a asynchronous one.

```

1> {ok, P} = gen_event:start().
{ok,<0.34.0>}
2> gen_event:add_handler(P, logger, {file, "alarmlog"}).
ok
3> gen_event:notify(P, {set_alarm, {no_frequency, self()}}).

```

```

ok
4> gen_event:sync_notify(P, {clear_alarm, no_frequency}).
ok

```

Note how both calls return the atom ok. The semantical difference is that shell command 4 does not return ok until all the handlers have executed their handle calls.

In shell command 5, we add a second instance of the handler, this time directing events to standard I/O. In shell command 6, we send a non OTP compliant message which is logged and printed to the shell by the handle_info/2 call back function of the two event handler instances we've added to the manager.

```

5> gen_event:add_handler(P, logger, standard_io).
ok
6> P ! sending_junk.
Id:1 Time:{18,59,25} Date:{2013,4,26}
Unknown:sending_junk
sending_junk

```

In shell command 7 and 8, we read the binary contents of the *alarmlog* file and print them out in the shell. We see the first two events we sent asynchronously and synchronously, as well as the unknown message received by the handle_info/2 call.

```

7> {ok, Binary} = file:read_file("alarmlog").
{ok,<<"Id:1 Time:{18,59,10} Date:{2013,4,26}\nEvent:{set_alarm,{no_frequency,...}}>>}
8> io:format(Binary).
Id:1 Time:{18,59,10} Date:{2013,4,26}
Event:{set_alarm,{no_frequency,<0.32.0>}}
Id:2 Time:{18,59,14} Date:{2013,4,26}
Event:{clear_alarm,no_frequency}
Id:3 Time:{18,59,25} Date:{2013,4,26}
Unknown:sending_junk
ok
9> gen_event:delete_handler(P, freq_overload, stop).
{error,module_not_found}
10> gen_event:stop(P).
ok

```

We wrap up this example by trying to delete `freq_overload`, an event handler that has not been added to this event manager. As expected, this returns the error `module_not_found`. Finally, we stop the event manager, by default terminating all of the event handlers.

Download the logger handler from the book's code repository and take it for a spin. Test sending it synchronous and asynchronous messages when the event manager has been stopped (or has crashed), start it using `start_link` and make the shell crash. Finally, try to figure out what happens if you provide an invalid filename when adding the handler.

Retrieving Data

Let's implement another event handler, one that stores metrics. So every time we log an event, we also bump up a counter in an ets table that tells us how many times this event has been logged. If it is the first occurrence of the event, we create a new entry in the table. Have a look at the code, and if necessary, use the manual pages of the ets module.

```
-module(counters).
-behaviour(gen_event).
-export([init/1, terminate/2, handle_event/2, handle_info/2]).
-export([get_counters/1, handle_call/2]).

get_counters(Pid) ->
    gen_event:call(Pid, counters, get_counters).

init(_) ->
    TableId = ets:new(counters, []),
    {ok, TableId}.

terminate(_Reason, TableId) ->
    Counters = ets:tab2list(TableId),
    ets:delete(TableId),
    {counters, Counters}.

handle_event(Event, TableId) ->
    try ets:update_counter(TableId, Event, 1) of
        _ok -> {ok, TableId}
    catch
        error:_ -> ets:insert(TableId, {Event, 1}),
        {ok, TableId}
    end.

handle_call(get_counters, TableId) ->
    {ok, {counters, ets:tab2list(TableId)}}, TableId}.

handle_info(_, TableId) ->
    {ok, TableId}.
```

Of interest in this example is how we retrieve the counters. Using `gen_event:sync_event/2` would not have worked, as despite it being synchronous, it forwards the event to all handlers and returns `ok`. We need to specify the handler to which we want to send our synchronous message, and we do so using the `gen_event:call(NameScope, Mod, Message)` function.

The event handler synchronously receives the request in the `Mod:handle_call/2` callback and returns a tuple of the format `{ok, Reply, NewData}`, where `Reply` is the return value of the request.

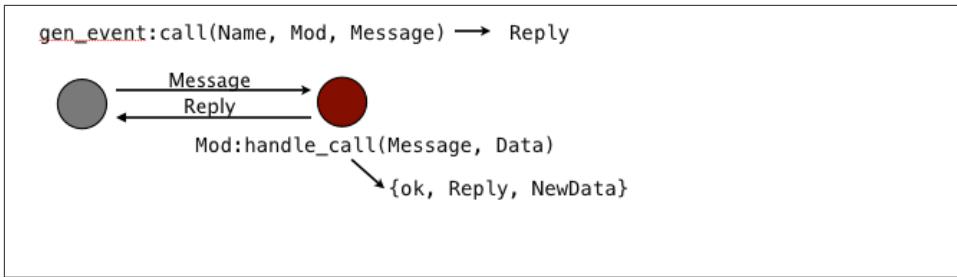


Figure 7-6. Calls

```

gen_event:call(NameScope, Mod, Request)
gen_event:call(NameScope, Mod, Request, Timeout) -> Reply
                                                {error, bad_module}
                                                {error, {'EXIT', Reason}}
                                                {error, Term}

Mod:handle_call(Event, Data) -> {ok, Reply, NewData}
                                         Term

```

The default timeout in `gen_event:call/3` is set to 5000 ms. It can be overridden by passing the `Timeout` value as an integer in milliseconds or the atom `infinity`. If `Mod` is not an event handler that has been added to `NameScope`, `{error, bad_module}` is returned. If the callback function `handle_call/2` terminates abnormally when handling the request, expect `{error, {'EXIT', Reason}}`. And finally, if `handle_call/2` returns any term other than `{ok, Reply, NewData}`, the return value of `gen_event:call` will be `{error, Term}`. In both of these error cases, the handler is removed from the list managed by the event manager without affecting the other handlers.

```

1> {ok, P} = gen_event:start().
{ok,<0.34.0>}
2> gen_event:add_handler(P, counters, {}).
ok
3> gen_event:notify(P, {set_alarm, {no_frequency, self()}}).
ok
4> gen_event:notify(P, {event, {frequency_denied, self()}}).
ok
5> gen_event:notify(P, {event, {frequency_denied, self()}}).
ok
6> counters:get_counters(P).
{counters,[{{event,{frequency_denied,<0.34.0>}},2},
           {{set_alarm,{no_frequency,<0.34.0>}},1}]}

```

Handling Errors and Invalid Return Values

It is not just when its `handle_call/2` terminates abnormally or returns an invalid reply that a handler gets deleted. An abnormal termination in any of its callbacks will result in deletion. The event manager and other handlers are not affected. This differs from

other behaviours in that the event handler is silently removed, without any notifications being sent to the event manager's supervisor. What also differs from other behaviours is that the event manager will by default trap exits. The assumption is that event managers are added and removed dynamically and independently of each other, and as a result, a crash should not affect anything in its surrounding environment. While fault isolation is a good property, failing silently isn't.

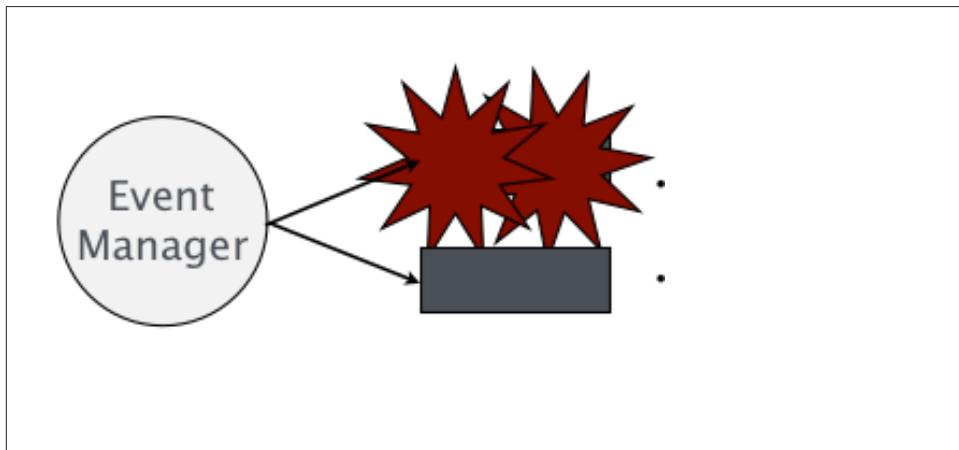


Figure 7-7. Handler Crash

To better understand how abnormal termination in event handlers works, let's use the following code snippet as an example:

```
-module(crash_example).
-behaviour(gen_event).
-export([init/1, terminate/2, handle_event/2]).

init(normal) -> {ok, null};
init(return) -> error;
init(ok)      -> ok;
init(crash)   -> exit(crash).

terminate(_Reason, _LoopData) -> ok.

handle_event(crash, _LoopData) -> 1/0;
handle_event(return, _LoopData) -> error.
```

Depending on what parameters we send to the event handler when adding it to the event manager and notifying it of an event, we generate runtime errors and return invalid values. Step through the shell commands in the example, mapping all requests to the error conditions that occur.

```
1> {ok,P}=gen_event:start().
{ok,<0.34.0>}
```

```

2> gen_event:which_handlers(P).
[]
3> gen_event:add_handler(P, crash_example, return).
error
4> gen_event:which_handlers(P).
[]
5> gen_event:add_handler(P, crash_example, normal).
ok
6> gen_event:which_handlers(P).
[crash_example]
7> gen_event:notify(P, crash).
ok
=ERROR REPORT==== 27-Apr-2013::09:27:49 ===
** gen_event handler crash_example crashed.
** Was installed in <0.34.0>
** Last event was: crash
** When handler state == []
** Reason == {badarith,
              [{crash_example,handle_event,2,
                [{file, "crash_example.erl"},{line,13}]}],
              ...}]
gen_event:which_handlers(P).
[]
8> gen_event:add_handler(P, crash_example, normal).
ok
9> gen_event:notify(P, return).
ok
=ERROR REPORT==== 27-Apr-2013::09:28:41 ===
** gen_event handler crash_example crashed.
** Was installed in <0.34.0>
** Last event was: return
** When handler state == []
** Reason == error
10> gen_event:which_handlers(P).
[]

```

While error reports are generated (covered in more detail in “[The SASL Application](#)” on page 222), no runtime errors occur, and as a result, no EXIT signals are generated. Sending notifications can fail silently, resulting in the handler being deleted without anyone (process or human) noticing. You get around this problem by connecting a handler to the calling process using `gen_event:add_sup_handler/3`. It works in the same way as `add_handler/3`, with the side effect that the calling process is now monitoring the handler, and the calling process is being monitored by the newly added instance of the handler in the manager.

If an exception occurs or an incorrect return value is returned in callbacks handling events, a message of the format `{gen_event_EXIT, Mod, Reason}` is sent to the process that added the handler. `Reason` can be one of the following:

- **normal** if a callback function returned `remove_handler` or the handler was removed using `delete_handler/3`
- **shutdown** if the event manager is being stopped, either by its supervisor or by the `stop/1` call
- **{'EXIT', Term}** if a runtime error occurred
- **Term** if the callback returned anything other than `{ok, LoopData}` or `{ok, Reply, LoopData}`
- **{swapped, NewMod, Pid}**, where `Pid` has swapped the handler.

We look into swapping handlers in the next section.

Monitoring goes two ways. If the process that added the handler terminates, the handler is removed with `{stop, Reason}` as an argument. This ensures that multiple instances of the handler are not included in the manager should the handler be added by a behaviour stuck in a cyclic restart.



Fail Loudly!

If you are writing a system with requirements on high availability and fault tolerance, the last thing you want is a handler being silently deleted or failing in `init/1` and not being added at all. Always check the return value of the `add_handler/3` and `add_sup_handler/3` calls. If you have to use `add_handler`, ensure that you execute any code that might fail as a result of a bug, external dependencies (such as a disk full error) or corrupt data within the scope of a try-catch expression. Where possible, use `add_sup_handler/3`, pattern matching on its return value to ensure that the handler has been properly added, and pay attention to all exception messages that you receive as a result. The last thing you want is your alarm system to fail without raising any alarms!

Swapping Event Handlers

The event manager provides functionality to swap handlers during runtime. It allows a handler to pass its state to a new handler, ensuring that no events are lost in the process. The second parameter of the `gen_event:swap_handler/3` call is a tuple containing the name of the handler callback module we want to replace, together with the arguments passed to its `terminate` function. The third parameter is a tuple with the callback module of the new handler and the arguments passed to its `init` function.

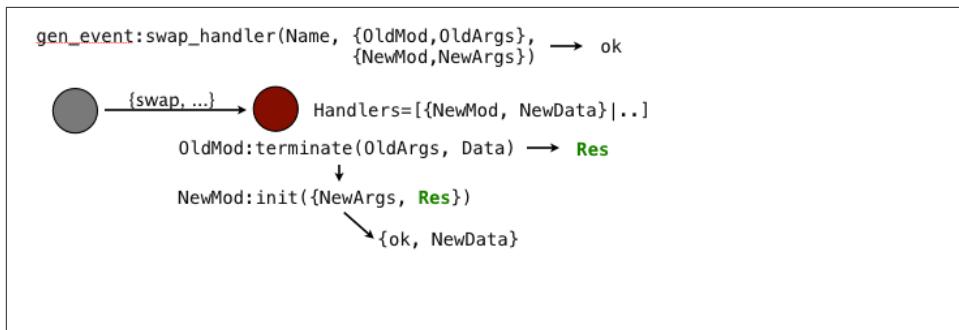


Figure 7-8. Swapping Handlers

The `terminate` callback function in the old handler is first called. Its return value `Res` is passed in a tuple together with the arguments intended for the `init` function of the new handler. It couldn't be simpler! If you want to swap the handler and start supervising the connection between the handler and the calling process, use `gen_event:swap_sup_handler/3`. The handler you are swapping does not have to be supervised.

An example is probably the best way to demonstrate swapping. Let's extend our logger handler to be able to flip between logging to file and printing to standard I/O. We extend our `terminate` function to handle the reason `swap`, returning `Res`, a tuple of the format `{Type, Count}`. `Type` is either the file descriptor or the atom `standard_io` and `Count` is the unique ID for the next item to be logged. As we do not know what the logger we are swapping to wants to do with the events, we do not close the file and let the handler deal with it in its `init/1` call.

In the `init` call, we add two cases where we accept the same `Args` as when we are adding the handler, but also the results sent back from `terminate`. So if we are logging to file and want to swap to standard I/O, we close the file and return `{ok, {standard_io, Count}}`. If we are printing to standard I/O, we open the file and start writing events in it. In both cases, we retain whatever value `Count` is set to.

```

init({standard_io, {Fd, Count}}) when is_pid(Fd) ->
  file:close(Fd),
  {ok, {standard_io, Count}};
init({File, {standard_io, Count}}) when is_list(File) ->
  {ok, Fd} = file:open(File, write),
  {ok, {Fd, Count}};
...
terminate(swap, {Type, Count}) ->
  {Type, Count};
...

```

If we test our code, starting the manager, adding the logger handler, raising an alarm, swapping the handlers and raising a second alarm, we get the following results:

```
1> {ok, P} = gen_event:start().
{ok,<0.34.0>}
2> gen_event:add_handler(P, logger, {file, "alarmlog"}).
ok
3> gen_event:notify(P, {set_alarm, {no_frequency, self()}}).
ok
4> gen_event:swap_handler(P, {logger, swap}, {logger, standard_io}).
ok
5> gen_event:notify(P, {set_alarm, {no_frequency, self()}}).
Id:2 Time:{10,1,16} Date:{2013,4,27}
Event:{set_alarm,{no_frequency,<0.32.0>}}
ok
6> {ok, Binary}=file:read_file("alarmlog"), io:format(Binary).
Id:1 Time:{10,1,16} Date:{2013,4,27}
Event:{set_alarm,{no_frequency,<0.32.0>}}
ok
```

Wrapping it all up

Now that we have a handler, let's wrap it in a module hiding the event manager API into a more intuitive and application-specific set of functions. We will do this in the *freq_overload* module, which is responsible for starting the manager along with providing an API for setting and clearing the no_frequency alarm and generating events when a client is denied a frequency. It also provides a wrapper around the functions used to add and delete handlers. We leave the handler-specific function calls such as retrieving the counters or swapping from file to standard I/O local to the handlers themselves.

```
-module(freq_overload).

-export([start_link/0, add/2, delete/2]).

-export([no_frequency/0, frequency_available/0, frequency_denied/0]).
```



```
start_link() ->
    case gen_event:start_link({local, ?MODULE}) of
        {ok, Pid} -> add(counters, {}), add(loggers, {}),
                        {ok, Pid}.
    end.
```



```
no_frequency()      -> gen_event:notify(?MODULE, {set_alarm, {no_frequency, self()}}).
frequency_available() -> gen_event:notify(?MODULE, {clear_alarm, no_frequency}).
frequency_denied()   -> gen_event:notify(?MODULE, {event, {frequency_denied, self()}}).
```



```
add(M,A)      -> gen_event:add_sup_handler(?MODULE, M, A).
delete(M,A)   -> gen_event:delete_handler(?MODULE,M,A).
```

Note how are adding the frequencies in our `freq_overload:start_link/0` call. This ensures that if the event manager is restarted, the counters and logger handlers will also be added. The downside is that we are unable to supervise the handler from the event

manager process in case it crashes. If you want another process to monitor the handlers, use `freq_overload:add/2` which uses `gen_event:add_sup_handler/3`.

When setting alarms and raising events, we are also including the Pid of the frequency allocator. This allows us to differentiate among different allocators (called the alarm or event originator), allowing operational staff to determine which servers are overutilized and need to be allocated a larger frequency pool. We want to raise an alarm every time the allocator runs out of frequencies and clear it when a frequency becomes available. If a client allocates the last frequency, we call `freq_overload:no_frequency/0` setting the `no_frequency` alarm. If a frequency is deallocated in a state where there were no frequencies available, we clear the alarm by calling `freq_overload:frequency_available/0`. We also raise an event every time a user is denied a frequency by calling the function `freq_overload:frequency_denied/0`. We handle this as a separate event, as we might be out of frequencies but do not reject requests, as attempts to allocate a frequency are made. The code additions should be straightforward.

```
allocate({[], Allocated}, _Pid) ->
    freq_overload:frequency_denied()
    {[[], Allocated], {error, no_frequency}};;
allocate({[Res|Resources], Allocated}, Pid) ->
    case Resources of [] -> freq_overload:no_frequency(); _ -> ok
    end,
    {[Resources, [{Res, Pid}|Allocated]], {ok, Res}}.

deallocate({Free, Allocated}, Res) ->
    case Free of [] -> freq_overload:frequency_available(); _ -> ok
    end,
    NewAllocated = lists:keydelete(Res, 1, Allocated),
    {[Res|Free], NewAllocated}.
```

Now that we have fixed other code in the frequency allocator and implemented our `freq_overload` event manager, let's add the `logger` and `counters` handlers to the event manager and run them alongside each other. Along with raising alarms, we also log them.

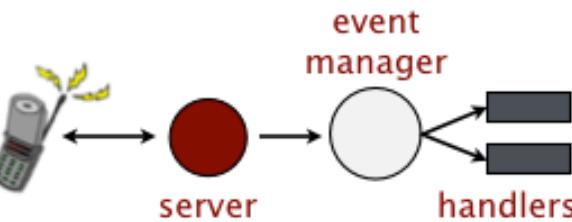


Figure 7-9. Handler Example

We start the frequency server, the event manager and add a second logger event handler, this one printing to the shell. In our example, the frequency allocator had six frequencies. In shell command 5, we allocate all of them, raising a no_frequency alarm. This happens despite the last request being successful and returning {ok, 15}.

```

1> frequency:start_link().
{ok,<0.34.0>}
2> freq_overload:start_link().
{ok,<0.36.0>}
3> freq_overload:add(logger, standard_io).
ok
4> frequency:allocate(), frequency:allocate(), frequency:allocate(),
   frequency:allocate(), frequency:allocate(), frequency:allocate().
Id:1 Time:{10,41,25} Date:{2015,2,28}
Event:{set_alarm,{no_frequency,<0.34.0>}}
{ok,15}
5> frequency:allocate().
Id:2 Time:{10,41,46} Date:{2015,2,28}
Event:{event,{frequency_denied,<0.34.0>}}
{error,no_frequency}
6> frequency:allocate().
Id:3 Time:{10,42,0} Date:{2015,2,28}
Event:{event,{frequency_denied,<0.34.0>}}
{error,no_frequency}
7> frequency:deallocate(15).
Id:4 Time:{10,42,16} Date:{2015,2,28}
Event:{clear_alarm,no_frequency}
ok
8> counters:get_counters(freq_overload).
{counters,[{{set_alarm,{no_frequency,<0.34.0>}},1},
           {{clear_alarm,no_frequency},1},
           {{event,{frequency_denied,<0.34.0>}},2}]}

```

Having set the alarm, we try to allocate two frequencies and fail both times. We clear the alarm when deallocating a frequency in shell command 8. When we retrieve the

counters, we see that a frequency was denied twice and that the no frequency alarm was set and cleared once.

The SASL Alarm Handler

We've been talking about alarm handlers in this chapter without giving a proper definition. The time has come to set the record straight. An alarm handler is the part of the system that records ongoing issues and takes appropriate actions. If your system reaches a high memory mark or is running out of disk space (or frequencies), you will want to set (or raise) an alarm. When memory usage decreases or old log files are deleted, the respective alarms are cleared. At any point in time, it should be possible to inspect the list of active alarms and get a snapshot of ongoing issues.

The SASL alarm handler process is an event manager and handler that comes as part of the Erlang run-time system and provides this functionality. It is a very basic alarm handler you are encouraged to replace or complement in your own project when more functionality is required. The philosophy of developing Erlang systems is to start simple and add complexity as your system grows. That is exactly what they have done with the alarm handler.

Depending on how you have installed Erlang on your computer, the SASL alarm handler might already have been started. Type `whereis(alarm_handler)`. in your shell to find out. If you get back the atom `undefined`, start the alarm handler by typing `application:start(sasl)`. in the shell. You might get some progress reports printed out in the shell, once again, depending on how you installed Erlang (don't worry about the reports for now).

If `whereis/1` returned a Pid, the alarm handler is already running and you do not need to do anything other than trying it out.

```
1> whereis(alarm_handler).
<0.42.0>
2> alarm_handler:set_alarm({103, fan_failure}).

=INFO REPORT==== 26-Apr-2013::08:23:27 ===
    alarm_handler: {set,{103,fan_failure}}
ok
3> alarm_handler:set_alarm({104, cabinet_door_open}).

=INFO REPORT==== 26-Apr-2013::08:23:43 ===
    alarm_handler: {set,{104,cabinet_door_open}}
ok
4> alarm_handler:clear_alarm(104).

=INFO REPORT==== 26-Apr-2013::08:24:04 ===
    alarm_handler: {clear,104}
ok
```

```
5> alarm_handler:get_alarms().
[{:103,fan_failure}]
```

In our example, picture a rack in which the cooling fan fails. A system administrator goes to the rack, opens the cabinet door to inspect what is going on, closes it and heads off to order a replacement fan. What we've done is raised two alarms with id 103 and 104. These ids are used to clear the alarm, something that happens in shell command 5 when the cabinet door is closed. The wrapper around the SASL event manager and event handler exports the following functions:

```
alarm_handler:set_alarm({AlarmId, Description}) -> ok
alarm_handler:clear_alarm(AlarmId) -> ok
alarm_handler:get_alarms() -> [{AlarmId, Description}]
```

In a complex system, you might have hundreds of alarms of varying severities, where clearing one will by default clear half a dozen other ones dependent on it. You will want to keep accurate statistics, log everything, and in advanced systems, run agents that take immediate action. In the case of the fan failure, for example, you want to start shutting down all equipment in that cabinet to avoid overheating. The existing handler does none of this and will not scale. But to start off, it works and fits in with the iterative design, develop and test cycles that are the norm when programming Erlang.

To replace or complement the existing handler is easy. You need to handle the events {set_alarm, {AlarmId, AlarmDescr}} and {clear_alarm, AlarmId}. If you want to swap the existing handler using

```
gen_event:swap_handler(alarm_handler,
                       {alarm_handler, swap}, {NewHandler, Args})
```

the init function in your new handler should pattern match the argument {Args, {alarm_handler, Alarms}}, where Args is passed in the swap_handler/3 call and {alarm_handler, Alarms} is the term returned from the handler terminate/2 call of the old handler. Alarms is a list of {AlarmId, Description} tuples.

Summing Up

In this chapter, we introduced how events are handled by the event manager behaviour. You should by now have a good understanding of the advantages of using the gen_event behaviour instead of rolling your own or increasing the complexity of one of your subsystems by integrating this functionality in it. The biggest difference between the event manager and other OTP behaviours is the one-to-many relationship, where you can associate many event handlers with one event manager. The most important functions and callbacks we have covered include:

Table 7-2. *gen_event callbacks*

gen_event function or action	gen_event callback function
gen_event:start/0, gen_event:start/1, gen_event:start_link/0, gen_event:start_link/1	
gen_event:add_handler/3, gen_event:add_sup_handler/3	Module:init/1
gen_event:swap_handler/3, gen_event:swap_sup_handler/3	Module1:terminate/2, Module2:init/1
gen_event:notify/2, gen_event:sync_notify/2	Module:handle_event/2
gen_event:call/3, gen_event:call/4	Module:handle_call/2
gen_event:delete_handler/3	Module:terminate/2
gen_event:stop/1	Module:terminate/2
Pid ! Msg, monitors, exit messages, messages from ports and socket, node monitors, and other non OTP messages	Module:handle_info/2

Before reading on, make sure you review the manual pages for the *gen_event* module. An example that complements the ones in this chapter is the *alarm_handler* module. Read through the code and you will notice how they have integrated the client functions to start and stop the event manager as well as the handler functions themselves.

What's Next?

The event manager is the last worker behaviour we cover. Event managers, along with generic servers, finite state machines and behaviours you have written yourself are all started and monitored in supervision trees. The next chapter covers the supervisor behavior, a behaviour whose only task is to start and monitor other supervisors and workers. We show you how to write your own behaviours in [Chapter 10](#).

CHAPTER 8

Supervisors

Now that we are able to monitor and handle predictable errors, such as running out of frequencies, we need to tackle unexpected errors arising as the result of corrupt data or bugs in the code. The catch is that unlike the errors returned to the client in the frequency allocator or alarms raised by the event managers, we will not know what the unexpected errors are until they have occurred. We could speculate, guess, and try to add code that handles the unexpected and hope for the best. Using automated test generation tools based on property-based testing, such as QuickCheck or PropEr, can definitely help create failure scenarios you would never devise on your own. But unless you have supernatural powers, you will never be able to predict every possible unexpected error that might occur, let alone handle it before knowing what it is.

Too often, developers try to cater for bugs or corrupt data by implementing their own error handling and recovery strategies in their code, with the result that they increase the complexity (and maintainability) of the code, handle only a fraction of the issues that can arise, and more often than not, end up inserting more bugs in the system than they solve. After all, how can you handle a bug if you don't know what the bug *is*? Have you ever come across a c programmer who checks the return value of their printf statements, but is unsure of what to do if an error actually occurs. If you've come to Erlang from another language that supports exception handling, such as Java or C++, how many times have you seen catch expressions that contain *TODO* comments to remind the development team to fix the exception handlers at some point in the future — a point that unfortunately never arrives?

This is where the generic supervisor behaviour makes its entrance. It takes over the responsibility for the unexpected error handling and recovery strategy from the developer. The behaviour, in a deterministic and consistent manner, handles monitoring, restart strategies, race conditions, and borderline cases most developers would not think of. This results in simpler worker behaviours, as well as a well-considered error recovery strategy. Let's examine how the supervisor behaviour works.

Supervision Trees

Supervisors are processes whose only task is to start, monitor, and manage children. They spawn processes and link themselves to these processes. By trapping exits and receiving EXIT signals, the supervisors can take appropriate actions when something unexpected occurs. Actions vary from restarting the child to not restarting it, terminating some or all the children that are linked to the supervisor, or even terminating itself. Child processes can be both supervisors and workers.

Fault tolerance is achieved by creating *supervision trees*, where the supervisors are the nodes and the workers are the leaves. Supervisors on a particular level monitor and handle children in the subtree they have started.

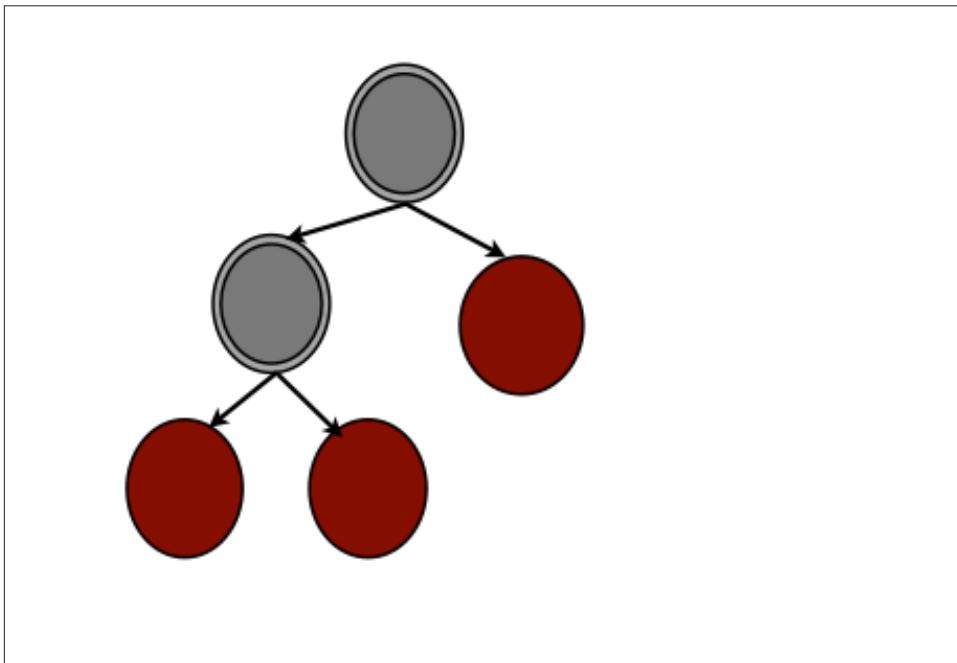


Figure 8-1. Supervision Trees

Figure 8-1 uses a double ring to denote processes that trap exits. Only supervisors are trapping exits in our example, but there is nothing stopping workers from doing the same.

Let's start by looking at a simple supervisor, which will allow us to better appreciate what needs to happen behind the scenes. Given a list of children, our simple supervisor starts and links itself to them. If any of the children terminate abnormally, the simple supervisor immediately restarts them. If they instead terminate normally, they are removed

from the supervision tree and no further action is taken. Stopping the supervisor results in all of the children being unconditionally terminated.

Here is the code that starts the supervisor and child processes:

```
-module(my_supervisor).
-export([start/2, init/1, stop/1]).

start(Name, ChildSpecList) ->
    register(Name, Pid = spawn(?MODULE, init, [ChildSpecList])),
    {ok, Pid}.

stop(Name) -> Name ! stop.

init(ChildSpecList) ->
    process_flag(trap_exit, true),
    loop(start_children(ChildSpecList)).

start_children(ChildSpecList) ->
    [{element(2, apply(M, F, A)), {M, F, A}} || {M, F, A} <- ChildSpecList].
```

When starting `my_supervisor`, we provided the `init/1` function with child specifications. This a list of `{Module, Function, Arguments}` tuples containing the functions that will spawn and link the child process to its parent. We assume that this function always returns `{ok, Pid}`, where `Pid` is the process ID of the newly spawned child. Any other return value is interpreted as a startup error.

When `Module:Function/Args` does not return `{ok, Pid}`, the supervisor terminates with a *bad match* error. Furthermore, if `Module` does not exist, `Function` is not exported, or `Args` contains the wrong number of arguments, the supervisor process terminates with a runtime exception. When the supervisor terminates, the runtime ensures that all processes linked to it receive an EXIT signal. If the linked child processes are not trapping exits, they will terminate. But if they are trapping exits, they need to handle the EXIT signal, most likely by terminating themselves, thereby propagating the EXIT signal to other processes in their link set.

It is a valid assumption that nothing abnormal should happen when starting your system. If a supervisor is unable to correctly start a child, it terminates all of its children and aborts the startup procedure. While we are all for a resilient system that tries to recover from errors, startup failures is where we draw the line.

The supervisor loops with a tuple list of the format `{Pid, {Module, Function, Argument}}` returned from the `start_children/1` call. This tuple list is the supervisor state. We use this information if a child terminates abnormally, mapping the `Pid` to the function used to start it and needed to restart it. If we want to register supervisors with an alias, we pass it as an argument using the variable name. The reason for not hard-coding it in the module is that you will often have multiple instances of a supervisor in your Erlang node.

```

loop(ChildList) ->
    receive {'EXIT', Pid, normal} ->      loop(lists:keydelete(Pid,1,ChildList)); {'EXIT', Pid,
    end.

restart_child(Pid, ChildList) ->
    {value, {Pid, {M,F,A}}} = lists:keysearch(Pid, 1, ChildList),
    {ok, NewPid} = apply(M,F,A),
    [{NewPid, {M,F,A}}|lists:keydelete(Pid,1,ChildList)].

terminate(ChildList) ->
    lists:foreach(fun({Pid, _}) -> exit(Pid, kill) end, ChildList).

```

Having started all the children, the supervisor process enters the receive - evaluate loop. Notice how this is no different than the process skeleton described in “[Process Skeletons](#)” on page 47, and similar to the generic loop in servers, finite state machines and event handler processes. The only difference from other behaviour processes we have implemented in Erlang is that here we handle only EXIT messages and take specific actions when receiving the stop message.

In our supervisor, if the child process terminates with reason *normal*, it is deleted from the ChildSpecList and the supervisor continues monitoring other children. If it terminates with a reason other than *normal*, the child is restarted and its new Pid is replaced in the tuple {Pid, {Module, Function, Argument}} of the child specification list. If our supervisor receives the stop message, it traverses through its list of child processes, terminating each one.

Let’s try out *my_supervisor* with the Erlang implementation of the coffee finite state machine. If you do the same, don’t forget to compile *coffee.erl* and *hw.erl*. Actually, on second thought, don’t compile *hw.erl*. Start your coffee FSM from the supervisor and see what happens if the *hw.erl* stub module is not available. When all of the error reports are being printed out, compile or load the *hw.erl* from the shell, making it accessible.

```

1> my_supervisor:start(coffee_sup, [{coffee, start_link, []}]).  

{ok, <0.39.0>}  
  

=ERROR REPORT==== 4-May-2013::08:26:51 ===  

Error in process <0.468.0> with exit value: {undef,[{hw,reboot,[],[]},{coffee,init,0,[...]}]}  
  

...<snip>...  
  

=ERROR REPORT==== 4-May-2013::08:26:58 ===  

Error in process <0.474.0> with exit value: {undef,[{hw,reboot,[],[]},{coffee,init,0,[...]}]}  
  

2> c(hw).  

Machine:Rebooted Hardware  

Display:Make Your Selection  

{ok,hw}  

3> Pid = whereis(coffee).  

<0.476.0>  

4> exit(Pid, kill).

```

```

Machine:Rebooted Hardware
Display:Make Your Selection
true
5> whereis(coffee).
<0.479.0>
6> my_supervisor:stop(coffee_sup).
stop
7> whereis(coffee).
undefined

```

So, what's is happening? The coffee FSM, in its `init` function, calls `hw:reboot/0` causing an *undef* error because the module cannot be loaded. The supervisor receives the EXIT signal and restarts the FSM. The restart becomes cyclic, because restarting the FSM will not solve the issue; it will continue to crash until the module is loaded and becomes available. Compiling the `hw.erl` module in shell command 2 also loads the module, allowing the coffee FSM to initialize itself and start correctly. This puts an end to the cyclic restart.

Cyclic restarts happen when restarting a process after an abnormal termination does not solve the problem, resulting in the process crashing and restarting again. The supervisor behaviour has mechanisms in place to escalate cyclic restarts. We will cover them later on in this chapter. Now back to our example.

In shell command 3, we find the Pid of the FSM and use it to send an exit signal, which causes the coffee FSM to terminate. It is immediately restarted, something visible from the printouts in the shell generated in the `init/0` function. We stop the supervisor in shell command 6, which, as a result, also terminates its workers.

Now comes the question we've been asking for every other behaviour. Have a look at the code in `my_supervisor.erl` and ask yourself what is generic and what is specific.¹

Spawning the supervisor and registering it will be the same, irrespective of what children the supervisor starts or monitors. Monitoring the children and restarting them is also generic, as is stopping the supervisor and terminating all of the children. In other words, all of the code in `my_supervisor.erl` is generic. All of the specific functionality is passed as variables. It includes the child spec list, the order in which the children have to be started, and the supervisor alias.

1. If you are someone who reads footnotes, good for you, as you can now consider yourself warned that this is a trick question.

Table 8-1. The Supervisor Generic and Specific Code

Generic	Specific
<ul style="list-style-type: none"> • Spawning the supervisor • Starting the children • Monitoring the children • Restarting the children • Stopping the supervisor • Cleaning up 	<ul style="list-style-type: none"> • What children to start • Specific child handling <ul style="list-style-type: none"> — Start, Restart — Child dependencies • Supervisor name • Supervisor behaviours

Although *my_supervisor* will cater for some use cases, it barely scrapes the surface of what a supervisor has to do. We've decided to keep our example simple, but could have added more specific parameters. We've already seen that child startup failures cause endless retries. Supervisors should provide the ability to specify the maximum number of restarts within a time interval, to allow a child process to restart or take further action if it does not. Or what about dependencies? If a child terminates, shouldn't the supervisor offer the option of terminating and restarting other children that depend on that child? These are some of the configuration parameters included in the OTP supervisor behaviour library module, which we will now cover.

OTP Supervisors

In OTP, we structure our programs with one or more supervision trees. We group together, under the same subtree, workers that are either similar in nature or are mutually dependent on each other, starting them in order of dependency. When describing supervision trees, worker behaviours are usually represented as circles, while supervisors are represented as squares. [Figure 8-2](#) shows what the supervision structure of the frequency allocator example we've been working on could look like.

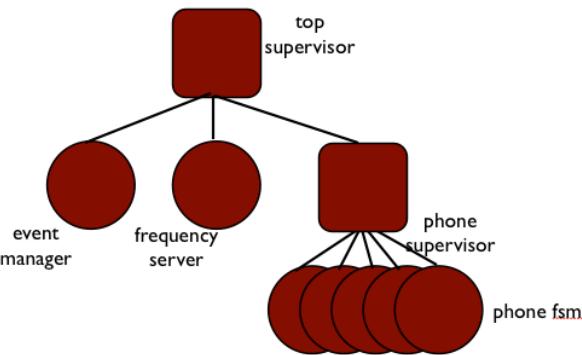


Figure 8-2. Supervision Trees

Taking dependencies into consideration, the top supervisor first starts the event manager worker that handles alarms, because it is not dependent on any other worker. The top supervisor then starts the frequency allocator, because it sends alarms to the event manager. The last process on that level is a phone supervisor, which takes responsibility for starting and monitoring all of the finite state machines representing the cell phones.

Note how we have grouped dependent processes together in one subset of the tree and related processes in another, starting them from left to right in order of dependency. This forms part of the supervision strategy of your system and in some situations is put in place not by the developer, who focuses only on what particular workers have to do, but by the architect, who has an overall view and understanding of the system and how the different components interact with each other.

The Supervisor Behaviour

In OTP, the supervisor behaviour is implemented in the `supervisor` library module. Like all behaviours, the callback module is used for non-generic code, including the behaviour and version directives. The supervisor callback module needs to export a single callback function used at startup to configure and start the subset of the tree handled by that particular supervisor (Figure 8-3).

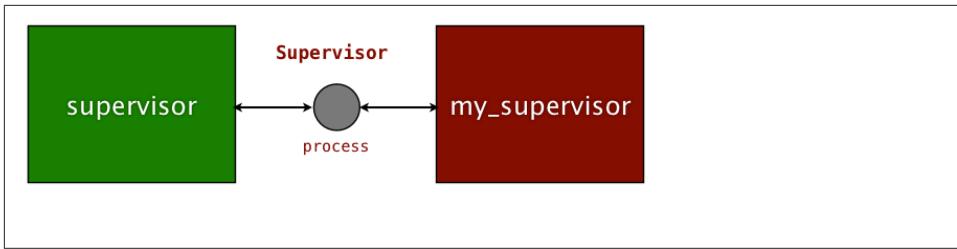


Figure 8-3. Generic Supervisors

You may have guessed that the single exported function is the `init/1` function, containing all of the specific supervisor configuration. The callback module usually also provides the function used to start the supervisor itself. Let's look at these calls more closely.

Starting the Supervisor

As a first step in getting our complete supervision tree in place, we will create a supervisor that starts and monitors our frequency server and overload event manager. Because the frequency server calls the overload event manager, it has a dependency toward the event manager. That means that the overload manager needs to be started before the frequency server, and if the overload manager terminates, we need to terminate the frequency server as well before restarting them both. Supervision tree diagrams, such as that in [Figure 8-4](#), show not only the supervision hierarchy, but also dependencies and the order in which processes are started.

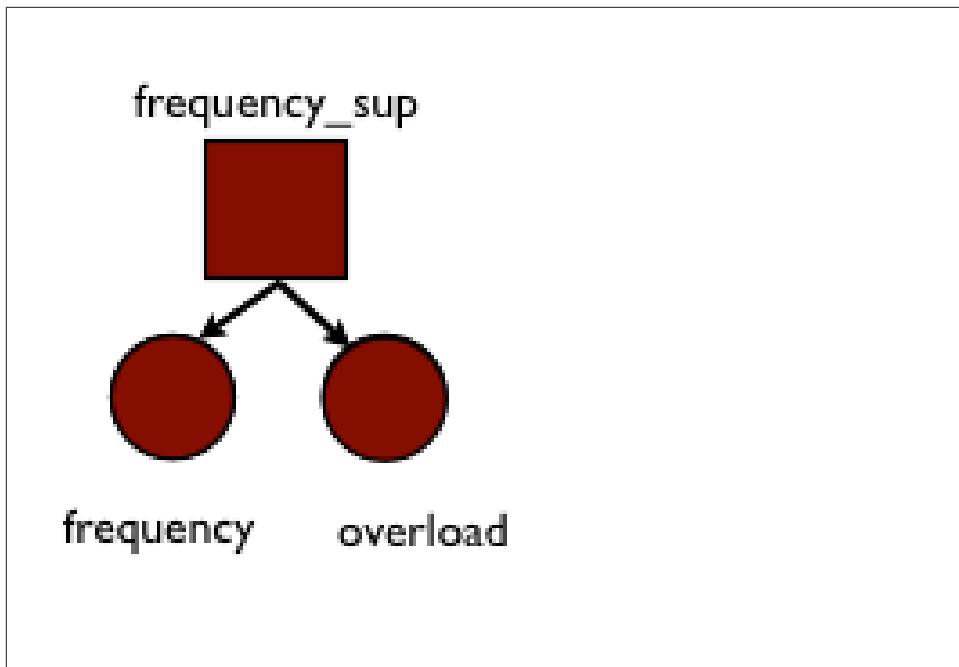


Figure 8-4. Frequency Server Supervision Tree

Let's look at the code for the frequency supervisor callback module. Like all other behaviours, you have to include the *behaviour* directive. You start the supervisor using the `start` or `start_link` functions, passing the optional supervisor name, the callback module, and arguments passed to `init/1`. As with event managers, there is no `Options` argument allowing you to set tracing, logging, or memory fine tuning.

```

-module(frequency_sup).
-behaviour(supervisor).

-export([start_link/0, init/1]).
-export([stop/0]).

start_link() ->
    supervisor:start_link({local,?MODULE},?MODULE, []).

stop() -> exit(whereis(?MODULE), shutdown).

init(_) ->
    ChildSpecList = [child(overload), child(frequency)],
    {ok,{rest_for_one, 2, 3600}, ChildSpecList}.

child(Module) ->
    {Module, {Module, start_link, []},
     permanent, 2000, worker, [Module]}.
  
```

In our example, the [] in the `start_link/3` call denotes the arguments sent to the `init/1` callback, not the Options. You cannot set sys options in supervisors at startup, but you can do so once the supervisor is started. Another difference with other behaviours is that supervisors do not expose built-in stop functionality to the developer. They are usually terminated by their supervisor or when the node itself is terminated. For those of you who do not want to write systems that never stop and insist on shutting down the supervisor from the shell, look at the `stop/0` function we've included; it simulates the shutdown procedure from a higher level supervisor.

Calling `start_link/3` results in invocation of the `init/1` callback function. This function returns a tuple in the format `{ok, SupervisorSpec}`, where `SupervisorSpec` is a tuple containing the supervisor configuration parameters and the child specification list ([Figure 8-5](#)). This OTP function is a bit more complicated than our pure Erlang example, because more is happening behind the scenes.

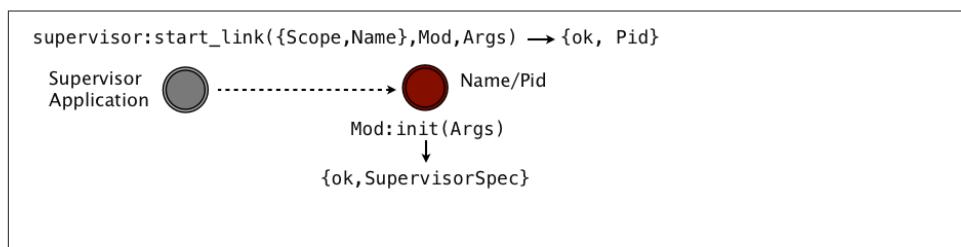


Figure 8-5. Generic Supervisors

In our example, the first element of the `SupervisorSpec` configuration parameter tuple tells the supervisor that if a child terminates, we want to terminate all children that were started after it before restarting them all. In general this element is called the restart strategy, and to obtain the desired restart approach we need for this case, we specify the `rest_for_one` strategy. Following the restart strategy, the numbers 2 and 3600 in the tuple tell the supervisor that it is allowed a maximum of two abnormal child terminations per hour (3600 seconds). If this number is exceeded, the supervisor terminates itself and its children, and sends an exit signal to all the processes in its link set with reason `shutdown`. So if this supervisor were part of a larger supervision tree, the supervisor monitoring it would receive the exit signal and take appropriate action.

The second element in the `SupervisorSpec` configuration parameter tuple is the child specification list. Each item in the list is a tuple specifying details for how to start and manage the static child processes. In our example, the first element in the tuple is a unique identifier within the supervisor in which it is started. Following that is the `{Module,Function,Arguments}` tuple indicating the function to start and link the worker to the supervisor , which is normally expected to return `{ok,Pid}`. Next, we find the

restart directive; the atom `permanent` specifies that when the supervisor is restarting workers, this worker should always be restarted.

Following the restart directive is the shutdown directive, specified here as 2000 ms. It tells the supervisor to wait 2000ms for the child to shut down (including the time spent in the `terminate` function) after sending the EXIT signal. There is no guarantee that `terminate` is called, as the child might be busy serving other requests and never reaches the EXIT signal in its mailbox.

Following that, the `worker` atom indicates that the child type is a worker as opposed to another supervisor, and finally the single-element list `[Module]` specifies the callback module implementing the worker.

```
supervisor:start_link(NameScope, Mod, Args)
supervisor:start_link(Mod, Args) -> {ok, Pid}
                                         {error, Error}
                                         ignore

Mod:init/1 -> {ok, [{RestartStrategy, MaxR, MaxT}, [ChildSpec]}]
                  ignore
```

Supervisors, just like all other behaviours, can be registered or referenced using their pid. If registering the supervisor, valid values to `NameScope` include `{local, Name}` and `{global, Name}`. You can also use the name registry represented in the `{via, Module, Name}` tuple, where `Module` exports the same API defined in the global name registry.

The `init/1` callback function normally returns the whole tuple comprising the restart tuple and a list of child specifications. But if it instead returns `ignore`, the supervisor terminates with reason *normal*. Note how supervisors do not export `start/2,3` functions, forcing you to link to the parent. In the next section, we'll look at all the available options and restart strategies in more detail. We refer to these options and strategies as the *supervisor specification*.

The Supervisor Specification

The supervisor specification is a tuple containing two elements ([Figure 8-6](#)):

- The non-generic information about the restart strategy for that particular supervisor
- The child specifications relevant to all static workers the supervisor starts and manages.

Let's look at these values in more detail, starting with the restart tuple.

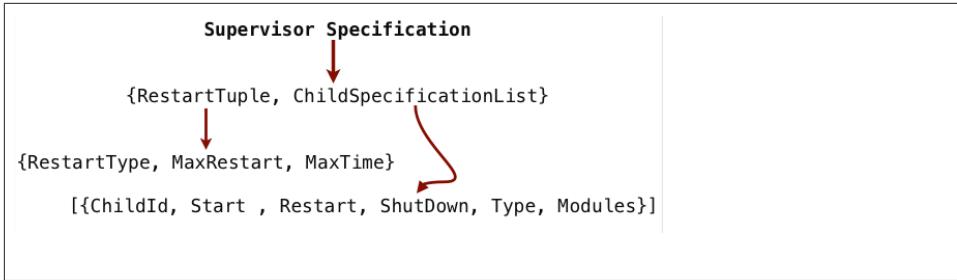


Figure 8-6. Supervisor specification

The restart tuple

The restart tuple `{RestartType, MaxRestart, MaxTime}` specifies what the supervisor does to the other children in the child specification list in case a child terminates abnormally. By “child” we mean either a worker or another supervisor. There are four different restart types; `one_for_one`, `one_for_all`, `rest_for_one` and `simple_one_for_one`.

Under `one_for_one` strategy (Figure 8-7), only the crashed process is restarted. This strategy is ideal if the workers don’t depend on each other and the termination of one will not affect the others. Imagine a supervisor monitoring the worker processes that control the instant messaging sessions of hundreds of thousands of users. If any of these processes crash, the crash will affect only the user whose session is controlled by the process. All other workers should continue running independently of each other, possibly receiving a status down update or a timeout.

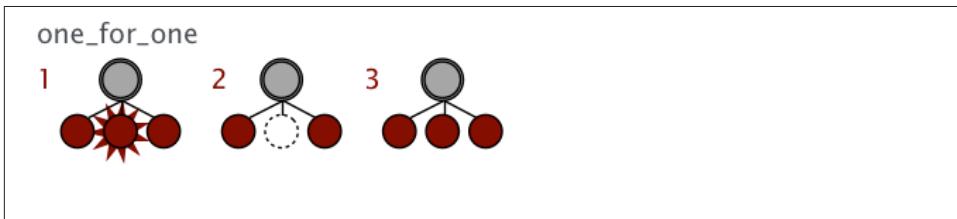


Figure 8-7. One for one

Under the `one_for_all` strategy shown in Figure 8-8, if a process terminates, all processes are terminated and restarted. This strategy is used if all or most of the processes depend on each other. Picture a very complex finite state machine handling a protocol stack. To simplify the design, the machine has been split into separate finite state machines that communicate with each other asynchronously, and these workers all depend on each other. If one terminates, the others would have to be terminated as well. For these cases, pick the `one_for_all` strategy.

one_for_all



Figure 8-8. One For All

Under the *rest_for_one* strategy (Figure 8-9), all processes started *after* the crashed process are terminated and restarted. Use this strategy if you start the processes in order of dependency. In our `frequency_sup` example, we first start the overload event manager, followed by the frequency allocator. The frequency allocator sends requests to the overload event manager whenever it runs out of frequencies. So if the overload manager has crashed and is being restarted, there is a risk the frequency server might send it requests that get lost. Under such circumstances, we want to first terminate the frequency allocator, and restart the overload manager and the frequency allocator in that order.

rest_for_one

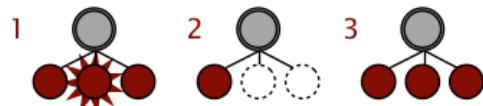


Figure 8-9. Rest for one

If losing the alarms sent to the frequency allocator did not matter (as the requests were asynchronous), we could have used a one-for-one strategy. Or we could have taken it a step further by making the raising and clearing of the alarms to the overload manager synchronous. In this case, if the overload manager had crashed and was being restarted, the frequency allocator would have also been terminated only when trying to make a synchronous call to it. Had the frequency allocator not run out of frequencies, thus not needing to raise or clear alarms, it could have continued functioning. As we have seen, there is no “one size fits all” solution; it all depends on the requirements you have and behaviour you want to give your system.

There is one last restart strategy to cover: *simple_one_for_one*. It is used for children of the same type added dynamically at runtime, not at startup. An example of when we would use this strategy is in a supervisor handling the processes controlling mobile phones that are added and removed from the supervision tree dynamically. We will cover dynamic children and the *simple_one_for_one* restart strategy later on in this chapter.

The last two elements in the restart tuple are `MaxRestart` and `MaxTime`. `MaxRestart` specifies the maximum number of restarts all child processes are allowed to do in `MaxTime` seconds. If maximum number of restarts is reached in this number of seconds, the supervisor itself terminates with reason *shutdown*, escalating the termination to its higher level supervisor. What is in effect happening is that we are giving the supervisor `MaxRestart` chances to solve the problem. If crashes still occur in `MaxTime` seconds, it means that a restart is not solving the problem, so the supervisor escalates the issue to its supervisor, who will hopefully be able to solve it.

Look at the supervision tree in [Figure 8-2](#). What if the phone FSMs under the phone supervisor are crashing because of corrupt data in the frequency handler? No matter how many times we restart them, they will continue to crash, because the problem lies in the frequency allocator, a worker we are doing nothing about. We solve cyclic restarts of this nature through escalation. If we allow the phone supervisor to terminate, the top supervisor will receive the exit signal and restart the frequency server and event manager workers before restarting the phone supervisor. Hopefully, the restart can clear the corrupt data, allowing the phone FSMs to function as expected.

The key to using supervisors is to ensure you have properly designed your start order and the restart strategy associated with it. Though you will never be able to fully predict what will cause your processes to terminate abnormally, you can nevertheless try to design your restart strategy to recreate the process state from known-good sources. Instead of storing the state persistently and assuming it is uncorrupted such that it reading it after a crash will correctly restore it, retrieve the various elements that created your state from their original source.

For example, if the corrupted data causing your worker to crash was the result of a transient transmission error, rereading it might solve the problem. The supervisor would restart the worker, which in turn would successfully reread the transmission and continue operating. And since the system would have logged the crash, the developer could look into its cause, modify the code to handle it appropriately, and prepare and deploy a new release to ensure that future similar transmission errors do not negatively impact the system.

In other cases, recovery might not be as straightforward. More difficult transmission errors might cause repeated worker crashes, in turn causing the supervisor to restart the worker. But since the restarts do not correct the problem, the client supervisor eventually reaches the restart threshold and terminates itself. This in turn affects the top-level supervisor, which eventually reaches its own restart threshold, and by terminating itself it takes the entire virtual machine down with it. When the virtual machine terminates, *heart*, a monitoring mechanism we cover in [Chapter 11](#) detects that the node is down and invokes a shell script. The recovery actions in this script could be as simple as restarting the Erlang VM or as drastic as rebooting the computer. Rebooting might reset the link to the hardware that is suffering from transmission problems and solve

the problem. If it doesn't, after a few reboot attempts, the script might decide not to try again, and instead alert an operator requesting manual intervention.

Hopefully, a load balancer will already have kicked in forwarding the requests to redundant hardware, providing seamless service to the end users. If not, this is when you receive a call in the middle of the night from a panicking first line support engineer informing you there is an outage. In either case, the crash is logged, hopefully with enough data to allow you to investigate and solve the bug: namely, ensuring that data is checked before being introduced into your system so that data corrupted by transmission errors is not allowed in the first place. We will look at distributed architectures and fault tolerance in [Chapter to Come]. For now, let's stay focused on recovery of a single node. Next in line are child specifications.

The child specification list

The child specification contains all of the information the supervisor needs to start, stop, and delete its child processes. The specification is a tuple of the format:

```
{Name,StartFunction,RestartType,ShutdownTime,ProcessType,Modules}
```

The elements of the tuple are:

Name

Any valid Erlang term, used to identify the child. It has to be unique within a supervisor, but can be reused across supervisors within the same node.

StartFunction

A tuple of the format {Module, Function, Args} which, directly or indirectly, calls one of the behaviour `start_link` functions. Supervisors can start only OTP-compliant behaviours, and it is their responsibility to ensure that the behaviours can be linked to the supervisor process. You cannot link regular Erlang processes to a supervision tree, because they do not handle the system calls.

RestartType

Tells the supervisor how to react to a child's termination. Setting it to `permanent` ensures that the child is always restarted, irrespective of whether its termination is normal or abnormal. Setting it to `transient` restarts a child only after abnormal termination. If you never want to restart a child after termination, set `Restart Type` to `temporary`.

ShutdownTime

`ShutdownTime` is a positive integer denoting the time in milliseconds, or the atom `infinity`. It is the maximum time allowed to pass between the supervisor issuing the `EXIT` signal and the `terminate` callback function completing returning. If the child is overloaded and it takes longer, the supervisor steps in and unconditionally terminates the child process. Note that `terminate` will be called only if the child process is trapping `EXITS`. If you are feeling grumpy or do not need the behaviour

to clean up after itself, you can instead specify `brutal_kill`, allowing the supervisor to unconditionally terminate the child using `exit(ChildPid, kill)`.

Choose your shutdown time with care, and never set it to `infinity` for workers, because it might cause the worker to hang in its `terminate` callback function. Imagine that your worker is trying to communicate with a defunct piece of hardware, the very reason for your system needing to be rebooted. You will never get a response, because that part of the system is down, stopping the system from restarting. If you have to, use an arbitrarily large number, which will eventually allow the supervisor to terminate the worker. For children that are supervisors themselves, on the other hand, it is common but not mandatory to select `infinity`, giving them the time they need to shut down their potentially large subtree.

ProcessType and Modules

These are used during a software upgrade to control how and which processes are being suspended during the upgrade. `ProcessType` is the atom `worker` or `supervisor`, while `Modules` is the list of modules implementing the behaviour. In the case of the frequency server, we would include `frequency`, whilst for our coffee machine we would specify `coffee_fsm`. If your behaviour includes library modules specific to the behaviour, include them if you are concerned that an upgrade of the behaviour module will be incompatible with one of library modules. For example, if you changed the API in the `hw` interface module as well as the `coffee_fsm` behaviour calling it, you would have to atomically upgrade both modules at the same time to ensure that `coffee_fsm` does not call the old version of `hw`. By listing both of these modules, you would be covered. But if you did not list `hw`, as in our example, you would have to ensure that any upgrade would be backward compatible and handle both the old and the new APIs. We will cover software upgrade in more detail in [Chapter to Come].

What if you don't know your `Modules` at compile time? Think of the event manager, which is started without any event handlers. When you do not know what will be running when you do a software upgrade, set `Modules` to the atom `dynamic`. When using dynamic modules, the supervisor will send a request to the behaviour module and retrieve the module names when it needs them.

Before looking at the interface and callback details, let's test our example with what we've learned. Looking at their child specifications, we see that both the overload event manager and the frequency server are permanent worker processes given two seconds to execute in their `terminate` function. We start the supervisor and its children, and see immediately that they have both started correctly. In shell command 4, we stop the frequency server, but because it is has its `RestartType` set to `permanent`, the supervisor will immediately restart it. We verify the restart in shell command 5 by retrieving the pid for the new frequency server process and noting that it differs from the pid of the

original server returned from shell command 2. In shell command 6 we explicitly kill the frequency server, and shell command 7 shows that once again, it restarted.

```
1> frequency_sup:start_link().
{ok,<0.34.0>}
2> whereis(frequency).
<0.36.0>
3> whereis(overload).
<0.35.0>
4> frequency:stop().
ok
5> whereis(frequency).
<0.40.0>
6> exit(whereis(frequency), kill).
true
7> whereis(frequency).
<0.43.0>
8> supervisor:which_children(frequency_sup).
[{frequency,<0.43.0>,worker,[frequency]},
 {overload,<0.35.0>,worker,[overload]}]
9> supervisor:count_children(frequency_sup).
[{specs,2},{active,2},{supervisors,0},{workers,2}]
```

In shell command 8, which_children/1 returns a tuple list containing the ChildId, its pid, worker or supervisor to denote its role, and the Modules list. Be careful when using this function if your supervisor has lots of children, because it will consume lots of memory. If you are calling the function from the shell, remember that the result will be stored in the shell history and not be garbage collected until the history is cleared.

```
supervisor:which_children(SupRef) -> [{Id, Child, Type, Modules}]
supervisor:count_children(SupRef) -> [{specs, SpecCount},
                                         {active, ActiveProcessCount},
                                         {supervisors, ChildSupervisorCount},
                                         {workers, ChildWorkerCount}]
supervisor:check_childspecs(ChildSpecs) -> ok
                                              {error, Reason}
```

The function count_children/1 returns a property list covering the supervisor's child specifications and managed processes. The elements are:

specs

The total number of children, both those that are active and those that are not.

active

The number of actively running children.

workers and supervisors

The number of children of the respective type.

And finally, check_childspecs/1 is useful when developing and troubleshooting child specifications and startup issues. It validates a list of child specifications, returning an error if any are incorrect or the atom ok if it finds no problems.

Supervisor specifications are easy to write. And as a result, they are also easy to get wrong. Too often, programmers pick configuration values which do not reflect the reality and conditions under which the application is running, or copy specifications from other applications, or even worse, use the default values skeleton templates that different editors provide. You must take care to get your supervision structure right when designing your start and restart strategy, and must build in fault tolerance and redundancy. The tasks include starting your processes in order of dependency, and setting restart thresholds that will propagate problems to supervisors higher up in the hierarchy and allow them to take control if supervisors lower down in the supervision tree cannot solve the issue.

Dynamic Children

Having gone through the supervisor specification returned by the `init/1` callback function, you must have come to the realization that the only child type we have dealt with so far are static children started along with the supervisor. But another approach is viable as well: dynamically creating the child specification list in our `init/1` call when starting the supervisor. For instance, we could inspect the number of active mobile devices and start a worker for each of them. We have already handled the end of the worker's lifestyle (by making the worker transient, so that if the phone is shut off, the worker is terminated) but we don't yet have similar flexibility for the start of the life cycle. What if a mobile device attaches itself to the network after we have started the supervisor?

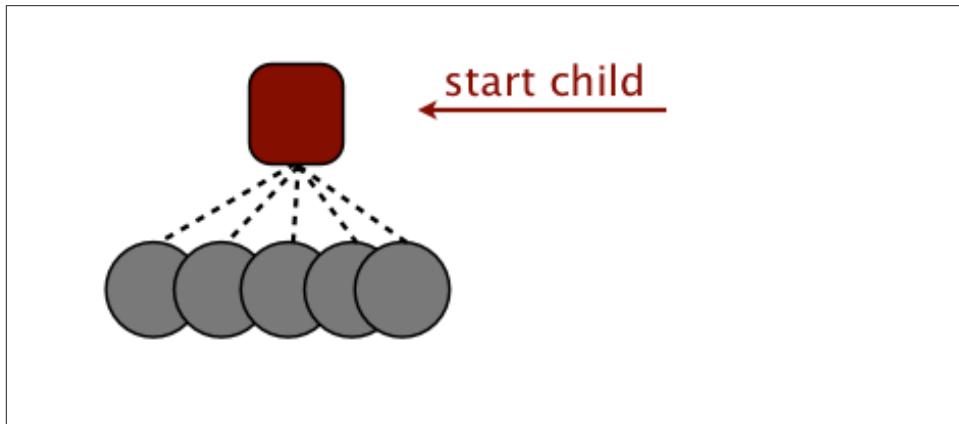


Figure 8-10. Dynamic Children

The solution to the problem is dynamic children, represented in the **Figure 8-10** figure. Let's start an empty supervisor whose sole responsibility will be that of dynamically starting and monitoring the finite state machine processes controlling mobile devices.

The FSM we'll be using is the one described but left as an exercise in “[The Phone Controllers](#)” on page 133. If you have not solved them, download the code from the book's code repository. The code includes a phone simulator, *phone.erl*, which starts a specified number of mobile devices and lets them call each other. We'll make the phone supervisor a child of the frequency supervision tree.

```
-module(phone_sup).
-behaviour(supervisor).

-export([start_link/0, attach_phone/1]).
-export([init/1]).

start_link() ->
    supervisor:start_link({local, ?MODULE}, ?MODULE, []).

init([]) ->
    {ok, [{one_for_one, 10, 3600}, []]}.

attach_phone(Ms) ->
    case hlr:lookup_id(Ms) of
        {ok, _Pid} -> {error, attached};
        _NotAttached -> end.
    ChildSpecs
```

In the `init/1` supervisor callback function, we have set the maximum numbers of restarts to ten per hour, and because mobile devices run independently of each other, the one for one restart strategy will do. Note that since we intend to start all children dynamically, the return value from `init/1` includes an empty list of child specifications. Further down in the module is the `phone_sup:attach_phone/1` call, which, given a mobile device number `Ms`, checks whether the number is already registered on the network. If not, it creates a child specification and uses the `supervisor:start_child/2` call to start it.

Let's experiment with this code. In shell commands 1 through 3, we start the supervisors and initialize the home location register database, the `hlr` (covered in “[ETS: Erlang Term Storage](#)” on page 40). We start two phones in shell commands 4 and 5, providing simple phone numbers as arguments. In shell command 6, we make phone 2, controlled by process P2, start an outbound call to the phone with phone number 1. Debug printouts are turned on for both phone FSMs, allowing you to follow the interaction between the phone FSMs and the phone simulator, implemented in the *phone* module. Following the debug printouts, we can see that phone 2 starts an outbound call to phone 1. Phone 1 receives the inbound call and rejects it, terminating the call and making both phones return to idle. As the simulator is based on random responses, you might get a different result when running the code.

```
1> frequency_sup:start_link().
{ok,<0.105.0>}
2> phone_sup:start_link().
{ok,<0.109.0>}
3> hlr:new().
```

```

ok
4> {ok, P1} = phone_sup:attach_phone(1).
{ok,<0.112.0>}
5> {ok, P2} = phone_sup:attach_phone(2).
{ok,<0.114.0>}
6> phone_fsm:action({outbound,1}, P2).
*DBG* <0.114.0> got {'$gen_sync_all_state_event',
                     {<0.103.0>, #Ref<0.0.0.292>},
                     {outbound,1}} in state idle
*DBG* <0.114.0> sent ok to <0.103.0>
      and switched to state calling
*DBG* <0.112.0> got event {inbound,<0.114.0>} in state idle
*DBG* <0.112.0> switched to state receiving
ok
*DBG* <0.112.0> got event {action,reject} in state receiving
*DBG* <0.112.0> switched to state idle
*DBG* <0.114.0> got event {reject,<0.112.0>} in state calling
*DBG* <0.114.0> switched to state idle
7> supervisor:which_children(phone_sup).
[{2,<0.114.0>,worker,[phone_fsm]},
 {1,<0.112.0>,worker,[phone_fsm]}]
8> supervisor:terminate_child(phone_sup, 2).
ok
9> supervisor:which_children(phone_sup).
[{2,undefined,worker,[phone_fsm]},
 {1,<0.112.0>,worker,[phone_fsm]}]
10> supervisor:restart_child(phone_sup, 2).
{ok,<0.122.0>}
11> supervisor:delete_child(phone_sup, 2).
{error,running}
12> supervisor:terminate_child(phone_sup, 2).
ok
13> supervisor:delete_child(phone_sup, 2).
ok
14> supervisor:which_children(phone_sup).
[{1,<0.112.0>,worker,[phone_fsm]}]

```

Have a look at the other shell commands in our example. You will find functions used to start, stop, restart, and delete children from the child specification list, some of which we use in our *phone_sup* module. Note how we get the list of workers when calling *supervisor:which_children/1*. We terminate the child in shell command 8, and note in shell prompt 9 that it is still part of the child specification list but with the pid set to undefined. This means that the child specification still exists, but the process is not running. We can now restart the child using only the child Name in shell command 10.

If the child has been terminated, we are able to delete the specification from the supervisor list. Keep in mind that these function calls do not use pids, but only unique names identifying the child specifications. This is because children crash and are restarted, so their pids might change. Their unique names, however, will remain the same.

Once the supervisor has stored the child specification, we can restart it using its unique name. To remove it from the child specification list, we need to first terminate the child as shown in shell command 12, after which we call `supervisor:delete_child/2` in shell command 13. Looking at the child specifications in shell command 14, we see that the specification of phone 2 has been deleted.

Simple one for one

The `simple_one_for_one` restart strategy is used when there is only one child specification shared by all the processes under a single supervisor. Our phone supervisor example fits this description, so let's rewrite it using this strategy. In doing so, we have added the `detach_phone/1` function, which we explain later. Note how we have moved the `hrl:new()` call to the supervisor `init` function:

```
-module(simple_phone_sup).
-behaviour(supervisor).

-export([start_link/0, attach_phone/1, detach_phone/1]).
-export([init/1]).

start_link() ->
    supervisor:start_link({local, ?MODULE}, ?MODULE, []).

init([]) ->
    hrl:new(),
    {ok, [{simple_one_for_one, 10, 3600}, {[{ms, {phone_fsm, start_link, []}}, transient, 2000, worker, [phone_fsm]}]}]}.

attach_phone(Ms) ->
    case hrl:lookup_id(Ms) of
        {ok, _Pid} -> {error, attached};
        _NotAttached -> supervisor:restart_child(?MODULE, Pid);
    end.

detach_phone(Ms) ->
    case hrl:lookup_id(Ms) of
        {ok, Pid} -> supervisor:terminate_child(?MODULE, Pid);
        _NotAttached -> supervisor:delete_child(?MODULE, Pid);
    end.
```

If you have looked at the code in detail, you might have spotted a few differences between the `simple_one_for_one` restart strategy and the one we used earlier for dynamic children. The first change is the arguments passed when starting the children. In the supervisor's `init/1` callback function, the `{phone_fsm, start_link, ChildSpecArgs}` in the child specification specifies no arguments (`ChildSpecArgs` is `[]`), whereas the function `phone_fsm:start_link(Args)` in the earlier example takes one, `Ms`. As the children are dynamic, they are started via the `supervisor:start_child(SupRef, StartArgs)` function. This function takes its second parameter, which it expects to be a list of terms, appends that list to the list of arguments in the child specification and calls `apply(Module, Function, ChildSpecArgs ++ StartArgs)`.

For the phone FSM, `ChildSpecArgs` in the child specification is empty, so the result of passing `[Ms]` as the second argument (`StartArgs`) to `supervisor:start_child/2` is that it calls `phone_fsm:start_link(Ms)`. It is also worth noting that we are initializing the ets tables using the `hrl:new()` call in the `init/1` callback, making the supervisor the owner of the tables.

The second difference is that in the simple one for one strategy, you do not use the child name to reference it, you use its pid. If you study the `detach_phone/1` function, you will notice this. You will also notice in the code that we are terminating the child without deleting it from the child specification list. We don't have to, as it gets deleted automatically when terminated. As a result, the functions `supervisor:restart_child/1` and `supervisor:delete_child/1` are not allowed. Only `supervisor:terminate_child/2` will work. Testing the `simple_one_for_one` supervisor leaves no surprises:

```
1> frequency_sup:start_link().
{ok,<0.34.0>}
3> simple_phone_sup:start_link().
{ok,<0.38.0>}
4> simple_phone_sup:attach_phone(1), simple_phone_sup:attach_phone(2).
{ok,<0.43.0>}
5> simple_phone_sup:attach_phone(3).
{ok,<0.45.0>}
6> simple_phone_sup:detach_phone(3).
ok
7> supervisor:which_children(simple_phone_sup).
[{undefined,<0.42.0>,worker,[phone_fsm]},
 {undefined,<0.43.0>,worker,[phone_fsm]}]
```

Once we've detached the phone, it does not appear among the supervisor children. This is specific to the simple one for one strategy, because with the other strategies, you need to both terminate and delete the children. Another difference is during shutdown. As simple one for one supervisors often grow to have many children running independently of each other (often a child per concurrent request), when shutting down, they are terminated in no specific, often concurrently. This is acceptable, as determinism in these cases is irrelevant, and most probably not needed. Finally, `simple_one_for_one` supervisors are the ones which scale better with a large numbers of dynamic children, as they use a `dict` key-value dictionary library module to store the child specifications, unlike other supervisor types who use a list. So while other supervisors might be faster for small amounts of children, performance deteriorates quickly if the frequency at which dynamic children are started and terminated is high.

Keeping ETS tables alive

You will recall that ets tables are linked to the process that creates them. If that process terminates, normally or abnormally, the ets table is deleted. You could use the `heir`

option when creating the table or call the `ets:give_away/3` call in your `terminate` function. An easier solution, however, is to place your ets table not in its own process, but in a supervisor. Pick the supervisor that monitors the processes using the table, so if the supervisor is terminated, you are guaranteed that the processes using it have also terminated. This approach requires the table to have public access so that non-owning processes can both read and write to it. In our example, we have placed our ets tables mapping pids to numbers and numbers to pids there. If the supervisor is terminated or shuts down, so will all of the processes accessing the table. The primary drawback to this approach is that if the data in the ets table gets corrupted, you need to restart the supervisor to clear it. Keep this in mind if you use this approach.

This is quite a bit of information to absorb. Before going ahead, let's review the functional API used to manage dynamic children. Keep in mind that `terminate_child/2`, `restart_child/2` and `delete_child/2` cannot be used with simple one for one strategies.

```
supervisor:start_child(Name, ChildSpecOrArgs)  -> {ok, Pid}
                                                {ok, Pid, Info}
                                                {error, already_started |
                                                 {already_present,Id} |
                                                 Reason}

supervisor:terminate_child(Name, Id)    -> ok
                                                {error, not_found | simple_one_for_one}

supervisor:restart_child(Name, Id)   -> -> {ok, Pid}
                                                {ok, Pid, Info}
                                                {error, running | restarting |
                                                 not_found | simple_one_for_one}

supervisor:delete_child(Name, Id)  -> -> ok
                                                {error, running | restarting |
                                                 not_found | simple_one_for_one |
                                                 Reason}
```

Gluing it all together

Before wrapping up this example, let's create the top level supervisor, `bsc_sup`, which starts both the `frequency_sup` and the `simple_phone_sup` functions. We will test the system using the `phone.erl` phone test simulator, which lets us specify the number of phones and the number of calls each phone should attempt, and then makes random calls, replying to and rejecting calls.

```
-module(bsc_sup).
-behaviour(supervisor).

-export([start_link/0, init/1]).
-export([stop/0]).

start_link() ->
    {ok, Pid} = supervisor:start_link({local,?MODULE}, ?MODULE, []).
```

```

stop() -> exit(whereis(?MODULE), shutdown).

init(_) ->
    ChildSpecList = [child(overload), child(frequency),
                     child(simple_phone_sup)],
    {ok, {{rest_for_one, 2, 3600}, ChildSpecList}}.

child(Module) ->
    {Module, {Module, start_link, []},
     permanent, 2000, worker, [Module]}.

```

We pick the rest for one strategy, because if the phones or the phone supervisor terminates, we do not want to affect the frequency allocator and overload handler. But if the frequency allocator or the overload handler terminates, we want to restart all of the phone FSMs. We allow a maximum of two restarts per hour, after which we escalate the problem to whomever is responsible for the `bsc` supervisor.

Suppose that corrupted data in the frequency server is causing the phone FSMs to crash. After the `simple_phone_sup` has terminated three times within an hour, thus surpassing its maximum restart threshold, `bsc_sup` will terminate all of its children, bringing the frequency server down with it. The restart will have hopefully cleared the problem allowing the phones to function normally. We will see how this escalation is handled in the upcoming chapters. Until then, let's use our `phone.erl` simulator and test our supervision structure and phone finite state machines by starting 150 phones, each attempting to make 500 calls.

```

1> bsc_sup:start_link().
{ok,<0.34.0>}
2> phone:start_test(150, 500).
*DBG* <0.106.0> got {'$gen_sync_all_state_event',
                      {<0.32.0>,#Ref<0.0.0.40>},
                      {outbound,109}} in state idle
...<snip>...
*DBG* <0.158.0> switched to state idle
3> counters:get_counters(overload).
{counters,[{{set_alarm,[no_frequency,<0.36.0>]},4},
           {{event,{frequency_denied,<0.36.0>}},29},
           {{clear_alarm,no_frequency},4}]}

```

For the sake of brevity, we've cut out all but one of the debug printouts. Having run the test, we retrieve the counters and see that during the trial run, we ran out of available frequencies four times, raising and eventually clearing the alarm accordingly. During these four intervals, 29 phone calls could not be set up as a result. Examining the logs, we can get the timestamps when these calls were rejected. If a pattern emerges, we can use the information to improve the availability of frequencies at various hours.

Before moving on to the next section, if you ran the above test in your computer and still have the shell open, try killing the frequency server three times using a `exit(whereis(frequency), kill)` statement. You will cause the top level supervisor to reach its maximum restart threshold and terminate. Note how, when the phone FSMs detaches itself in the `terminate` function, you get a `badarg` error as a result of the `hlr_ets` tables no longer being present. The error reports originate in the `terminate` function if the supervisor has terminated before the phone FSM, taking the `ets` table with it. These error reports might shadow more important errors, so it is always a good idea within a `terminate` function to embed calls that might fail within a `try-catch` and, by default, return the atom `ok`.

Non OTP-Compliant Processes

Child processes linked to an OTP supervision tree have to be OTP behaviours or follow the behaviour principles and be able to handle and react to OTP system messages. There are, however, times when we want to bypass behaviours and use pure processes, either because of performance reasons or simply as a result of legacy code. We get around this problem using supervisor bridges, implementing our own behaviours, or having a worker spawn and link itself to regular Erlang processes.

Supervisor bridges

In the mid-1990s when major projects for the next generation of Telecom infrastructure of that time were started at Ericsson, OTP was being implemented. The first releases of these systems, while following many of the design principles, were not OTP-compliant because OTP did not exist. With OTP R1 released, we ended up spending more time in meetings discussing whether we should migrate these systems to OTP than what it would actually have taken to do the job. It is at times like this, when no progress is made, that the `supervisor_bridge` behaviour comes in handy.

The supervisor bridge is a behaviour that allows you to connect a non-OTP compliant set of processes to a supervision tree. It behaves like a supervisor towards its supervisor, but interacts with its child processes using predefined start and stop functions ([Figure 8-11](#)).

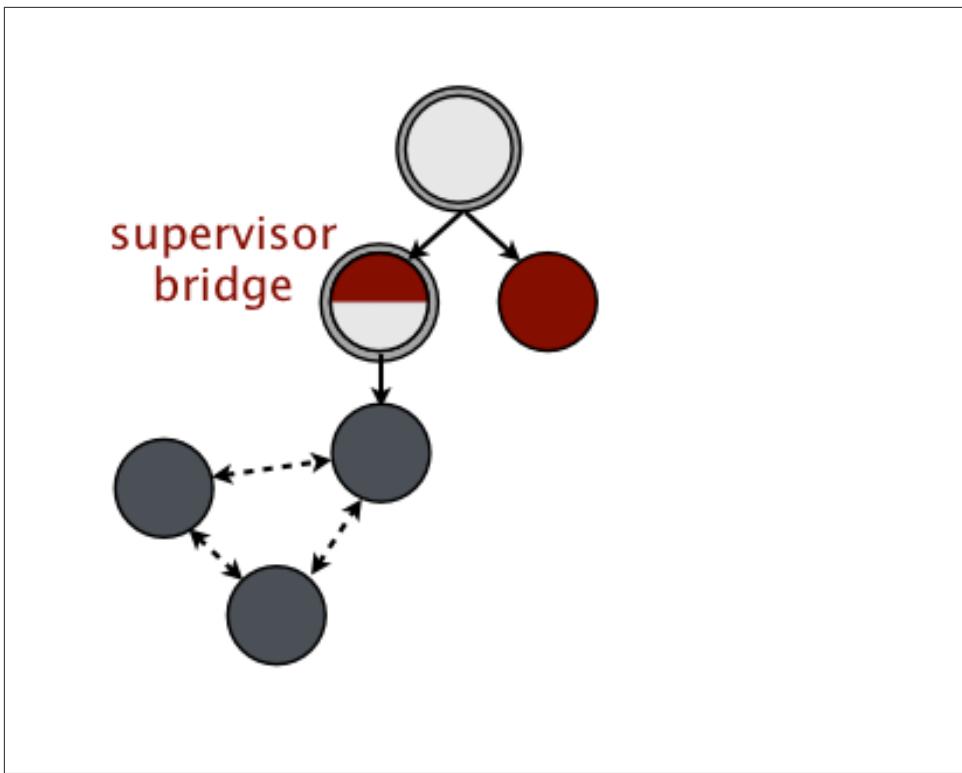


Figure 8-11. Supervisor Bridges

Start a supervisor bridge using the `supervisor_bridge:start_link/2,3` call, passing the optional `Name`, the callback `Module`, and the `Args`. This results in calling the `init(Args)` callback function, in which you start your Erlang process subtree, ensuring all processes are linked to each other. The `init/1` callback, if successful, has to return `{ok, Pid, State}`. Save the `State` and pass it as a second argument to the `terminate/2` callback.

If `Pid` terminates, the supervisor bridge will terminate with the same reason, causing the `terminate/2` callback function to be invoked. In `terminate/2`, all calls required to shut down the non-OTP compliant processes have to be called. At this point, the supervisor bridge's supervisor takes over and manages the restart. If the supervisor bridge receives a shutdown message from its supervisor, `terminate/2` is also called. While the supervisor bridge handles all of the debug options in the `sys` module, the processes it starts and is connected to have no code upgrade and debug functionality. Supervision will be limited to what has been implemented in the subtree.

```
supervisor_bridge:start_link(NameScope, Mod, Args) -> {ok, Pid} |
                                                ignore |
{error,{already_started,Pid}}
```

```
Mod:init(Args)          -> {ok,Pid,State} | ignore | {error,Reason}
Mod:terminate(Reason, State) -> term()
```

Adding non OTP Compliant Processes

Remember that supervisors can only take OTP compliant processes as part of their supervision tree. They include workers, supervisors and supervisor bridges. There is one last group, however, which can be added. They are processes which follow a subset of the OTP design principles; the same ones standard behaviours follow. We call processes which follow OTP principles but are not part of the standard behaviours *special processes*. You can implement your own special processes by using the *proc_lib* module to start your processes and handle system messages in the *sys* module. With little effort, the *sys*, *debug*, and *stats* options can be added. Processes implemented following these principles can be connected to the supervision tree. We will be covering them in more detail in [Chapter 10](#).

Scalability and Short-Lived Processes

Typical Erlang design creates one process for each truly concurrent activity in your system. If your system is a database, you will want to spawn a process for every query, insert, or delete operation. But don't get carried away. Your concurrency model will depend on the resources in your system, as in practice, you could have only one connection to the database. This becomes your bottleneck, as it ends up serialising your requests. In this case, is sending this process a message easier than spawning a new one? If your system is an instant messaging server, you will want a process for every inbound and outbound message, status update, or login and logout operation. We are talking about tens or possibly hundreds of thousands of simultaneous processes that are short-lived and reside under the same supervisor. At the time of writing, supervisors that have a large number of dynamic children starting and terminating at very short continuous intervals will not scale well because the supervisor becomes the bottleneck. The implementation of the *simple_one_for_one* strategy scales better, as unlike other supervisor types which store their child specifications in lists, it uses the dict key-value library module. But despite this, it will also have its limits. Giving a rule-of-thumb measure of the rate at which dynamic children can be started and terminated is hard, because it depends on the underlying hardware, OS, and cores, as well as the behaviour of the process themselves (including the amount of data which needs to be copied when spawning the process). These issues are rare, but if a supervisor message queue starts growing to thousands of messages, you know you are affected. There are two approaches to the problem.

The clean approach, shown in [Figure 8-12](#), is to create a pool of supervisors, ensuring that each does not need to cater for more children than it can handle. This is a recommended strategy if the children have to interact with other processes and are often long

lived. The process on the left is the dispatcher, which manages coordination among the supervisors, and if necessary, starts new ones. You can pick a supervisor in the pool using an algorithm which best suits your needs. It could be (but not limited to) round robin, consistent hashing or at random.

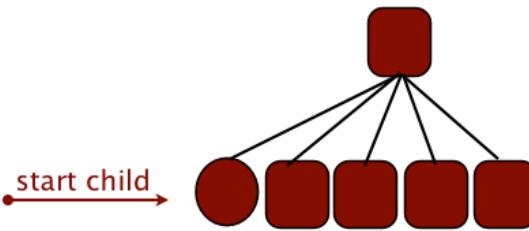


Figure 8-12. Supervisor Pools

The second approach taken by many is to have a worker, more often than not a generic server, that `spawn_links` a non-OTP compliant process for every request (Figure 8-13). You will often find this strategy in messaging servers, web servers, and databases. This non-OTP compliant process usually executes a sequential, synchronous set of operations and terminates as soon as it has completed its task. This solution potentially sacrifices OTP principles for speed and scalability, but it ensures that your process is linked to the behaviour that spawned it. So if the process tree shuts down, the linked processes will also terminate.

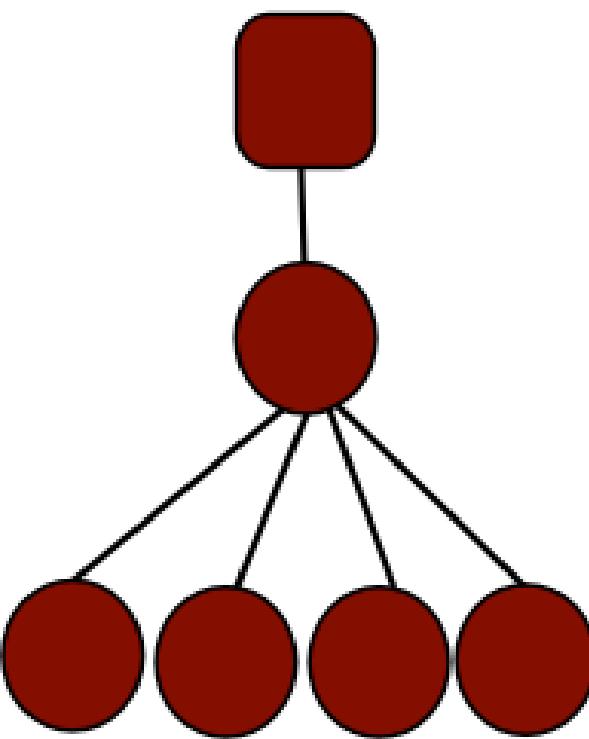


Figure 8-13. Linking to a worker

Why link? Don't forget that your system will run for years without being restarted. You can't predict what upgrades, new functionality, or even abnormal terminations will occur. The last thing you want is a set of dangling processes you can't control, left there after the last failed upgrade. Because you link the non-OTP-compliant children to their parent, if the parent terminates, so do the children.

Multiple Supervision Policies

Every child may be associated with one supervisor or parent. OTP supervision trees are not set up to handle cases where a behaviour might belong to two process groups with different policies. If you come across such use cases where it might make sense to have

multiple processes monitor the same behaviour, use links and monitors, and ensure that only one of the behaviours is responsible for handling the restart strategy.

Synchronous Starts for Determinism

Remember that when you start behaviours with either the `start` or `start_link` calls, process creation and the execution of the `init/1` function are synchronous. The functions return only when the `init/1` callback function returns. The same applies to the supervisor behaviour. A crash during the start of any behaviour will cause the supervisor to fail, terminating all the children it has already started. Because starts are synchronous, try to minimize the amount of work done in the `init/1` callback. A trick you can use to postpone your initialization is to set the timeout to 0 in your `init/1` behaviour callback function. Setting a timeout in this manner results in your callback module receiving a `timeout` message immediately after `init/1` returns, allowing you to asynchronously continue initializing your behaviour while the supervisor proceeds with starting other behaviours. A more general alternative is for `init/1` to send a suitable message to `self()`, which can be handled after `init/1` returns in order to asynchronously proceed with initialization.

Repairing Mnesia Tables

Remember Mnesia, the distributed database introduced in the *Erlang Programming* book? An unexpected restart issue we had many times in live systems was for Mnesia to load and fix its tables during a restart. This caused after a node is shut down abnormally as the result of a VM crashing or it being killed, a power outage or a hardware fault. Upon restarting, Mnesia loads its tables asynchronously, so as to not block other behaviours from starting. Fixing tables has been known to take a long time, as logs and backups are scanned. If you try to read a value from a table that has not been completely loaded, the call will raise an exception.

If a behaviour is dependent on a set of tables, you can get around this problem by calling `mnesia:wait_for_tables/2` when initializing your behaviour. This will work without any issues in a test environment when tables are small, but in production systems, the data being loaded is substantial. In fact, data sets in test environments are usually so small that you will probably get away without calling `wait_for_tables/2`. But in the worse case, in a live system after a major outage, can your supervisor startup handle waiting a couple of minutes for a mnesia table being repaired as the result of an abnormal termination? Will it cause message queues elsewhere or result in a knock-on effect? These are issues you have to validate when testing your system.

Why are synchronous starts important? Imagine first spawning all your child processes asynchronously and then checking that they have all started correctly. If something goes

wrong at startup, the issue might have been caused by the order in which processes were started or in the order the expressions in their respective `init` callbacks were executed. Recreating the race condition that resulted in the startup error might not be trivial. Your other option is to start a process, allow it to initialize, and start the next one only when the `init` function returns. This will give you the ability to reproduce the sequence that led to a startup error without having to worry about race conditions. Incidentally, this is the way we do it when using OTP, where the combination of application (covered in [Chapter 9](#)), supervisors, and the synchronous startup sequence together provide a “simple core” that guarantees a solid base for the rest your system.

Testing Your Supervision Strategy

In this chapter, we’ve explained how to architect your supervision tree, group and start processes based on dependencies, and ensure that you have picked the right restart strategy. These tasks should not be overlooked or underestimated. Although you are encouraged to avoid defensive programming and let your behaviour terminate if something unexpected happens, you need to make sure that you have isolated the error and are able to recover from this exception. You might have missed dependencies, picked the wrong restart strategy, or set your allowed number of restarts too high (or low) in a possibly incorrect time interval. How to you test these scenarios and detect these design anomalies?

Abnormal or Normal Termination?

I was involved in a project where each generic server managed by the supervisor owned a TCP/IP connection. When the socket was closed as a result of a connectivity error, it would terminate the behaviour abnormally, be restarted, and attempt to re-establish the connection. Each network connectivity error, although perfectly legitimate, would increment the counter for the number of abnormal terminations, occasionally resulting in shutting the node down. This was particularly evident when experiencing network connectivity issues as a result of a firewall misconfiguration, router and load balancer failures, or something as simple as a system administrator tripping over a network cable. On top of creating outages, other abnormal issues happening in the system where being lost in the sheer volume of error reports being generated. Because these actions can happen under normal operations, the socket closings that were not initiated by the program itself should have been handled as normal events and not resulted in abnormal termination.

All correctly written test specifications for Erlang systems will contain negative test cases where recovery scenarios and supervision strategies have to be validated by simulating abnormal terminations. You need to ensure that the system is not only able to start, but also to restart and self-heal when something unexpected happens.

In our first test system, `exit(Pid, Reason)` was used to kill specific processes and validate the recovery scenarios. In later years, we used **chaos monkey**, an open source tool that randomly kills processes, simulating abnormal errors. Try it while stress testing your system, complementing it with fault injections where hardware and network failures are being simulated. If your system comes out of it alive, it is on track to becoming production ready.



Don't tell the world you are killing children!

We were working on the R1 release of OTP when a group of us left the office and took the commuter train into Stockholm. We were talking about the ease of killing children, children dying, and us not having to worry about it, as supervisors would trap exits and restart them. We were very excited and vocal about this, as it was at the time a novel approach to software development, one we were learning about as we went along. We were all so engrossed into this conversation that we failed to notice the expression of horror on the faces of the old ladies sitting next to us until we started approaching our stop. I have never seen an expression of horror turn so quickly into an expression of relief when finally we got off the train. **Pro Tip:** When in public, talk about behaviours, not children, and do not kill them - terminate them instead. It will help you make friends and you won't risk you having to explain yourself to a law enforcement officer who probably has no sense of humor.

How Does This Compare?

How does the approach of non-defensive programming, letting supervisors handle errors, compare to conventional programming languages? The urban legend among us Erlang programmers boasted of less code and faster time to market. But the numbers we quoted were based on gut feel or studies which were not public. The very first study, in fact, came from Ericsson where a sizable amount of features in the MD110 corporate switch were rewritten from PLEX (a proprietary language used at the time) to Erlang. The result was a ten-fold decrease in code volume. Worried that no one would believe them, the official stance was that you could implement the same features with four times less code. Four was picked because it was big enough to be impressive, but small enough not to be challenged. We finally got a formal answer when Heriot-Watt University in Scotland ran a study focused on rewriting C++ production systems to Erlang/OTP. One of the systems was Motorola's Data Mobility (DM), a system handling digital communication streams for two-way radio systems used by emergency services. The DM had been implemented in C++ with fault tolerance and reliability in mind. It was rewritten in Erlang using different approaches, allowing the various versions to be compared and contrasted.

Many academic papers and talks have been written on this piece of research. One of the interesting discoveries was an 85% reduction in code in one of the Erlang implementations. This was in part explained by noting that 27% of the C++ code consisted of error handling and defensive programming. The counterpart in Erlang, if you assumed OTP to be part of the language libraries, was a mere 1%!

Just by using supervisors and using the fault tolerance built into OTP behaviours, you get a code reduction of 27% compared to other conventional languages. Remove the 11% of the C++ code that consists of memory management, remove another 23% consisting of high-level communication - all features that are part of the Erlang semantics or part of OTP - and include declarative semantics and pattern matching, and you can easily understand how an 85% code reduction becomes possible. Read these papers² and have a look at the recordings of the presentations available online if you want to learn more about this study.

Summing Up

Building on previous chapters that covered OTP worker processes, this chapter explained how to group them together in supervision trees. We have looked at dependencies and recovery strategies, and how they allow you to handle and isolate failure generically. The bottom line is for you not to try to handle software bugs or corrupt data in your code. Focus on the positive cases and, in case of unexpected ones, let your process terminate and have someone else deal with the problem. This strategy is what we refer to as *fail-safe*.

Here are the functions exported by the *supervisor* and *supervisor_bridge* behaviours, together with their respective callback functions. You can read more about them in their respective manual pages.

Table 8-2. gen_event callbacks

supervisor function or action	gen_event callback function
supervisor:start_link/2, supervisor:start_link/3	Module:init/1
supervisor_bridge:start_link/2, supervisor_bridge:start_link/3	Module:init/1, Module:terminate/2

Before reading on, you should also read through the code of the examples provided in this chapter and look for examples of supervisor implementations online. It will help you understand how to design your system keeping fault tolerance and recovery in mind.

2. The most comprehensive being Nyström, J.H., P.W. Trinder, and D.J. King. *Concurrency and Computation: Practice & Experience*, 20(8), 2008.

What's Next?

In the next chapter, we will be covering how to package supervision trees into a behaviour called an application. Applications contain a supervision tree and provide operations to start and stop it. They are seen as the basic building block of Erlang systems. And in [Chapter 11](#), we will look at how we group applications in a release, giving us an Erlang node.

CHAPTER 9

Applications

In our previous chapters, we've looked at worker behaviours and how they can be grouped together to form a supervision tree. In this chapter, we explore the application behaviour, which allows us to package together supervision trees, modules and other resources into one semi-independent unit, providing the basic building blocks of large Erlang systems. An OTP application is a convenient way to package code and configuration files and distribute the result around the world for others to use.

An Erlang node typically consists of a number of loosely coupled OTP applications that interact with each other. OTP applications come from a variety of sources:

- Some are available as part of the standard Ericsson distribution, including *Mnesia*, *SASL*, and *OS_Mon*.
- Other generic applications that are not part of the Ericsson distribution but are necessary for the functionality of many Erlang systems can be obtained commercially or as open source. Examples of generic applications include *alarm* for alarming, *folsom* or *exometer* for metrics, and *lager* for logging.
- Each node also has one or more non-generic applications that contain the system's business logic. These are often developed specifically for the system that contains the core of the functionality.
- A final category of OTP applications are those that are full user applications themselves which together with their dependencies could run on a stand alone basis in an Erlang node. The bundle of applications is referred to as a release. Examples include the *Yaws* web server, the *Riak* database, the *RabbitMQ* message broker or the *MongooseIM* chat server. While not a common practice, inter-application throughput and overall performance can sometimes be improved by running business logic applications together on the same node with these types of full applications.

Regardless of their sources, though, OTP applications are generally structured the same way. We explore the details of this structure in the remainder of this chapter. In the rest of the book, we use the term *application* to refer specifically to an OTP application, and not an application in the broader sense of the word.

How Applications Run

One way to view an application is as a means of packaging resources into reusable components. Resources can consist of modules, processes, registered names and configuration files. They could also include other non-Erlang source or executable code such as bash scripts, graphics, or drivers. Though different OTP applications contain different resources and perform different functions or services, to the Erlang runtime system they all look the same; it doesn't distinguish between them in terms of how it loads and runs them, allows them to be accessed and invoked from other applications, or terminates them. [Figure 9-1](#) shows how various components run together on the Erlang runtime.

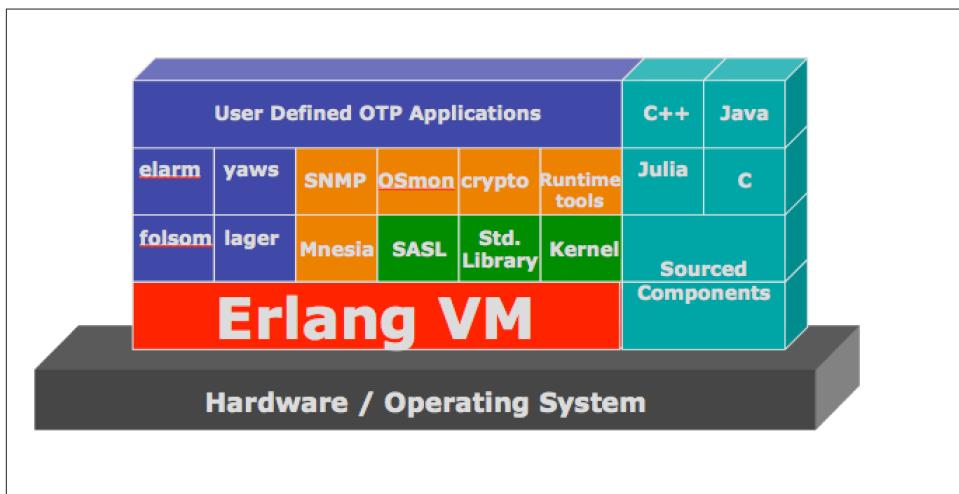


Figure 9-1. An Erlang Release

Applications can be configured, started, and stopped as a whole. This allows a system to easily manage many supervision trees, running them independently of each other. One application can also depend on another one; for example, a server-side web application might depend on a web server application such as Yaws. Supporting application dependencies means the runtime has to handle starting and stopping applications in the right order. This provides a basis for cleanly encapsulating functionality and encourages reusability in a way that goes far beyond that of modules.

There are two types of applications: *normal applications* and *library applications*. Normal applications start a top-level supervisor, which in turn starts its children forming the supervision tree. Library applications contain library modules but do not start a supervisor or processes themselves; the function calls they export are invoked by workers or supervisors running in a different application. A typical example of a library application is *stdlib*, which contains all of the OTP standard libraries: *supervisor*, *gen_event*, *gen_server*, and *gen_fsm*.

Behind the scenes in the Erlang VM, a process called the *application controller* starts on every node. For every OTP application, the controller starts a pair of processes called the *application master*. It is the master that starts and monitors the top-level supervisor and takes action if it terminates (Figure 9-2).

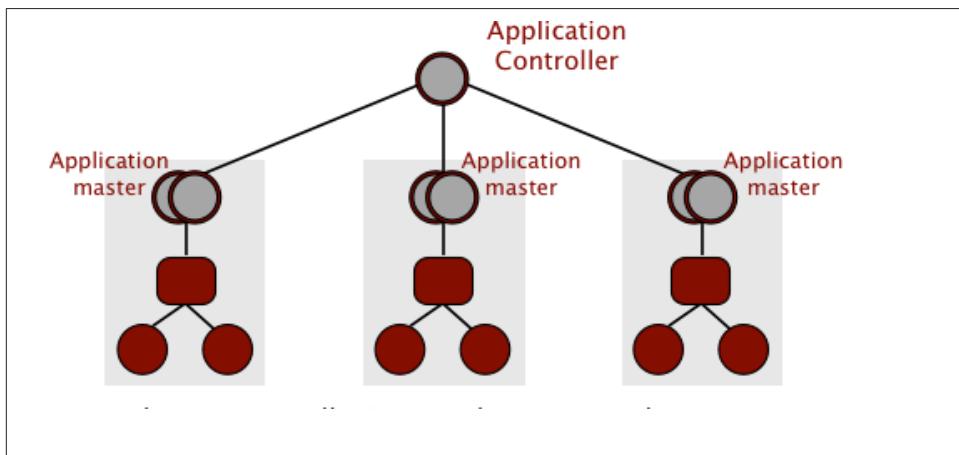


Figure 9-2. Application Controller

When using releases (covered in Chapter 11), the Erlang runtime treats each application as a single unit; it can be loaded, started, stopped, and unloaded as a whole. When loading an application, the runtime system loads all modules and checks all its resources. If a module is missing or corrupt, startup fails and the node is shut down. When starting an application, the master spawns the top-level supervisor, which in turn starts the remainder of the supervision tree. If any of the behaviours in the supervision tree fail at startup, the node is also shut down. When stopped, the application master terminates the top-level supervisor, propagating the shutdown exit signal to all behaviour processes in the supervision tree. Finally, when unloading an application, the runtime purges all modules for that application from the system. Now that we have a high level overview on how everything is glued together, let's start looking at the details.

The Application Structure

Applications are packaged in a directory that follows a special structure and naming convention. Tools depend on this structure, as do the release handling mechanisms. A typical application directory has the structure shown in [Figure 9-3](#), containing the *ebin*, *src*, *priv*, and *include* directories:

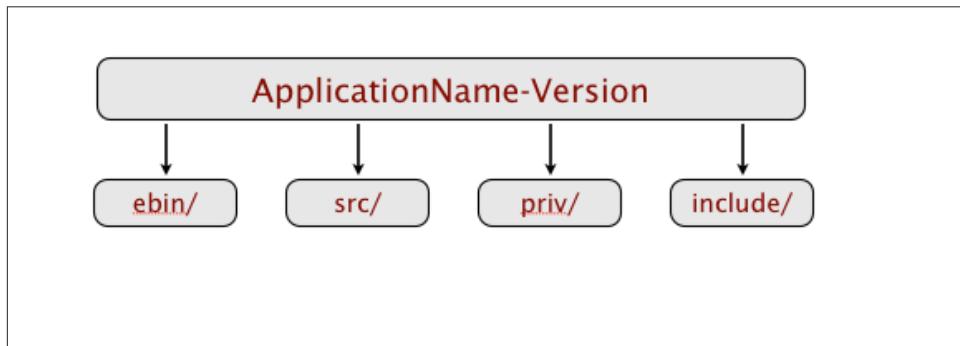


Figure 9-3. Application Structure

The name of the application directory is the name of the application followed by its version number. This allows you to store different versions of the application in the same library directory, using the code search path to point to the one being used. Sub-directories of an application include:

ebin

Contains the beam files and the application configuration file, also known as the *app* file.

src

Contains the Erlang source code files and include files that you do not want other applications to use.

priv

Contains non-Erlang files needed by the application, such as images, drivers, scripts or proprietary configuration files.

include

Contains exported include files that can be used by other applications.

Other non standard directories such as **doc** for documentation, **tests** for test cases and **examples** can also be part of your application. What sets them apart from the ones in the previous list is that runtime system and tools allow you to access the standard directories by application name, without having to reference the version. For instance, when you load an application, the code search path for that application will point straight

to the *ebin* directory of the version you are using. Or if you want to include the *hrl* file of another application, the include path in the makefiles will point to the correct version. You can not do this with non standard directories, and as such, you or your tools have to figure out the path.

Let's have a look at this structure in more detail by following an example in the OTP distribution. Remember that the directory structure of any OTP application in the Erlang distribution will be the same as the applications you are implementing in your system.

Go to the Erlang root directory, and from there, *cd* into the *lib* directory. If you are unsure where Erlang is installed, start a shell and determine the *lib* directory by typing `code:lib_dir()`. The *lib* directory contains all of the applications included when installing Erlang. If you have upgraded your release or installed patches, you might find more than one version of some applications. Let's have a look at the latest version of the runtime tools application *runtime_tools*, which should be included in every release.

```
1> code:lib_dir().
"/usr/local/lib/erlang/lib"
2> halt().
$ cd /usr/local/lib/erlang/lib
$ ls
...<snip>...
appmon-2.1.14.2      erts-5.7.5      public_key-0.18
asn1-1.6.13           erts-5.8.1      public_key-0.5
asn1-1.6.14.1         et-1.4        public_key-0.8
asn1-2.0.1            et-1.4.1       reltool-0.5.3
common_test-1.4.7     et-1.4.4.3    reltool-0.5.4
common_test-1.5.1     eunit-2.1.5    reltool-0.6.3
common_test-1.7.1     eunit-2.2.4    runtime_tools-1.8.10
compiler-4.6.5        gs-1.5.11     runtime_tools-1.8.3
compiler-4.7.1        gs-1.5.13     runtime_tools-1.8.4.1
...<snip>...
$ cd runtime_tools-1.8.10/
$ ls
doc      examples   info      src
ebin     include     priv
```

When in the directory, list its contents. The *doc* directory and *info* file are non-standard files or directories, and as such have nothing to do with OTP. The Ericsson OTP team uses them for documentation purposes. Erlang developers often add other application-specific directories and files as well, such as *test* and *examples* directories. If you for example look at later versions of the runtime tools application, you will see that the *info* file is no longer there.

Let's focus on the OTP standard directories. If you *cd* into the *ebin* directory and examine its contents, you will find *beam* files, an *app* file, and possibly an *appup* file. The beam files, as you likely already know, contain Erlang byte code. The app file is a mandatory application resource file we will explore in more detail in “[Application Resource Files](#)”

on page 205. The *appup* file might also be there if you have at some point upgraded your application. We cover this file in more detail in chapter [Chapter to Come] when looking at software upgrades.

The *src* directory contains the Erlang source code. If the modules in this directory use one or more *hrl* files that are not exported to be used by other applications, put them here. The current working directory is by default always included in the include search path, so when compiling, they will be picked up. It is the responsibility of your build system to ensure that beam files resulting from compilation are moved from the *src* to the *ebin* directory.

Macros and records defined in include files are often part of interface descriptions, requiring modules in other applications to have access to these definitions. The *include* directory is used in the build process to provide access to the *hrl* files stored in it. Without having to know the location of the include file directory or the application version, you can use the following directive:

```
-include_lib("Application/include/File.hrl").
```

where *Application* is the application name without the version, and *File.hrl* is the name of the include file (Remember that file paths are case sensitive). The compiler will know which version of the application you are working with, find the directory, and automatically include the file without you having to change the version numbers between releases. Even if include files do not require the *hrl* extension, it is good practice to always use it. Version dependencies are handled in release files, covered in chapter [Chapter to Come].

If you run `grep ^-include_lib ssl*/src/*.erl` from your Erlang lib directory to examine the *src* directories of all the versions of the *ssl* application installed on your system, you will notice that some of the modules include *hrl* files from other applications, such as *public_key* and *kernel*. There will also be a few include files stored directly in the *src* directory, which are used only by the *ssl* application itself.

The *priv* directory contains non-Erlang specific resources. They could be linked-in drivers, NIFs, executables, graphics, HTML pages, Javascript, or application-specific configuration files - basically, any source the application needs at runtime that is not directly Erlang-related resides here. In the case of the *runtime_tools* application, the *priv* directory includes source and object code of its trace drivers. Because the path of the *priv* directory will differ based on the version of the application you are running, use

```
code:priv_dir(Application).
```

in your code to generically find it when building a release.

The *ebin* and *priv* directories are usually the only ones shipped and deployed on target machines. This will probably answer your question as to why the mandatory *app* resource file is included in the *ebin* directory and not *src* or *priv*. If you look around other

applications shipped as part of the standard distribution, you will also notice that the *priv* directory is not mandatory if it is not used. The *sasl* application, for example, has no *priv* directory, and there are other such applications as well.

Although it is up to you whether you ship source code and documentation with your products, it is not a good idea to bundle them up with your release deployed on target machines, because once you've upgraded your beam files, no checks are made to ensure the source code is up to date. Once, when called in to resolve an outage, we were reading the code on the production machines until we realized it was the first release of the code, now woefully out of date as the sources had since been patched, rewritten, cleaned up and redeployed. After all, those who deployed the new beam files knew the source code on the target machines was not up to date. They also knew that they were not always the ones supporting the system, but assumed we would be using the source code repository, or that we would just ask. Should you find yourself in a similar predicament, follow our words of wisdom and always start with the assumption that those supporting the systems you have written and deployed are anti-social axe murderers who know where you live. They will not speak to you in the middle of the night when called to deal with an outage caused by a bug in your code, but might come knocking on your door at dawn once the system is operational again.

And while we have your attention, please, never ever ship the compiler application with production systems. If you do, you are really asking for trouble, because you will end up changing and compiling the code on target machines in an attempt to resolve the issue. Assuming it is the correct version of the code (which it probably isn't), and assuming it actually solves the problem (which it probably won't), there is still the risk you will forget to commit it to the repository. Don't forget all of this is happening at 3 AM, and all you want is to return to sleep. Code should be taken from the repository and tested in a test environment before deploying it to a live system. No matter how urgent the fix, don't cut corners, because you will risk paying the price later, irrespective of the time of day (or night).

The Callback Module

The application behaviour is no different than other OTP behaviours. The module containing the generic code, *application*, is part of the *kernel* library, and a callback module contains all of the specific code ([Figure 9-4](#)).

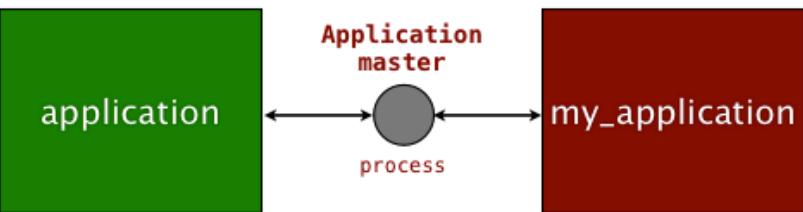


Figure 9-4. Application Behaviour

The *behaviour* directive must be included in the callback module, alongside the mandatory and optional callbacks. Of all behaviours, the application callback module is the simplest. Unless you are dealing with take-overs and fail-overs in distributed environments or complex startup strategies, expect your application callback module to require no more than a few simple lines of code.

Starting Applications

The callback module is invoked when starting your application. You start it by calling `application:start(Application)`, where `Application` is the application name. This call loads all of the modules which are bundled with the application and starts the master processes, one of which calls the `Mod:start(StartType, StartArgs)` callback function in the application callback module. The `start/2` function has to return `{ok, Pid}`, where `Pid` is the process identifier of the top-level supervisor. If the application is not already loaded, `application:load(Application)` is called prior to the starting the master processes. Our application callback module looks something like this:

```
-module(bsc).

-behaviour(application).

%% Application callbacks
-export([start/2, stop/1]).

start(_StartType, _StartArgs) ->
    bsc_sup:start_link().

stop(_Data) ->
    ok.
```

The first argument, `_StartType`, ignored by most applications, is usually the atom `normal`, but if we're running distributed applications with automated fail-over and take-over, it could take on the value `{takeover, Node}` or `{failover, Node}`. We will look at these values later in the chapter. The second argument, `_StartArgs`, comes from the

`mod` key of the application resource file, also described later in “Application Resource Files” on page 205.

The Figure 9-5 figure shows how the application callback module starts the top-level supervisor. The application callback module’s `start/2` function typically just calls the `start_link` function provided by the top-level supervisor. For example, the `bsc:start/2` function shown earlier simply calls `bsc_sup:start_link/0`.

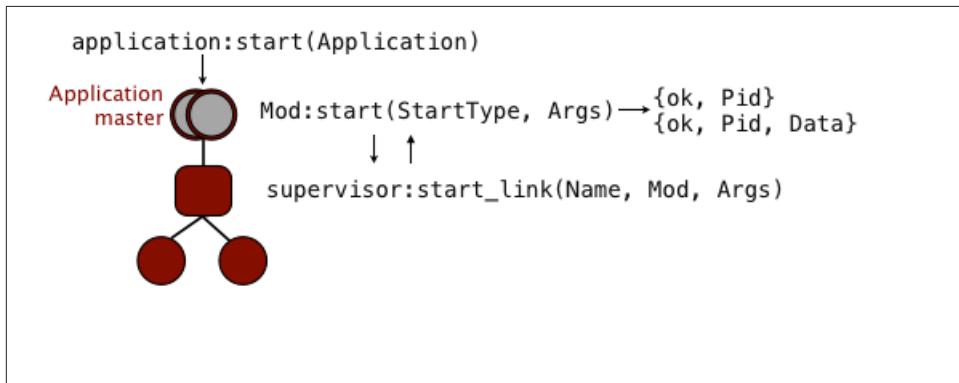


Figure 9-5. Starting Applications

In our case, `bsc_sup:start_link/0` returns `{ok, Pid}`, which is also what `bsc:start/2` returns. Another valid return value is `{ok, Pid, Data}`, where the contents of `Data` are stored and passed on to the `stop/1` callback function (Figure 9-6). If you do not return any `Data`, just ignore the argument passed to `stop/1` (in case you’re curious, it will be bound to `[]`).

Stopping Applications

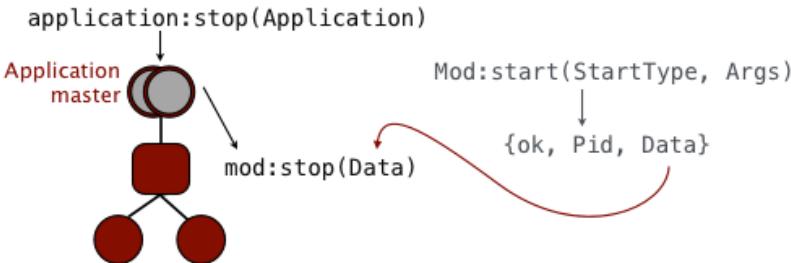


Figure 9-6. Stopping Applications

To stop an application, use `application:stop(Application)`. This results in the callback function `Mod:stop/1` being called after the supervision tree has been terminated, including all workers and supervisors. `Mod:prep_stop/1` is an equally important, but optional callback invoked before the processes are terminated. If you need to clean anything up before terminating your supervision tree, `prep_stop/1` is where you trigger it.

Let's try loading, starting, and stopping an application from the standard OTP distribution. Depending on how you installed Erlang, it might or might not be started automatically when you start the shell. You can find out by typing `application:which_applications()`. In the example, we do this in the first prompt of the shell command, getting back a list of tuples. The first element is the application name, the second a descriptive string,¹ and the third is a string denoting the application version. When you start Erlang, its boot script determines which applications it starts. If the SASL application is in there, just stop it before attempting to run the example. In our installation of Erlang, it is not started:

Example 9-1. Loading an application

```

1> application:which_applications().
[{stdlib,"ERTS CXC 138 10","2.0"},
 {kernel,"ERTS CXC 138 10","3.0"}]
2> application:load(sasl).
ok
3> application:start(sasl).
ok
4>

```

1. In case you are wondering, CXC is an internal Ericsson product numbering scheme. It is rumored that a copy of every product with a CXC number is stored in a nuclear-proof bunker at a secret location somewhere in the Swedish woods.

```
=PROGRESS REPORT==== 17-Feb-2014::19:51:08 ===
  supervisor: {local,sasl_safe_sup}
    started: [{pid,<0.42.0>},
               {name,alarm_handler},
               {mfargs,{alarm_handler,start_link,[[]]}},
               {restart_type,permanent},
               {shutdown,2000},
               {child_type,worker}]

...<snip>...

4> application:stop(sasl).

=INFO REPORT==== 17-Feb-2014::19:51:23 ===
  application: sasl
  exited: stopped
  type: temporary
ok
```

SASL is the *systems architecture support libraries* application, a collection of tools for building, deploying, and upgrading Erlang releases. It is part of the minimal OTP release; together with the *kernel* and *stdlib* applications, it has to be included in all OTP compliant releases. We'll cover all of this in more detail later on.

In our example, we load SASL in shell command 2 and start it in shell command 3. You will notice that when we start the application, a long list of progress reports are printed in the shell (we deleted all but the first one from our output). SASL starts its top-level supervisor, which in turn starts other supervisors and workers. These progress reports come from the supervisors and workers started as part of the main supervision tree. We stop the application in shell command 4. Before reading on, have a look at the source code of the SASL callback module, defined in file *sasl.erl*. If unsure where to find it, use the shell command `m(sasl)`. It will tell you where the beam file is located. The source code is up a level, and then down again in a directory called *src*. The functions to look at in the source code are `start/2` and `stop/1`.

Application Resource Files

Every application must be packages with a resource file, often referred to as the *app* file. It contains a specification consisting of configuration data, resources, and information needed to start the application. The specification is a tagged tuple of the format `{application, Application, Properties}`, where *Application* is the atom denoting the application name and *Properties* is a list of tagged tuples.

Let's step through the *sasl* application resource file before putting one together ourselves for the mobile phone example. This is version 2.3.3 of the application, the contents of your app file might differ based on the release you downloaded. Looking at it, you should immediately spot `mod`, which points out the application callback module and arguments passed to the `start/2` callback function. Let's step through the properties in order.

```

{application, sasl,
 [{description, "SASL CXC 138 11"},
  {vsn, "2.3.3"},
  {modules, [sasl, alarm_handler, format_lib_supp, misc_supp, overload, rb,
             rb_format_supp, release_handler, release_handler_1, erlsrv,
             sasl_report, sasl_report_tty_h, sasl_report_file_h, si,
             si_sasl_supp, systools, systools_make, systools_rc, systools_relup,
             systools_lib ]},
  {registered, [sasl_sup, alarm_handler, overload, release_handler]},
  {applications, [kernel, stdlib]}, 
  {env, [{sasl_error_logger, tty}, {errlog_type, all}]},
  {mod, [sasl, []]}]}.

```

The property list contains a set of standard items. All items are optional — if an item is not included in the list, a default value is set — but there are a few that almost all applications set. The list of standard items includes:

{description, Description}

where *Description* is a string of your choice. You will see the description string surface when we called `application:which_applications()` in the shell. The default value is an empty string.

{vsn, Vsn}

where *Vsn* is a string denoting the version of the application. It should mirror the name of the directory, and in automated build systems is set by scripts, not by hand. If omitted, the default value is an empty string.

{modules, Modules}

where *Modules* is a list of modules defaulting to the empty list. The module list is used when creating your release and loading the application, with a one to one mapping between the modules listed here and the beam files included in the *ebin* directory. If your module beam file is in the *ebin* directory, but not listed here, it will not be loaded automatically.² This list is also used to check module name space for clashes between applications, ensuring names are unique.

Each module is specified as an atom denoting the module name, as in the `sasl` example. Up to R15, it was also possible to specify the module version name `{Module, Vsn}`, as it appeared in the `-vsn(Vsn)` directive in the module itself. This is no longer the case.

{registered, Names}

where *Names* contains a list of registered process names running in this application. Including this property ensures that there will be no name clashes with registered names in other applications. Missing a name will not stop the process from running,

2. The code server might load it later when you try to call it.

but could result in a runtime error later when another application tries to register the same name. If omitted, the default value is the empty list.

{applications, AppList}

where *AppList* is a list of application dependencies which must be started in order for this application to start. All applications are dependent on the *kernel* and *stdlib*, and many also depend on *sasl*. Dependencies are used when generating a release to determine the order in which applications are started. Sometimes, only an application such as *sasl* is provided, which in turn depends on *kernel* and *stdlib*. This will work, but it makes the system harder to maintain and understand. The default for this property is the empty list, but it is extremely unusual to omit it since doing so implies there are no dependencies towards other applications.

{env, EnvList}

where *EnvList* is a list of {Key, Value} tuples that set environment variables for the application. Values can be retrieved using functions from the application module: `get_env(Key)` or `get_all_env()` by processes in the application, or via `get_env(Application, Key)` and `get_all_env(Application)` for processes that are not part of the application. Environment variables can also be set through other means covered later in this chapter. This property defaults to the empty list.

{mod, Start}

where *Start* is a tuple of the format {Module, Args} containing the application callback module and arguments passed to its start function. Each tuple results in a call to `Module:start(normal, Args)` when the application starts. Omitting this property will result in the application being treated as a library application, started by a supervisor or worker in another application, and no supervision tree will be created at startup.

Other properties not included in the *sasl.app* file example but that are useful and included in other app files:

{id, Id}

where *Id* is a string denoting the product identifier. It is used by overzealous configuration management trolls, but as you can see, not by the OTP team. The default value is the empty string.

{included_applications, Apps}

where *Apps* is a list of applications included as sub-applications to the main one. The difference with included applications is that their top-level supervisor has to be started by one of the other supervisors. We will cover included applications in more depth in this chapter. Omitting this property will default it to the empty list.

{start_phases, Phases}

which allows the application to be started in phases, allowing it to synchronize with other parts of the system and start workers in the background. *Phases* is a list of

tuples of the format `{Phase, Args}`, where `Phase` is an atom and `Args` is a term. Before `Module:start/2` returns, `Module:start_phase(StartPhase, StartType, Args)` will be called for every phase. `StartType` is the atom `normal`, or the tuples `{takeover, Node}` or `{failover, Node}`. We cover start phases in more detail later in this chapter.

The Base Station Controller Application File

Having looked at how `app` files are constructed, let's create one we can use in the base station controller. Alongside the `description` and application `vsn`, we've listed all of the `modules` which form the application. We follow with a list of the registered worker and supervisor process names, and state in the `applications` list that the `bsc` application is dependent on `sasl`, `kernel` and `stdlib`. We do not set any `env` variables, but explicitly keep the list empty for readability reasons. And finally, the application callback module `mod` is set to `bsc`, passing `[]` as a dummy argument.

```
{application, bsc,
  [{description, "Base Station Controller"},
   {vsn, "1.0"},  

    {modules, [bsc, bsc_sup, frequency, freq_overload, logger, simple_phone_sup,  

              phone_fsm]},  

   {registered, [bsc_sup, frequency, frequency_sup, overload, simple_phone_sup]},  

   {applications, [kernel, stdlib, sasl]},  

   {env, []},  

   {mod, {bsc, []}}}].
```

With the app file completed, all that remains is to place it in the `ebin` directory, compile the source code and make sure the `beam` files are placed in the `ebin` directory.

Starting an Application

When starting the Erlang emulator, include the path to your application `ebin` directory. This is a good habit when testing, because `bsc` might be one of the many applications we have written and need a path to, so starting Erlang directly from the `ebin` directory might not always be an option. Adding a path will no longer be a problem when implementing a release, but do it for now, as it is not set automatically. In our example, we add the path when starting Erlang using:

```
erl -pa bsc-1.0/ebin/
```

but you could also use `code:add_patha/1` to add the path within the Erlang shell.

Let's try starting the `bsc` application. In shell prompt 1, we fail because `sasl`, one of the applications `bsc` depends on, has not been started. We could have avoided that by using `application:ensure_all_started/1`, which starts up an application's dependencies and then starts the application itself, but here we simply resolve it by starting `sasl` in shell command 2 and then starting `bsc` again in shell command 3. For every child started

by our top-level supervisor `bsc_sup`, we get a progress report from `sasl`. This is all happening behind the scenes as a result of using OTP behaviours.

```
1> application:start(bsc).
{error,{not_started,sasl}}
2> application:start(sasl).

...<snip>...

=PROGRESS REPORT==== 26-Nov-2013::11:07:57 ===
    application: sasl
        started_at: nonode@nohost
ok
3> application:start(bsc).

=PROGRESS REPORT==== 26-Nov-2013::11:07:59 ===
    supervisor: {local,bsc_sup}
        started: [{pid,<0.49.0>},
                   {name,freq_overload},
                   {mfargs,{freq_overload,start_link,[]}},
                   {restart_type,permanent},
                   {shutdown,2000},
                   {child_type,worker}]

=PROGRESS REPORT==== 26-Nov-2013::11:07:59 ===
    supervisor: {local,bsc_sup}
        started: [{pid,<0.50.0>},
                   {name,frequency},
                   {mfargs,{frequency,start_link,[]}},
                   {restart_type,permanent},
                   {shutdown,2000},
                   {child_type,worker}]

=PROGRESS REPORT==== 26-Nov-2013::11:07:59 ===
    supervisor: {local,bsc_sup}
        started: [{pid,<0.51.0>},
                   {name,simple_phone_sup},
                   {mfargs,{simple_phone_sup,start_link,[]}},
                   {restart_type,permanent},
                   {shutdown,2000},
                   {child_type,worker}]

=PROGRESS REPORT==== 26-Nov-2013::11:07:59 ===
    application: bsc
        started_at: nonode@nohost
ok
4> l(phone), phone:start_test(150, 500).
*DBG* <0.130.0> got {'$gen_sync_all_state_event',
                      {<0.32.0>,#Ref<0.0.0.1683>},
                      {outbound,109}} in state idle
...<snip>...
```

After starting the base station, we took it for a test run by starting a few hundred phones that randomly call each other. Because the *phone* module is not part of the application, we load it before calling `phone:start_test/2`. In our case, it would not make a difference, but it might if we are running in embedded mode in production, where modules are not loaded automatically. We cover different start modes when looking at release handling in [Chapter 11](#).

If you have run this example, keep the Erlang shell open, type `observer:start()`, and read on.

The Observer Tool

The *observer* is a graphical tool that provides an overview of Erlang-based systems. It replaces and complements deprecated utilities that you might have come across in older versions of Erlang, including the process manager, the table visualizer, and *appmon*,³ the application monitor. To reduce performance overhead in live systems, you should start the observer tool in a separate hidden node, connecting to the cluster you want to observe through distributed Erlang. Because our *bsc* application is still in development mode, we can be lazy and get away with starting the *observer* locally.

The observer window opens up in the System tab, where you can view general information such as hardware architecture, version of the runtime system, and operating system specific data. You will also find details of the CPUs and schedulers, memory usage, and general runtime statistics. The Load Charts tab will plot memory usage, scheduler utilization, and IO usage in real time. Although the observer will not replace proper metrics and monitoring or store historical data, it helps you understand the behaviour of a system under development.

The Applications tab contains a list of applications sorted in alphabetical order ([Figure 9-7](#)). Click on any of the applications and you will see the respective supervision trees, showing how workers and supervisors are linked to each other. Narrow down on the *bsc* app.

3. Up to Erlang R16B, you can still start *appmon* using `appmon:start()`.

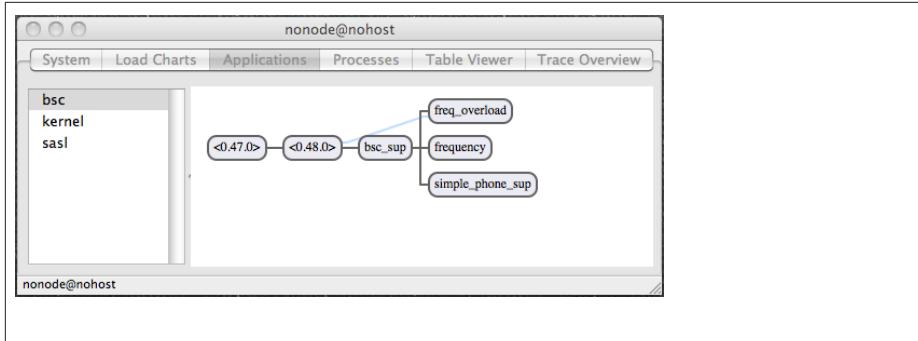


Figure 9-7. The Observer

The first thing you should notice is the two application master processes. Note how one of them is linked to the frequency overload event manager. You might recall that in the `start_link` function of the `bsc_sup` module covered in “[Gluing it all together](#)” on page 183, we added supervised handlers to the `freq_overload` event manager. Adding the supervised handlers in the `start_link` function of the `freq_overload` worker was executed by process `<0.48.0>`. As a result, this process is now monitored by the event manager.

Click on any of the processes and you will get a window containing information on the process itself, the message queue, the dictionary, and the stack trace. You can view the same window from the Processes tab. The Table Viewer is a port of the table visualizer, allowing you to inspect Mnesia and ets tables. Finally, the Trace Overview is a graphical interface to the trace BIFs and dbg. You can read more about all these options in the Observer User’s Guide and Reference Manual.

Environment Variables

Erlang uses environment variables mainly to obtain configuration parameters when initializing the application behaviours. You can set, inspect and change these variables. Start an Erlang shell, make sure the `sasl` application is running, and type `application:get_all_env(sasl)`. Don’t worry about the meaning of environment variables for now — we explain them later when we cover `sasl` reports — but be aware that they are not the same as the environment variables supported by your operating system shells. For now, we focus just on how they are set and retrieved.

If you ran the `get_all_env(sasl)` call as we suggested, you saw that it returns the environment variables belonging to the `sasl` application. If you want a specific variable, say `errlog_type`, use `application:get_env(sasl, errlog_type)`. If the process retrieving the environment variables is part of an application’s supervision tree, you can

omit the application name and just call `application:get_all_env()` or `application:get_env(Key)`.

Using functionality similar to that in the `application:get_application()` call, OTP uses the process group leader to determine the application to which the process belongs. In our examples, we are using the shell, which is not part of the `sasl` application supervision tree, so we have to specify the application.

Where are these environment variables set? If you look at the `sasl.app` file, you will find them in the `env` attribute of the application resource file. The `app` file usually contains default values you might want to override on a case-by-case basis, depending on the system and use of the application. This is best done using the system configuration file. It is a plain text file with the `.config` suffix containing an Erlang term of the format

```
[{Application1, [{Key1, Value1}, {Key2, Value2}, ...]},  
 {Application2, [{Key2, Value2}|...]}].
```

Inform the application controller which configuration file to read when starting the Erlang VM by using:

```
$ erl -config filename
```

where `filename` is the name of the system configuration file, with or without the `.config` suffix.

If prototyping, testing, or troubleshooting, you can override values set in the `app` and `config` files at startup in the command line prompt using:

```
$ erl -application key value
```

Although convenient, this approach should not be used to set values in production systems. For the sake of clarity, stick to `app` and `config` files, as they will be the first point of call by anyone debugging or maintaining the system.

With this knowledge at hand, let's write our own `bsc.config` file containing the frequencies for our frequency allocator example and override some of the `sasl` environment variables:

```
[{sasl, [{errlog_type, error}, {sasl_error_logger, tty}]}],  
 {bsc, [{frequencies, [1,2,3,4,5,6]}]}].
```

The file overrides the `errlog_type` and the `sasl_error_logger` environment variables set in the `app` file. To test the configuration parameters from the shell, start the Erlang node and provide it with the name of the configuration file, placed in the same directory where you start Erlang. In production systems, config files are placed in specific release directories. We will look at them in more detail in [Chapter 11](#).

In the following command starting the `erl` shell, we take configuration a step further and override `sasl_error_logger`, setting its value to `false`.

```

$ erl -config bsc.config -sasl sasl_error_logger false -pa bsc-1.0/ebin/
Erlang R16B (erts-5.10.1) [source-05f1189] [smp:4:4] [async-threads:10] [hipe]
[kernel-poll:false]
Eshell V5.10.1 (abort with ^G)
1> application:start(sasl).
ok
2> application:get_all_env(sasl).
[{errlog_type,error}, {included_applications,[]}, {sasl_error_logger,false}]
3> application:start(bsc).
ok
4> application:get_env(bsc, frequencies).
{ok,[1,2,3,4,5,6]}
5> application:set_env(bsc, frequencies, [1,2,3,4,5,6,7,8,9]).
ok

```

In shell command 1, we start *sasl*, retrieving all of its environment variables in shell command 2. Note the final values of the environment variables:

- `errlog_type` is set in the *bsc.config* file, overriding the value set in the *app* file.
- `included_applications` comes from the *app* file. Not originally an environment variable, it is converted into one by the application controller.
- `sasl_error_logger` is set in the *app* file, overridden in the *config* file, and overridden again on the Unix prompt level when starting Erlang.

The `frequencies` environment variable can be used in the `get_frequencies()` call of the *frequency* server to retrieve the frequencies. Note how we do not have to specify the application name in the code, because the runtime can determine the application from the group leader of the process making the call:

```

get_frequencies() ->
    {ok, FreqList} = application:get_env(frequencies),
    FreqList.

```

Note that if you downloaded the code from the book repository, within the `frequency.erl` file under the `bsc-1.0` application directory, the `get_frequencies/0` function shown here is commented out. Comment out the default version and uncomment this version and recompile the module to have it work properly with the `bsc.config` file.

In shell command 5, we set environment variables directly in the Erlang shell. The application name is optional; if not provided, the environment variables will override those of the application belonging to the process executing the call. In our example from the shell, we provided the application because the shell process is not part of the `bsc` or `sasl` supervision trees.



Although there is nothing stopping you from setting environment variables in the shell using the `application:set_env` functions, it is advisable to do so only for applications you have written yourself or know well. For third party applications, including those that are part of the Erlang distribution, changing environment variables once the application has been started is dangerous. As you do not know when and where the application reads these environment variables, changing them may cause it to enter an inconsistent state and behave unexpectedly. Do so at your own risk, and only if you know how the values are read and refreshed by the behaviours using them.

Distributed Applications

OTP comes with a convenient distribution mechanism for migrating applications across nodes. It can handle the majority of cases where you need an instance of an application running in your cluster, and can act as a stopgap measure until a more complex solution can be put in place. The majority of cases assumes reliable networks, so use with care and make sure you have covered your edge cases should a netspilt occur.

Distributed applications are managed by a process called the *distributed application controller*, implemented in the `dist_ac` module and registered with the same name. You will find an instance of this process in the kernel supervision tree running on every distributed node.

To run your distributed application, all you need to do is configure a few environment variables in the `kernel` application, ensure that requests are transparently forwarded to the node where the applications are running, and then test, test, and test again. You have to specify the node precedence order where you want the application to run. If the node on which an application is running fails, the application will *failover* to the next node in the precedence list. If a newly started or connected node with higher precedence appears in the cluster, the application will be migrated to that node in what OTP calls a *takeover*.

Let's assume our system consists of a cluster of four nodes, `n1@localhost`, `n2@localhost`, `n3@localhost` and `n4@localhost`. Let's create a configuration file, `dist.config`, setting the kernel environment variables distributing our `bsc` application across them.

```
[{kernel, [{distributed, [{bsc, 1000, [n1@localhost, {n2@localhost, n3@localhost, n4@localhost}]}]}, {sync_nodes_mandatory, [n1@localhost]}, {bsc, [{frequencies, [1,2,3,4,5,6]}]}}].
```

Note that if you intend to run the distributed `bsc` example, you will need to replace all occurrences of the string “localhost” in the `dist.config` file with your own computer’s host name.

Of the environment variables in the kernel application, the first we need to set is distributed. It consists of a list of tuples containing the application we want to distribute, a timeout value, and the distributed list of node and node tuples, which defines the order of precedence of nodes on which we want the application to run. So this list:

```
[{bsc, 1000, [n1@localhost,{n2@localhost,n3@localhost},n4@localhost]}]
```

specifies bsc as the application, 1000 (measured in milliseconds) as the time waited for the node to come back up, and the following node precedence:

```
[n1@localhost,{n2@localhost,n3@localhost},n4@localhost]
```

The precedence specifies that the application App will start on *n1*. Should that node fail or be shut down, the distributed application controller will wait 1 second and then fail over the application to either *n2* or *n3*. They have been given the same precedence by being grouped into the same tuple. If both *n2* and *n3* fail, the controller will check to see whether *n1* has come back up, and if it is still down, will fail the application over to *n4*. If one of the other nodes comes back up, the application is later moved via a takeover to the node with the highest precedence.

The sync_nodes_mandatory and sync_nodes_optional environment variables specify the nodes to be connected into the distributed system. When starting the system, the distributed application controller tries to connect the specified nodes, waiting for the number of milliseconds specified in the {sync_nodes_timeout, Timeout} environment variable. If you omit the timeout when defining the nodes in your kernel environment variables, the timeout defaults to 0.

The {sync_nodes_mandatory, NodeList} environment variable defines the nodes with which the distributed application controller *must* synchronize; the system will start only if all of these nodes are started and connected to each other within Timeout milliseconds.

The environment variable {sync_nodes_optional, NodeList} specifies nodes that can also be connected at system startup, but unlike mandatory nodes, the failure of any of these nodes to join the cluster within the specified Timeout does not prevent the system from starting up.

The best way to understand the environment variable settings is to play with the dist.config configuration file. Let's first start node *n2* on its own:

```
$ erl -sname n2 -config dist -pa bsc-1.0/ebin
```

The node will wait the 15 seconds set in the sync_node_timeout value for *n1* to come up. If the node fails to connect to *n1* within that time frame, it will terminate, regurgitating a long, and to the untrained eye incomprehensible, error message. Nodes *n3* and *n4* are optional, so assuming *n1* comes up within the timeout period, *n2* will also wait for these two nodes within the same period, after which it starts normally whether or not *n3* and *n4* have connected.

Let's try again, but this time, before starting *n2*, start *n1* and *n3*:

```
$ erl -sname n1 -config dist -pa bsc-1.0/ebin  
$ erl -sname n3 -config dist -pa bsc-1.0/ebin
```

The nodes will wait 15 seconds for the optional nodes to come up. If they don't, the nodes will start regardless. You can try deleting *n4* from the config file (or decide to start it), avoiding the timeout if the other nodes are up.

When all nodes are up, let's start the *sasl* and the *bsc* application on all nodes starting with *n3*, followed by *n2* and *n1*. Type the following in the Erlang shells and pay attention to when the shell command returns:

```
application:start(sasl), application:start(bsc).
```

You will notice that the shell will hang in *n2* and *n3*, returning only when the *bsc* application is started in *n1*, as it is the node running with the highest priority. If you start the observer and inspect the application tab on the different nodes, you will notice that the supervision tree is started only on *n1*. Looking at the progress reports in *n2* and *n3*, you will notice that the *bsc* application is also started, but without its supervision tree.

Keeping an eye on the nodes *n2* and *n3*, shut down node *n1* using the `halt()` shell command.

The application controller will wait 1000 ms for *n1* to restart. If it doesn't, you will see the progress reports for the *bsc* app being started on either *n2* and *n3*. In our config file, because both *n2* and *n3* have the same precedence, either one will be chosen non-deterministically. In [Figure 9-8](#), we are assuming that the chosen node is *n2*.

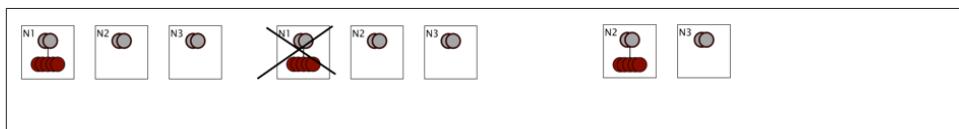


Figure 9-8. Failing over with different precedence

Now that *n1* is down, let's shut down *n2* (or *n3* if the *bsc* application was started on it instead). You will see that application will failover to the remaining node ([Figure 9-9](#)). Use the observer to check that the supervision tree has started correctly ([Figure 9-7](#)). Restart the node you just shut down and observe what happens. You will notice that it hangs for 15 seconds, waiting for *n1* to restart. Because *n1* is mandatory and has not restarted, the node fails to restart.



Figure 9-9. Failing over with the same precedence

Restart both *n1* and *n2* (or *n3* if it was the node that shut down) within 15 seconds of each other. Both will wait 15 seconds for the nonmandatory node *n4* to start. After the timeout, start both *sasl* and *bsc* on *n2* using `application:start/1`. Just as the first time you started the cluster, the application hangs waiting for *bsc* to start on *n1* so that the nodes can coordinate among each other. When you start *bsc* on *n1*, there will be a takeover from *n3*, where the behaviours are terminated and the supervision tree is taken down (Figure 9-10).



Figure 9-10. Application Takeover

This is a limited approach which might cover some use cases and not others. The moral of the story if you go down this route is to pick your mandatory nodes with care. When designing your system with no single point of failure, you should not assume or require any of the nodes to be up at any one time. If there are services you require for a failover or a takeover to be successful, do the checks in start phases when starting the applications or in the worker processes themselves. While this layer can be thin and consist only of a couple hundred lines of code, it is application dependent. Make sure you've thought through your design. We will look at other approaches to distributed architectures when discussing clusters in [Chapter to Come].

Start Phases

Some systems are so complex that it is not enough to start each application one at a time. In such systems, applications need to be started in phases and synchronized with each other. Imagine a node that is part of a cluster handling instant messaging:

1. In a first phase, as a background task, you might want to start loading all of the mnesia tables containing routing and configuration data. This could take time, as some of the tables might have to be restored because of an abrupt shutdown or node crash.

- Once the tables load, the next phase gets your system to a state where you are ready to start accepting requests. We refer to this as enabling the administration state. This might include checking links towards other clusters in the federation that users might want to connect to, configuring hardware and waiting for all of the other parts of the system such as the authentication server or logging facility to start correctly.
- When this phase completes, you will be able to inspect and configure the system, but not allow any users to initiate sessions. Your final start phase might be to provide the go-ahead and start allowing users to log on and traffic to run through this node. We refer to this phase as enabling the operational state.

If we add the following parameter in our bsc.app file,

```
{start_phases, [{init, []}, {admin, []}, {oper, []}]}
```

we allow three start phases. In our application callback module source, *bsc.erl*, we need to export and define the callback function `start_phase(StartPhase, StartType, Args)`. This function will be called for every phase defined in the app file, after the supervision tree has been started, but before `application:start(Application)` returns. The `StartPhase` argument reflects which phase is currently being processed. So, in our example, if we added

```
start_phase(StartPhase, StartType, Args) ->
    io:format("bsc:start_phase(~p,~p,~p).~n", [StartPhase, StartType, Args]).
```

to our application callback module, we would get the following sequence of events when starting the application:

```
$ erl -pa bsc-1.0/ebin/ -sasl -sasl_error_logger false
Erlang R16B (erts-5.10.1) [source-05f1189] [smp:4:4] [async-threads:10] [hipe] [kernel-poll:false]

Eshell V5.10.1  (abort with ^G)
1> application:start(sasl), application:start(bsc).
bsc:start_phase(init,normal,[]).
bsc:start_phase(admin,normal,[]).
bsc:start_phase(oper,normal,[]).
ok
```

Here, the `StartType` argument is always the atom `normal`, indicating this is a normal startup. Each phase invokes a synchronous or asynchronous call that triggers certain operations, as well as setting the internal state that allows or disallows requests to be handled by the node.

When shutting down the system, we can disable the operational state, stopping new requests from executing, but allowing all existing requests to execute to completion. This could make the system to reject user login attempts while allowing existing sessions to expire. When there are no more requests going through the node, the operational state can be disabled and the node shut down. This could happen when all the users

have logged out, or after a timeout, where the system times out the remaining sessions and disables the operational state. To shut down the node, disable the operational state. A simple example using start phases appears in the next section.

Included Applications

In your *app* resource file, you have the option of specifying the `included_applications` parameter. The directory structure of included applications should be placed in the `lib` directory, alongside all other applications in that release. When the main application is started, all included applications are loaded but not started. It is up to the top-level supervisor of the main application to start the included applications' supervision trees. You could start them as dynamic children or as static ones by returning the child specification in the supervisor `init/1` callback function.

When starting your application, you can either call the `start/2` function in the application callback module, assuming it returns `{ok, Pid}` (and not `{ok, Pid, Data}`, since it is not possible for us to pass that data to the callback module's `prep_stop/1` callback function as it expects when it is stopped), or by directly calling the `start_link` function of top-level supervisor. There is no more to it, it is as simple as that!

In every node, included applications may be included only once by other applications. This restriction avoids clashes in the application name space, ensuring that each module and registered process (local or global) are unique. If you need to start several identical supervision trees in the same node, place the code in a stand-alone library application. Do not include this application anywhere else other than by dependency and ensure that there are no name clashes with the locally and globally registered processes.

You might be asking yourself, why go through the hassle of included applications when we can instead have a flat application structure, starting the applications individually? The answer lies in start phases.

Start Phases in Included Applications

You can use *start phases* to synchronize your included applications at startup. As the included application supervision trees are started by the main application, you need to follow a few steps to invoke the `start_phase/3` callback function in the application callback module.

First, in your included application app files, make sure you have included the `mod` and `start_phases` parameters. The callback module is used to determine where the `start_phase/3` call is made. The arguments are ignored, because the ones in the `start_phases` item are used.

Finally, in your top-level application, alongside your start phases, you need to change your `mod` parameter to:

```
{mod, {application_starter,[Mod,Args]}}
```

passing the application callback module Mod and Args as arguments. The OTP application_starter module provides the logic to start your top-level application and coordinate the start phases of the included applications.

The process is straightforward: the top-level application's supervision tree starts the included applications. The first `start_phase/3` function is called in the callback module of the top-level application, after which all included applications are traversed in the order they are defined. If one or more of the included applications have the same phase defined as the one in the top-level application, `start_phase/3` is called for each of these included applications.

The next start phase in the top-level application is recursively triggered. Start phases defined in the included applications but not in the top-level application are never triggered.

All of what we've described is best shown in an example. We will create a top-level application, `top_app` that includes the `bsc` application. The `top_app` callback module is responsible for starting the supervision tree of the included `bsc` application:

```
-module(top_app).
-behaviour(application).
-export([start/2, start_phase/3, stop/1]).

start(_Type, _Args) ->
    {ok, _Pid} = bsc_sup:start_link().

start_phase(StartPhase, StartType, Args) ->
    io:format("top_app:start_phase(~p,~p,~p).~n", [StartPhase, StartType, Args]).

stop(_Data) ->
    ok.
```

In our top application's `top_app.app` file, we define the `start`, `admin`, and `stop` phases. They are different from the start phases in `bsc`, which in “[Start Phases](#)” on page 217, our previous example, were set to `init`, `admin`, and `oper`. Note also the included applications attribute and the value we give the `mod` attribute.

```
{application, top_app,
  [{description, "Included Application Example"},
   {vsn, "1.0"}, {modules, [top_app]}, {applications, [kernel, stdlib, sasl]}, {included_applications, [bsc]}, {start_phases, [{start, []}, {admin, []}, {stop, []}]}, {mod, {application_starter, [top_app, []]}}]}.
```

The start phases work as follows. The top application is started, which in turn starts the *bsc* supervision tree. Once that is successful, the first start phase in the *top_app*, *start*, is triggered. If any of the included applications, in the order they appear in the `included_applications` list, also has this phase, it is also called:

```
$ erl -pa bsc-1.0/ebin/ -sasl sasl_error_logger false
Erlang R16B (erts-5.10.1) [source-05f1189] [smp:4:4] [async-threads:10] [hipe] [kernel-poll:false]

Eshell V5.10.1 (abort with ^G)
1> application:start(sasl), application:start(top_app).
top_app:start_phase(start,normal,[]).
top_app:start_phase(admin,normal,[]).
bsc:start_phase(admin,normal,[]).
top_app:start_phase(stop,normal,[]).
ok
```

We have kept the example simple so as to demonstrate the principles without getting lost in the business logic. In our example, we call all of the start phases in the top application, but only `admin` in the included one, as it is the only phase they both have in common.

Combining Supervisors and Applications

Some supervisor callback modules contain only a few lines of code. And if your application does not have to deal with complex initialization procedures, start phases and distribution, but needs only to start the top-level supervisor, it will be just as compact. A common practice is to combine the two callback modules, as their callback function names do not overlap. Whilst some people will strongly disagree with this practice, you are bound to come across it when reading other people's code, even code with is part of the standard Ericsson distribution.

For example, `cd` into the `sasl` directory of your OTP installation and have a look at the `sasl.erl` file. At the time of writing, version 2.3.3 of the `sasl` application combined the supervisor `init/1` callback function in its application module together with the application `start/2` and `stop/1` callback functions. In this example, they included only the `-behaviour(application)`. directive, but there is nothing stopping you from also including the `-behaviour(supervisor)`. directive as well. The only side effect is a compiler warning telling you about two behaviour directives in the same callback module. We recommend including both directives, because it facilitates the understanding of the purpose of the callback module. Here is a simple example of what combining the supervisor and application callback modules would look like in our `bsc` example:

```
-module(bsc).
-behaviour(application).
-behaviour(supervisor).

-export([start/2, stop/1, init/1]).
```

```

%% Application callbacks

start(_Type, _Args) ->
    {ok, Pid} = supervisor:start_link([{local, ?MODULE}, ?MODULE, []]).

stop(_Data) ->
    ok.

%% Supervisor callbacks

init(_) ->
    ChildSpecList = [child(freq_overload), child(frequency), child(simple_phone_sup)],
    {ok, {{rest_for_one, 2, 3600}, ChildSpecList}}.

child(Module) ->
    {Module, {Module, start_link, []}, permanent, 2000, worker, [Module]}.

```

The SASL Application

Throughout this chapter, we've been telling you to look at SASL applications's callback modules, app files, directory structure, and supervision tree. All of this without telling you what SASL actually does. SASL stands for the System Architecture Support Libraries.

The SASL application is a container for useful items needed in large-scale software design. It is one of the mandatory applications (alongside *kernel* and *stdlib*) required in a minimal OTP release. It is mandatory because it contains all of the common library modules used for release handling and software upgrades.

SASL doesn't stop, however, at release and software upgrades. In “[The SASL Alarm Handler](#)” on page 157, we looked at the alarm handler, a simple alarm manager and handler that is started by default when you start any OTP-based system. We cover releases in [Chapter 11](#) and software upgrades in [\[Chapter to Come\]](#). SASL has a very basic way, through its *overload* library module, to regulate CPU load in the system. We cover load regulation in more detail in [\[Chapter to Come\]](#), when we discuss the architecture of a typical Erlang node. Have patience.

What we will concentrate on in this chapter are the SASL reports used to monitor the activity in supervision trees when processes are started, terminated, and restarted. You will have come across SASL reports in the previous chapters of this book. They are the printouts you see in the shell when starting applications, supervisors, and worker processes. You might have noticed that they appeared only when the SASL application was started and the `sasl_error_logger` environment variable was not set to `false`.

SASL starts an event handler that receives the following reports:

Supervisor reports

Issued by a supervisor when one of its children terminates abnormally.

Progress reports

Issued by a supervisor when starting or restarting a child or by the application master when starting the application.

Error reports

Issued by behaviours upon abnormal termination.

Crash reports

Issued by processes started with the `proc_lib` library, which by default include behaviours. We cover `proc_lib` in the next chapter.

Default settings print reports to standard I/O. You can override this by setting environment variables, which allow you to send the reports to wraparound binary logs as well as to limit which reports are forwarded. The format of the reports vary depending on the version of the OTP release you are running. Let's have a look at the SASL environment variables that allow you to control the reports:

`sasl_error_logger`

Defaults to `tty` and installs the `sasl_report_tty_h` handler module, which prints the reports to standard out. If you instead specify `{file, FileName}`, where *FileName* is a string containing the relative or absolute path of a file, the `sasl_report_file_h` handler is installed, storing all reports in *FileName*. If this environment variable is set to `false`, no handlers are installed, and as a result, no SASL reports are generated.

`errlog_type`

Can take the values `error`, `progress`, or `all`, the default if you omit the variable. Use this variable to restrict the types of error or progress reports printed or logged to file by the installed handler.

`utc_log`

An optional environment variable, which if set to true, will convert all timestamps in the reports to Universal Coordinated Time.

The following configuration file stores all the SASL reports in a text file called `SASLlogs`. We do this by setting the `sasl_error_logger` to `{file, "SASLlogs"}`. We also enable UTC time with the `utc_log` environment variable. `errlog_type` is by default set to `all`.

```
[{sasl, [{sasl_error_logger, {file, "SASLlogs"}}, {utc_log, true}]},  
 {bsc, [{frequencies, [1,2,3,4,5,6]}]}].
```

If you start the the `sasl` and `bsc` applications in a local, non-distributed node, you will find all of the logs stored as plain text in the running directory. In our example, we are just showing the first and the last reports. Note how the UTC tag is appended to the time stamp:

```
$ cat SASLlogs

=PROGRESS REPORT==== 8-Dec-2013::10:09:25 UTC ===
    supervisor: {local,sasl_safe_sup}
        started: [{pid,<0.40.0>},
                   {name,alarm_handler},
                   {mfargs,{alarm_handler,start_link,[[]]}},
                   {restart_type,permanent},
                   {shutdown,2000},
                   {child_type,worker}]

...<snip>...

=PROGRESS REPORT==== 8-Dec-2013::10:09:33 UTC ===
    application: bsc
    started_at: nonode@nohost
```

Text files might be good during your development phase, but when moving to production, it is best to move to wrap-around logs that store events in a searchable binary format. Because text and binary formats are implemented by different handlers, they can be added and run alongside each other. To install the binary log handler, `error_logger_mf_h`, you have to set three environment variables. If any one is disabled, the handler will not be added. The environment variables needed are:

error_logger_mf_dir

A string specifying the directory that stores the binary logs. The default is a period ("."), which specifies the current working directory. If this environment variable is set to `false`, the handler is not installed.

error_logger_mf_maxbytes

An integer defining the maximum size in bytes of each log file.

error_logger_mf_maxfiles

An integer between 1 and 256 specifying the maximum number of wrap-around log files that are generated.

Sticking to our *bsc* example, let's try storing the SASL logs in a binary file using the *rb.config* configuration file found in the book's code repository. Note how we are explicitly turning off the events sent to the shell by setting the *sasl_error_logger* to *false* and the frequencies to the atom *crash*, ensuring that the process fails when we try to allocate a frequency:

```
[{sasl, [{sasl_error_logger, false},  
        {error_logger_mf_dir, "."}, {error_logger_mf_maxbytes, 20000}, {error_logger_mf  
        bsc, [{"frequencies, crash}]}]}].
```

We start the `bsc` application in shell command 1, and cause a crash of the frequency server in shell command 2 when we try to pattern match the atom `crash` into a head and a tail in the `allocate/2` function of the `frequency` module.

```

$ erl -pa bsc-1.0/ebin/ -config rb.config
Erlang R16B (erts-5.10.1) [source-05f1189] [smp:4:4] [async-threads:10] [hipe] [kernel-poll:false]

Eshell V5.10.1 (abort with ^G)
1> application:start(sasl), application:start(bsc).
ok
2> frequency:allocate().

=ERROR REPORT==== 8-Dec-2013::19:39:05 ===
** Generic server frequency terminating
** Last message in was {allocate,<0.32.0>}
** When Server state == {data,[{"State",{{available,crash},{allocated,[]}}}]}}
** Reason for termination ==
** {function_clause,[{frequency,allocate,
                     [{crash,[]},<0.32.0>],
                     [{file,"frequency.erl"}]},.....
...<snip>...

3> rb:start().
rb: reading report...done.
{ok,<0.56.0>}
4> rb:list().

No          Type      Process      Date       Time
--          ===      =====      ===       ===
14          progress   <0.35.0>  2013-12-08 19:28:57
13          progress   <0.35.0>  2013-12-08 19:28:57
12          progress   <0.35.0>  2013-12-08 19:28:57
11          progress   <0.35.0>  2013-12-08 19:28:57
10          progress   <0.24.0>  2013-12-08 19:28:57
9           progress   <0.44.0>  2013-12-08 19:28:57
8            progress   <0.44.0>  2013-12-08 19:28:57
7            progress   <0.44.0>  2013-12-08 19:28:57
6            progress   <0.24.0>  2013-12-08 19:28:57
5            error     <0.44.0>  2013-12-08 19:39:05
4          crash_report frequency  2013-12-08 19:39:05
3 supervisor_report <0.44.0>  2013-12-08 19:39:05
2            progress   <0.44.0>  2013-12-08 19:39:05
1            progress   <0.44.0>  2013-12-08 19:39:05
ok

```

Try it out yourself in the shell, as it will help you understand how applications and supervision trees work. The first thing you will notice is that, even though we set the `sasl_error_logger` to false, we still get an error report. This is because all these environment variable controls are `supervisor`, `crash` and `progress` reports. `Error` reports are printed out irrespective of configuration file settings. We've reduced the size of this particular error report in the trial run, because our focus is on the report browser.

Having caused a crash, we start the report browser using `rb:start()`. After it reads in all of the reports, we list them in shell command 4 with `rb:list()`. If at any time you do not recall the report browser commands, `rb:help()` will list them. The progress

reports 14-6 (they are listed in reverse order, with the oldest having the highest number), are the ones starting the application and its supervision tree. Let's start by inspecting reports 1-5.

- The frequency server generates reports 5 and 4 as a result of its abnormal termination. The reports contain complementary information needed for post mortem debugging and troubleshooting.
- The supervisor generates report 3 as a result of the termination. It contains the information stored by the supervisor of that particular child.
- Report 1 and 2 are issued by the children being restarted. In our case, it is the frequency server that crashed and the `simple_phone_sup` supervisor that was terminated and restarted as a result of the rest for all strategy of the top-level `bsc_sup` supervisor.

Progress Reports

Progress reports are issued by a supervisor when starting a child, worker or supervisor alike. These reports include the name of the supervisor and the child specification of the child being started. They are also issued by the application master when starting or restarting an application. In this case, the report shows the application name and the node on which it is started.

```
5> rb:show(6).  
  
PROGRESS REPORT <0.7.0> 2013-12-08 19:28:57  
=====  
application bsc  
started_at nonode@nohost  
  
ok
```

The progress report in our example is the one telling us that the `bsc` application was started correctly. Note how we are using `rb:show/1` to view individual reports.

Error Reports

Error reports are raised by behaviours upon abnormal termination. In our case, the frequency server generates the report when terminating abnormally. You can generate your own error reports using the `error_logger:error_msg(String, Args)` call, but it is advised not to. Use this command sparingly and only for unexpected errors, as too many user generated reports will hide serious issues and clutter the logs, making it harder to find important details when you are looking for crash reports and other real errors.

```

6> rb:show(5).

ERROR REPORT  <0.48.0>                               2013-12-08 19:39:05
=====
** Generic server frequency terminating
** Last message in was {allocate,<0.32.0>}
** When Server state == {data,[{"State",{{available,crash},{allocated,[]}}}}}
** Reason for termination ==
** {function_clause,[{frequency,allocate,
                     [{crash,[]},<0.32.0>],
                     [{file,"frequency.erl"},{line,101}]}],
  {frequency,handle_call,3,
   [{file,"frequency.erl"},{line,68}]}},
  {gen_server,handle_msg,5,
   [{file,"gen_server.erl"},{line,588}]}},
  {proc_lib,init_p_do_apply,3,
   [{file,"proc_lib.erl"},{line,239}]}]}
ok
7> error_logger:error_msg("Error in ~w. Division by zero!~n", [self()]). 
ok

=ERROR REPORT==== 8-Dec-2013::19:45:13 ===
Error in <0.54.0>. Division by zero!

```

Crash Reports

Crash reports are issued by processes started with the `proc_lib` library. If you look at the exit reason in our example, you will realize that this applies to all behaviours, which are started from that library. A `try-catch` in the main behaviour loop will trap abnormal terminations and generate a crash report. No reports are generated if the behaviour or process terminates with reason `normal` or when the supervisor terminates the behaviour with reason `shutdown`. A crash report will contain information on the crashed process, including exit reason, initial function, and message queue, as well as other process information typically found using the `process_info` BIFs.

```

8> rb:show(4).

CRASH REPORT  <0.48.0>                               2013-12-08 19:39:05
=====
Crashing process
  initial_call                                {frequency,init,['Argument_1']}
  pid                                         <0.48.0>
  registered_name                            frequency
  error_info
    {exit,
     {function_clause,
      [{frequency,allocate,
        [{crash,[]},<0.32.0>],
        [{file,"frequency.erl"},{line,101}]}],
      {frequency,handle_call,3,[{file,"frequency.erl"},{line,68}]}},
    }

```

```

{gen_server,handle_msg,5,
 [{file,"gen_server.erl"},{line,588}]],
{proc_lib,init_p_do_apply,3,
 [{file,"proc_lib.erl"},{line,239}]}],
[{gen_server,terminate,6,[{file,"gen_server.erl"},{line,747}]}],
{proc_lib,init_p_do_apply,3,
 [{file,"proc_lib.erl"},{line,239}]}]}
ancestors [bsc,<0.45.0>]
messages []
links [<0.46.0>]
dictionary []
trap_exit false
status running
heap_size 610
stack_size 27
reductions 395

```

ok

Supervisor Reports

Supervisor reports are issued by supervisors upon abnormal child termination. They usually follow the error reports issued by the children themselves. The supervisor report will contain the name of the reporting supervisor and the phase of the child in which the error occurred.

```

9> rb:show(3).

SUPERVISOR REPORT <0.46.0> 2013-12-08 19:39:05
=====
Reporting supervisor {local,bsc}

Child process
  errorContext child_terminated
  reason
    {function_clause,
     [{frequency,allocate,
       [{crash,[]},<0.32.0>],
       [{file,"frequency.erl"},{line,101}]}},
     {frequency,handle_call,3,[{file,"frequency.erl"},{line,68}]}},
     {gen_server,handle_msg,5,[{file,"gen_server.erl"},{line,588}]}},
     {proc_lib,init_p_do_apply,3,
      [{file,"proc_lib.erl"},{line,239}]}]}
  pid <0.48.0>
  name frequency
  mfargs {frequency,start_link,[]}
  restart_type permanent
  shutdown 2000
  child_type worker

ok

```

If you look close to the top of the example output, you will find the report phase of the child when the error occurred: `start_error`, `child_terminated`, or `shutdown_error`. In our case, the termination happened because of a runtime error, resulting in the report phase being `child_terminated`. It is followed with the reason for termination and the child specification.

You can look at the last two progress reports on your own. They are the progress reports generated when the frequency server and phone supervisor are restarted. Using `rb:help()`, and spend some time experimenting with the commands in the report browser, especially the filters and regular expressions.

The SASL Logs Will Bail You Out

The SASL logs should by default be enabled on all nodes in production, as they will be your first point of call when investigating a node crash or trying to restart the node. In the majority of cases, the error, crash, and supervisor reports will contain enough information to figure out what happened. Always have a separate start script that allows you to start Erlang (and SASL) on its own, using the command `rb:start([{re port_dir, Dir}])` to load the logs, because there is a good chance the Erlang node with your release will not be able to restart. Do not rely on your Erlang node you are investigating to read them, as it is the very node which you are investigating and most likely will not start. If you have an external alarm and monitoring system, it is always a good idea to generate notifications when you receive error, crash, and supervisor reports to ensure you investigate them. With many, potentially thousands of nodes in production, aggregating these notifications in one place will make life much easier for you. You can easily forward them to third party tools by writing your own event handler and hooking it into the SASL event manager.

Summing Up

In this chapter, we covered the behaviour that allows us to package code, resources, configuration files and supervision trees into what we call an application. Applications are the reusable building blocks of your systems; they are loaded, started, and stopped as a single unit. They provide functionality such as start phases, synchronization, and failover in distributed clusters, as well as basic monitoring and logging services.

Table 9-1 lists the major functions used to control applications.

Table 9-1. application callbacks

application call	application callback function
<code>application:start/1</code>	<code>Module:start/2, Module:start_phase/3</code>
<code>application:stop/1</code>	<code>Module:prep_stop/1, Module:stop/2</code>

You can read more about applications in the *application* manual pages, particularly about their resource files in the *app* section (see <http://www.erlang.org/doc/man/application.html>). The *OTP Design Principles User's Guide*, which comes with the standard Erlang documentation, has sections covering *general*, *included*, and *distributed* applications (see http://www.erlang.org/doc/design_principles/des_princ.html). To learn more about the tools we've covered, consult the manual pages for the report browser, *rb* (<http://www.erlang.org/doc/man/rb.html>), as well as the *observer* (<http://www.erlang.org/doc/man/observer.html>). Read through the code of the examples provided in this chapter and see how applications in the Erlang distribution are packaged and configured.

What's Next?

Now that we know how to create our applications, the basic building blocks for Erlang systems, next we look at how to group them together in a release and start our systems using boot files. But first, we look at some of the libraries used to implement special processes, and using that knowledge to define our own behaviours. What are special processes, I hear you say? They are processes which, despite not being OTP behaviours which come as part of the stdlib application, can be added to OTP supervision trees. Read on to find out more.

Special Processes and Your Own Behaviours

OTP behaviours, in the vast majority of cases, provide you with the concurrency design patterns you need in your projects. There might, however, be occasions where you want to create an OTP compliant application whilst attaching processes to your supervision tree which are not standard behaviours. For instance, existing behaviours might have performance impacts caused by the overhead of the layers added as a result of abstracting out the generic parts and error handling. You may want to write new behaviours after separating your code into generic and specific modules. Or you might want something as simple as adding pure Erlang processes to a supervision tree, making your release OTP compliant beyond the capabilities provided by supervision bridges. For instance, you might have to preserve that proof of concept you wrote when you first started exploring Erlang that, against better judgment, wound up in production.¹.

We refer to a process that can be added to an OTP supervision tree and packaged in an application as a *special process*. This chapter explains how to write your own special processes, providing you with the flexibility of pure Erlang whilst retaining all of the advantages of OTP. We then explain how you can take your special process a step further, turning them into an OTP behaviors by splitting the code into generic and specific modules that interface with each other through predefined callback functions. If you are not planning on implementing your own behaviours or are uninterested in how they work behind the scenes, feel free to jump to the next chapter (or go to the pub) without a bad conscience. You can always come back and reference this chapter when you need to. If, on the other hand, we've piqued your curiosity, keep on reading.

1. For those of you working in large companies, we're referring to the projects where we've spent more time in meetings discussing and trying to get approval for a migration to OTP than what it would have actually taken you to refactor the code.

Special Processes

In order for processes to be called special processes, and as such be part of an OTP supervision tree, they must:

- Be started using the `proc_lib` module and link to their parent.
- Be able to handle system messages, system events, and shutdown requests.
- Return the module list if running dynamic modules, as we did with event managers when defining their child specs.
- While optional, it is useful if they are capable of handling debug flags and generate trace messages.

We will show you how to implement special processes by walking through an example where we implement a mutex, serializing access to critical resources.

The Mutex

Mutex stands for mutual exclusion. It ensures only one process is allowed to execute the code in the critical section at any one time. A critical resource could be a printer, shared memory, or any other device for which requests must be serialized because it can handle only one client at a time. A process executing code that accesses this resource is said to be in the *critical section*. It needs to finish executing all the code in the critical section and exit it before a new process is allowed to enter.

In Erlang, programmers can implement a mutex as a finite state machine, serializing client requests through a process and managing the request queue using the mailboxes and selective receives. Because we are implementing a finite state machine, you must be asking yourself why we are not using the `gen_fsm` behaviour module. The reason is that the `gen_fsm` behaviour, or any of the other standard OTP behaviours, for that matter, do not allow us to selectively receive messages through pattern matching. Instead, they force us to handle events in the order in which they arrive. In contrast, by using the process mailbox and selective receives to manage the queue of client processes waiting for the mutex, we simplify our code because we have to handle only one client request at a time, without having to worry about the others waiting in the queue.

Mutexes are finite state machines with two states, *free* and *busy*. A client wanting to enter the critical section does so by calling the client function `mutex:wait(Name)`, where `Name` is the variable bound to the registered name associated with the mutex. The `wait` call is synchronous, returning only when the calling process is allowed to enter the critical section. When that occurs, the FSM transitions to state *busy*.

Requests are stored in the mailbox and handled on a first in, first out basis. If the mutex is being blocked by another process in state *busy*, the request is left in the mailbox and handled when the mutex returns to state *free*. When the busy process is ready to leave

the critical section, it calls `mutex:signal(Name)`, an asynchronous call that releases the mutex. When that occurs, the finite state machine transitions back to state *free*, ready to handle the next request. [Figure 10-1](#) shows the state transitions of a mutex.

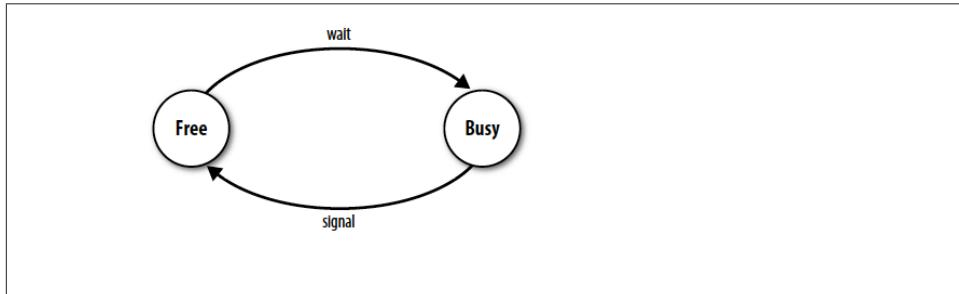


Figure 10-1. State transitions in a mutex

Let's have a look at the `mutex` module, starting with the client functions. Other exported functions will be defined shortly.

```
-module(mutex).
-export([start_link/1, start_link/2, init/3, stop/1]).
-export([wait/1, signal/1]).

wait(Name) ->
    Name ! {wait,self()},
    Mutex = whereis(Name),
    receive      {Mutex,ok} -> ok
    end.

signal(Name) ->
    Name ! {signal,self()},
    ok.
```

Lots of borderline cases are handled gracefully in standard OTP behaviours and are often taken for granted by the programmer. You might have seen them yourself when looking at the code in the `gen_server` or `gen_fsm` modules. When implementing special processes, however, you need to decide which borderline cases to handle and take care of them yourself. In our example, we've opted for simplicity and will not be covering any of them. But to give you an idea of what we are talking about, have a look at the `wait/1` function, where we do not check if `Name` exists. We are not monitoring whether the mutex terminates while the client process is suspended in its `receive` clause. Nor are we handling the case where the mutex terminates right before `whereis/1` and is restarted and re-registered immediately, leaving `wait/1` in a `receive` clause waiting for a message from a live process it will never receive. Nor have we implemented any time-outs in case the mutex process is deadlocked or hanging.

Starting Special Processes

When starting special processes, use the `start` and `spawn` functions defined in the `proc_lib` library module instead of Erlang's standard `spawn` and `spawn_link` BIFs. The `proc_lib` functions store the process name, identity, parent, ancestors, and initial function call in the process dictionary. If the process terminates abnormally, SASL crash reports are generated and forwarded to the error logger. They contain all the process info stored at startup, together with the reason for termination. And like other behaviours, there is functionality allowing for a synchronous start-up with an init phase.

A common error is to attach to the supervision tree a process that doesn't implement a behaviour. There are no warnings at compile time or run-time for this, as the only check made by the supervisor is to ensure the tuple `{ok, Pid}` is returned. No checks are made on `Pid` either. You will notice things going wrong only after a crash, restart, or upgrade. And because these processes do not follow standard behaviours, unless you've tested your restart strategy, hunting down the issue will resemble more of a wild goose chase than a routine and civilized troubleshooting session. For non OTP compliant processes, use supervisor bridges covered in “[Supervisor bridges](#)” on page 185.

Basic template for starting a special process

The recommended approach to starting a special process is to use the `proc_lib:start_link(Mod, Fun, Args)` call. Given a module, function, and a list of arguments, it synchronously spawns a process and waits for this process to notify that it has correctly started through the `proc_lib:init_ack(Value)` call. `Value` is sent back to the parent process, becoming the return value of the `start_link/3` call. Note how we are passing optional `DbgOpts` debug option parameters in our `start_link` call. We covered them in [Chapter 5](#). For now, assume `DbgOpts` is an empty list. Note also how we are passing the Parent process ID to the `init/3` function; we need it in our main loop. It is the result of the `self()` BIF in the `start_link/2` call.

```
start_link(Name) ->
    start_link(Name, []).

start_link(Name, DbgOpts) ->
    proc_lib:start_link(?MODULE, init, [self(), Name, DbgOpts]).

stop(Name) -> Name ! stop.

init(Parent, Name, DbgOpts) ->
    register(Name, self()),
    process_flag(trap_exit, true),
    Debug = sys:debug_options(DbgOpts),
    proc_lib:init_ack({ok, self()}),
    free(Name, Parent, Debug).
```

When initializing the process state, we first register the mutex with the alias `Name`. We set the `trap_exit` flag so we can receive exit signals from processes in our linked set (We use links instead of monitors to notify or terminate the caller if the mutex fails). And finally, we initialize the debug trace flags using the `sys:debug_options/1` call. The return value of `debug_options/1` is passed as loop data and stored in the process state. It will be needed whenever the special process has to generate a trace message or receives a system message requesting it to update its trace flags.

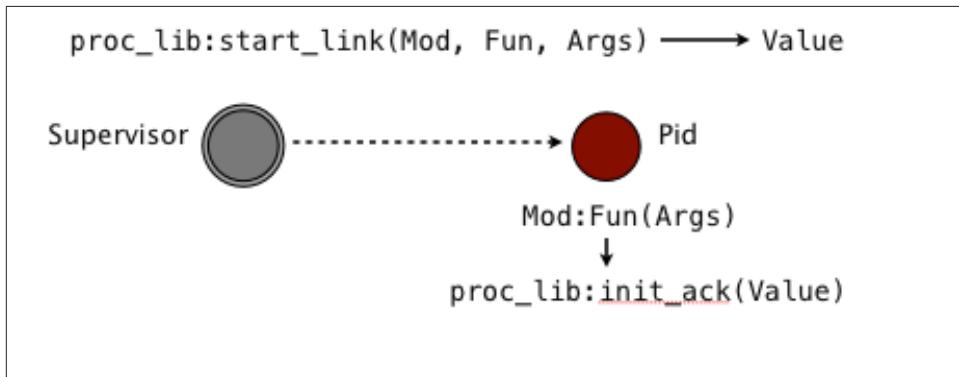


Figure 10-2. Starting Special Processes

As seen in [Figure 10-2](#), once the state is initialized, we call `proc_lib:init_ack(Value)` to inform the parent that the special process has started correctly. `Value` is sent back and becomes the return value of the `proc_lib:start_link/3` call. Although it isn't mandatory, it is common practice to return `{ok, self()}` because supervisors expect their children's start functions to return `{ok, Pid}`. If any part of the initialization fails before calling `init_ack/1`, `proc_lib:start_link/3` terminates with the same reason. Have a look at the last line of the `init/3` function and differentiate between the function call `free`, which points to the FSM's first state and `Name`, `Parent` and `Debug`, which is the process state.

The calls you can use to synchronously start a special process are:

```

proc_lib:start(Module, Function, Args)
proc_lib:start(Module, Function, Args, Time)
proc_lib:start(Module, Function, Args, Time, SpawnOpts) -> Ret
proc_lib:start_link(Module, Function, Args)
proc_lib:start_link(Module, Function, Args, Time)
proc_lib:start_link(Module, Function, Args, Time, SpawnOpts) -> Ret
proc_lib:init_ack(Ret)
proc_lib:init_ack(Parent, Ret) -> ok
  
```

As with other behaviours, `SpawnOpts` is a list containing all options the spawn BIFs accept, `monitor` excluded. If within `Time` milliseconds `init_ack` is not called, the start function returns `{error, timeout}`. If you use `spawn` or `spawn_opt`, do not forget to link the child to the parent process, either through the `link/1` BIF or by passing the `link` option in `SpawnOpts`.

Asynchronously starting a special process

The following variations on the standard `spawn` and `spawn_link` functions are used in situations where you need asynchronous starts, such as the simultaneous launch of hundreds of new processes. They spawn the child process and immediately return its Pid:

```
proc_lib:spawn(Fun)
proc_lib:spawn_link(Fun)
proc_lib:spawn_opt(Fun, SpawnOpts) -> Pid
proc_lib:spawn(Module, Function, Args)
proc_lib:spawn_link(Module, Function, Args)
proc_lib:spawn_opt(Node, Function, SpawnOpts) -> Pid
```

Other options to synchronously start special servers include spawning a process using a fun and spawning a process with the `spawn` options `SpawnOpts`.

Use asynchronous spawning with care, because the functions might cause multiple processes to run in parallel resulting in race conditions that make your program non-deterministic. The same arguments we put forward in “[Starting a Server](#)” on page 73 when discussing generic servers are valid here. A startup error might be hard to reproduce if it is dependent on a certain number of concurrent events happening in a specific order, an issue that is becoming more evident with multi-core architectures. To be able to deterministically reproduce a startup error, create your process synchronously.

Regardless of how you start your special process, they have to always be linked to their parent (by default, the supervisor). This happens automatically if you use `start_link`, `spawn_link`, or pass the `link` option in `SpawnOpts`. No checks are made to ensure that the process is actually linked to the supervisor, so even here, omissions of this type can be difficult to troubleshoot and detect.

The Mutex States

As we saw, a mutex has two states, free and busy, that are implemented as tail recursive functions. The synchronous `wait` and asynchronous `signal` events are sent as messages together with the client Pid. The combination of state and event dictates the actions and state transition. Note how when in the free state, we accept only the `wait` event, informing the client through the message `{self(), ok}` that it is allowed to enter the critical section. The mutex will then transition to the busy state, where the only event

that will pattern match is *signal* sent by `Pid`. You should have noticed that `Pid` was bound in the function head to the client holding the mutex. Upon receiving the *signal* event, the mutex transitions back to the free state.

```
free(Name, Parent, Debug) ->
  receive {wait,Pid} -> Pid ! {self(),ok}, busy(Pid, Name, Parent, Debug); stop ->
    end.

busy(Pid, Name, Parent, Debug) ->
  receive {signal,Pid} -> free(Name, Parent, Debug)
  end.
```

Note how we accept the *stop* message only if the mutex is in the free state. If you stop the mutex in the busy state, you'll leave the client executing the code in its critical section in an unknown and possibly corrupt state, because the mutex might have been restarted and blocked by other client processes. By stopping the mutex only in the free state, you can guarantee a clean shutdown.

So far, so good. We are going back to Erlang 101 with the basics of FSMs. Let's now start expanding the states to handle the system messages required by special processes.

Handling Exits

If the parent of your special process terminates, your process must terminate as well. If your process does not trap EXIT signals, the run-time will take care of this for you because you should be linked to your parent. Non-normal EXIT signals propagate to all processes in the link set, terminating them with the same reason that terminated the original process. An EXIT with reason `normal` doesn't propagate, but in OTP, the supervisor guarantees that a parent will never terminate with that reason, so you don't have to worry about it.

Special processes that trap exits have to monitor their parents, as they might receive messages of the format

```
{'EXIT', Parent, Reason}
```

where `Parent` is the parent pid and `Reason` is the reason for termination. If they do, they should clean up after themselves, possibly in their `terminate` or `clean-up` function, followed by a call to the `exit(Reason)` BIF.

In our previous example, the mutex is trapping exits, so we have to monitor parent termination. Let's expand the state functions, handling the EXIT messages from the parent process by calling `terminate/2`. We will also call `terminate/2` when receiving the `stop` message.

```
free(Name, Parent, Debug) ->
  receive {wait,Pid} ->
    link(Pid), Pid ! {self(),ok}, busy(Pid, Name, Parent, Debug); stop ->
      terminate(Name, Parent, Debug).
    end.
```

```

busy(Pid, Name, Parent, Debug) ->
    receive {signal,Pid} ->     free(Name, Parent, Debug); {'EXIT',Parent,Reason} ->      exit(P
        end.
```

```

terminate(Reason, Name) ->
    unregister(Name),
    terminate(Reason).
```

```

terminate(Reason) ->
    receive {wait,Pid} ->     exit(Pid, Reason),      terminate(Reason)
    after 0 ->      exit(Reason)
    end.
```

The first thing `terminate/2` does is unregister the mutex, ensuring that any processes that try to send it requests terminate with reason `badarg`. The mutex goes on to terminate all processes in the queue by traversing its mailbox and extracting wait requests. When done, it knows no client processes are kept hanging and terminates itself with reason `Reason`.

System Messages

In addition to monitoring parents, special processes need to manage system messages of the format

```
{system, From, Msg}
```

where `From` is the request originator and `Msg` is the system message itself. They could be messages originating from the supervisor used to suspend and resume processes during software upgrades or from a client manipulating or retrieving trace outputs using the `sys` module. What they are, however, is irrelevant to you as a developer, as you handle them as opaque data types and just pass them on.

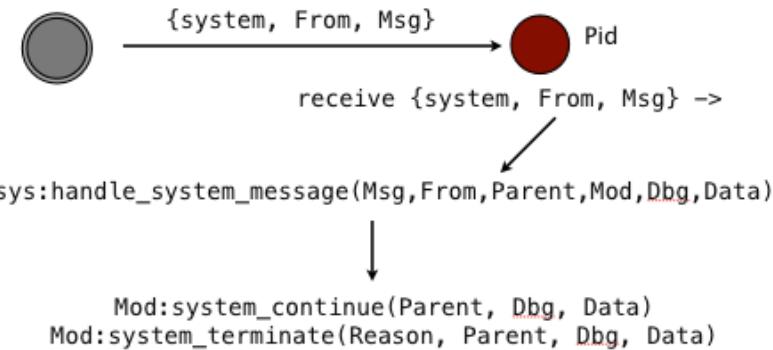


Figure 10-3. Handling System Messages

No matter what the request is, these calls are handled behind the scenes in the `sys:handle_system_message(Msg, From, Parent, Mod, Dbg, Data)` function. The arguments to the `sys:handle_system_message/6`, although numerous, are straightforward:

- `Msg` and `From` are provided by the system message
- `Parent` is the parent pid, passed when spawning the special process
- `Mod` is the name of the module implementing the special process
- `Dbg` is the debug data, initially returned by `sys:debug_options/1` call
- `Data` is used to store the loop data of the process

The functions in the special process module that executes the call should be tail recursive as they never return. Not making them tail recursive will cause a memory leak every time a system message is received. Control is handed back to the special process in the `Mod` module by calling one of the following callback functions:

```

Mod:system_continue(Parent, Debug, Data)
Mod:system_terminate(Reason, Parent, Debug, Data)

```

If control is returned through the `system_continue/3` callback function, your special process needs to return to its main loop. If `system_terminate/4` is instead called, probably as a result of the parent ordering a shutdown, the special process needs to clean up after itself and terminate with reason `Reason`. We are going to show you all of this in the mutex example, but first, let's understand how debug printouts work.

Trace and Log Events

When we covered the start functions earlier in this chapter, we discussed the `Spaw n0pts` argument, which among other options, allows us to pass debug flags to special processes. In our `mutex:start_link/2` call, we can pass these debug options in the second argument, binding them to the `DbgOpts` variable. `DbgOpts` contains zero or more of the `trace`, `log`, `statistics` and `{log_to_file, FileName}` flags described in [Chapter 5](#). This list is passed by the special process to the `sys:debug_options(DbgOpts)` call, which initiates the debug routines. Unrecognized or unsupported debug options are ignored. The return value of the call, stored in the variable `Debug` in our example, is kept in the special process loop data passed to all system calls. Remember the example in “[Tracing and Logging](#)” on page 95 where we turned the trace and logs on or off during runtime, printing them in the shell and diverting them to file? If everything is initialized correctly, you can generate the similar trace logs with your special processes, turning the options on and off at run-time. All requests originating from calls such as `sys:trace/3` or `sys:log/2` are received and handled as system messages. What might change in between calls are the contents of the `Debug` list, returned as part of the `system_continue/3` callback function.

Generating trace events is a straightforward operation done by calling

```
sys:handle_debug(Debug, DbgFun, Extra, Event)
```

where

- `Debug` are the initialized debug options
- `DbgFun` is a fun of arity 3 that formats the trace event
- `Extra` is data that can be used when formatting the event, usually the process name or the loop data
- `Event` is the trace event you want to print out

`DbgFun` is a fun that formats the event, sometimes by calling another function to do so. The arguments passed to it by the `sys` module include the I/O device you are writing to, which can be either the `standard_io` or `standard_error` atom or the pid returned by the `file:open` call. `Extra` and `Event` come from the arguments to the `handle_debug/4` call:

```
fun(Dev, Extra, Event) ->
    io:format(Dev, "mutex ~w: ~w~n", [Extra,Event])
end
```

You can also add your own trace functions at run time using the `sys:install/2` call, using pattern matching in the fun head to examine events and decide on the flow of execution. With system messages and trace outputs in place, let's see how it all fits together by adding them to our mutex example.

Putting it Together

For your convenience, we've put the whole mutex example in one place. Note how we've expanded the *free* and *busy* states to include trace messages and system messages. Let's focus on this functionality, starting with trace messages.

When we receive the *wait* and *signal* events, we call `sys:handle_debug(Debug, fun debug/3, Name, Event)`, where Event is either `{wait, Pid}` or `{signal, Pid}`. This call hands control over to the `sys` module, which eventually calls the `debug` fun. In our case, it is the local function `debug/3`. Have a look at it, paying special attention as to how the I/O device, extra arguments and event passed to it are used. `handle_debug/4` returns `NewDebug`, which is passed as an argument to the next state. When reviewing the example, remember the mutex process does not implement the services it protects. It just implements the semaphore which gives other processes access to these services.

```
-module(mutex).

-export([start_link/1, start_link/2, init/3, stop/1]).
-export([wait/1, signal/1]).
-export([system_continue/3, system_terminate/4]).

wait(Name) ->
    Name ! {wait,self()},
    Mutex = whereis(Name),
    receive      {Mutex,ok} -> ok
    end.

signal(Name) ->
    Name ! {signal,self()},
    ok.

start_link(Name) ->
    start_link(Name, []).

start_link(Name, DbgOpts) ->
    proc_lib:start_link(?MODULE, init, [self(), Name, DbgOpts]). 

stop(Name) -> Name ! stop.

init(Parent, Name, DbgOpts) ->
    register(Name, self()),
    process_flag(trap_exit, true),
    Debug = sys:debug_options(DbgOpts),
    proc_lib:init_ack({ok,self()}),
    free(Name, Parent, Debug).

free(Name, Parent, Debug) ->
    receive      {wait,Pid} ->          %% The user requests.      NewDebug = sys:handle_debug(Debug,
    Pid ! {self(),ok},      busy(Pid, Name, Parent, NewDebug);  {system,From,Msg} ->  %% The
    end.
```

```

end.

busy(Pid, Name, Parent, Debug) ->
    receive {signal,Pid} ->      NewDebug = sys:handle_debug(Debug, fun debug/3, Name, {signal,Pi
end.

terminate(Reason, Name) ->
    unregister(Name),
    terminate(Reason).
terminate(Reason) ->
    receive {wait,Pid} ->      exit(Pid, Reason),      terminate(Reason)
    after 0 ->      exit(Reason)
end.

debug(Dev, Event, Name) ->
    io:format(Dev, "mutex ~w: ~w~n", [Name,Event]).

system_continue(Parent, Debug, {busy, Pid, Name}) ->
    busy(Pid, Name, Parent, Debug);
system_continue(Parent, Debug, {free, Name}) ->
    free(Name, Parent, Debug).

system_terminate(Reason, _Parent, _Debug, {busy, Pid, Name}) ->
    exit(Pid, Reason),
    terminate(Reason, Name);
system_terminate(Reason, _Parent, _Debug, {free, Name}) ->
    terminate(Reason, Name).

```

When the `free` and `busy` functions receive `{system, From, Msg}`, they tail recursively invoke `sys:handle_system_msg(Msg, From, Parent, ?MODULE, Debug, {State, LoopData})`, handing control over to the `sys` module. The system message is handled behind the scenes, after which the function returns by calling `system_continue/3` or `system_terminate/4` in the `mutex` module. If the function is not tail recursive, there will be a memory leak for every system message received.

In our example, if `system_continue` is called, we just return to the state we were in, determined by the `Name` loop data in state `free` and the `{Name, Pid}` loop data in `busy`, where we wait for the next event or system call. In the case of `system_terminate`, if in state `busy`, we terminate the process that held the mutex (potentially leaving the system in an inconsistent state), followed by `terminate/2`. If in state `free`, we just call `terminate/2`. In both cases, we employ pattern matching on the final argument to ensure we take the correct actions for continuation and termination.

System messages and debug options are straightforward to handle in your own special processes. All you need to do is reuse the code from this example, ensuring that when you get handed back the control, you go back into your loop or state with a tail recursive function. Before looking at the trial run of the mutex, read through the code one more time and make sure you understand the what, why, and hows of special processes.

In our trial run, we create a child specification for our special process, starting it as a dynamic child in a supervisor `mutex_sup`. We've not included the supervisor code in this example, it is boilerplate code. All `init/1` does is return the supervisor specification with a restart tuple with a `one_for_one` strategy allowing a maximum of five restarts per hour and an empty child list. You can find the source code in the book's github repository.

Note how in the `mutex:start_link/2` arguments of the child specification, we turn on the trace flag. This leads to the trace printout when the mutex is started as a result of shell command 3. We turn on other debug options using the `sys` module in shell commands 4 and 5.

```

1> ChildSpec = {mutex, {mutex, start_link, [printer, [trace]]}},
   transient, 5000, worker, [mutex]}.
{mutex,{mutex,start_link,[printer,[trace]]},
 transient,5000,worker,
 [mutex]}
2> mutex_sup:start_link().
{ok,<0.35.0>}
3> supervisor:start_child(mutex_sup, ChildSpec).
mutex printer: init
{ok,<0.37.0>}
4> sys:log(printer, {true,10}).
ok
5> sys:statistics(printer, true).
ok
6> mutex:wait(printer), mutex:signal(printer).
mutex printer: {wait,<0.32.0>}
mutex printer: {signal,<0.32.0>}
ok
7> sys:log(printer, get).
{ok,[{{wait,<0.32.0>},printer,#Fun<mutex.1.94496536>},
 {{signal,<0.32.0>},printer,#Fun<mutex.2.94496536>}]}
8> sys:log(printer, print).
mutex printer: {wait,<0.32.0>}
mutex printer: {signal,<0.32.0>}
ok
9> sys:get_status(printer).
{status,<0.37.0>,
 {module,mutex},
 [{{'$ancestors',[mutex_sup,<0.32.0>]},
   {'$initial_call',{mutex,init,3}}},
  {running,<0.35.0>,
   [{statistics,{{2014,1,6},{8,50,36}},{{reductions,66},0,0}}},
    {log,{10,
         [{{{signal,<0.32.0>},printer,#Fun<mutex.2.94496536>},
           {{wait,<0.32.0>},printer,#Fun<mutex.1.94496536>}}]}},
     {trace,true}],
    {free,printer}}]}
10> exit(whereis(printer), kill).
mutex printer: init

```

```

true
11> exit(whereis(mutex_sup), shutdown).
mutex printer: {terminate,shutdown}
** exception exit: shutdown

```

In shell command 6, we wait for the mutex and signal for it to be released, both requests generating two trace events. In shell commands 7, 8 and 9, we retrieve some of the trace and status information through the sys module, followed by some tests with termination and restarts in shell commands 10 and 11.

Do some tests of your own, experimenting with multiple clients, the SASL report browser, and other sys commands such as suspending and restarting the modules.

Dynamic Modules and Hibernating

You might recall from [Chapter 8](#) that we need to provide the list of modules implementing the behaviour in the child specification. They are used to determine which processes to suspend during software upgrades. There are occasions, as is the case with event managers and handlers, where the modules are not known at compile time. In the supervisor child specification module list, these behaviours were tagged with the atom `dynamic`. Special processes can also have dynamic modules.

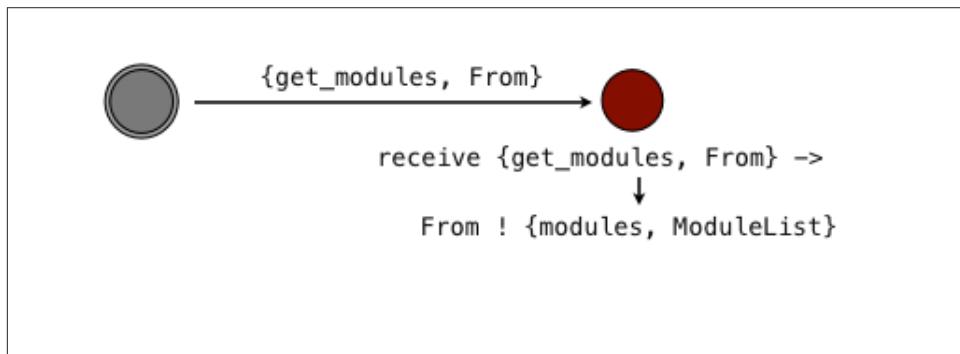


Figure 10-4. Retrieving Dynamic Modules

If your special process modules are tagged as `dynamic` in child specification, you need to handle the system message

```
{get_modules, From}
```

where `From` is the pid of the supervisor, used to return the list of modules in the `From ! {modules, ModuleList}` expression.

If you need to hibernate your special processes, instead of the BIF, you need to use:

```
proc_lib:hibernate(Mod, Fun, Args)
```

It hibernates the process just like the BIF and the standard OTP behaviour return values, but as an added feature, it also ensures that logging and debugging still function when the process wakes up.

Your Own Behaviours

Now that you understand special processes, let's take the concept further by splitting the code into generic and specific parts to implement our own behaviours. You want to implement behaviours when several processes follow a pattern that cannot be expressed using existing OTP behaviours. Generic servers, finite state machines, and event managers cater to most programmers' needs, so don't get caught up in the excitement and start writing new behaviours in every project. Chances are you are over-engineering a solution that could easily be abstracted in a simple library module.

Having said that, there will be times when there are good reasons to implement your own behaviors. Patterns can be abstracted in generic and specific modules, when the generic part is substantial enough to make it worthwhile. If you go down this route, chances are good that your behaviour (or library) can be built on top of generic servers. If not, or if you prefer to avoid generic servers because of the performance overhead, make sure your behaviour follows the design rules required by special processes using the `sys` and `proc_lib` modules.



If you are into software archeology and have an interest in the evolution of software, try to get your hands on the source code of the early versions of Erlang/OTP. Skim through the old behaviour code and you will find that most of the behaviours were built on top of generic servers. Current OTP behaviours, generic servers included, are built using a module called `gen`. It is a wrapper on top of the `sys` and `proc_lib` modules, handling a lot of the tricky and borderline cases associated with concurrent and distributed programming we've discussed in previous chapters. Look for it in the source directory of your `stdlib` application and skim through the code. If you are implementing your own behaviours and do not want to get caught out, you might want to use `gen` instead of rolling our your own. Be warned, however, as it is undocumented, it might change in between releases with little or no notice.

Rules for creating behaviours

The steps to creating your own behaviour are straightforward, requiring you to break up your code into generic and specific modules and define the callback functions and their return values. When doing so, you need to follow these simple rules:

- The name of the generic module has to be the same as the behaviour name.

- You need to list the callback functions in the behaviour module.
- In your callback module, include the `-behaviour(BeaviourName)` directive.

Once you've compiled your generic behaviour code, compiling your callback modules with the behaviour directives will result in warnings should you omit any callbacks.

A Example Handling TCP Streams

Let's have a look at some parts of an example in which we implement our own behaviour, focusing on the code specific to our behaviour's implementation. We've omitted functions not relevant to the example, marking them with `...` in the code. If you want to look at the whole module, you can find it in the code repository with the book examples. There is no need, however, to view the full example if you are interested only in understanding the specifics of implementing your own behaviour.

Our example is a wrapper that encapsulates activities associated with TCP/IP streams, including connections, configuration, and error handling, exposing only the stream of data being received. Upon receiving a socket accept request, the behaviour spawns a new process that is kept alive for as long as the socket is open. The behaviour receives the packets, forwarding them to the callback module as they arrive. The socket can be closed by the callback module through a return value of a callback function, or indirectly when the TCP client closes its side of the connection.

The callback functions in the callback module consist of an initialization function called once when the socket is opened, a data handling call invoked for every packet received, and a termination function called when the socket is closed:

```
-module(tcp_print).
-export([init_request/0, get_request/2, stop_request/2]).
-behaviour(tcp_wrapper).

init_request() ->
    io:format("Receiving Data~n."),
    {ok, []}.
get_request(Data, Buffer) ->
    io:format("."),
    {ok, [Data|Buffer]}.
stop_request(_Reason, Buffer) ->
    io:format("~n"),
    io:format(lists:reverse(Buffer)),
    io:format("~n").
```

The callback function `init_request/0` returns `{ok, LoopData}`. `get_request/2` receives the TCP packet bound to the variable `Data` and the `LoopData`, returning either `{ok, NewLoopData}` or `{stop, Reason, NewLoopData}`. In this example, `LoopData` is a buffer of received TCP packets bound to the variable `Buffer`. Upon closing the socket,

`stop_request/2` is given the Reason for termination and the LoopData, and has to return the atom ok.

Note how we have included the `-behaviour(tcp_wrapper)`. directive in the code. This points to the `tcp_wrapper` module, where the behaviour is implemented.

When starting the `tcp_wrapper` behaviour, we pass the callback module Mod and the Port number. We spawn a process that initializes the behaviour state, opens a listener socket, and eventually makes its way to the `accept/4` function. For every concurrent stream, we accept a connection on the listener socket, spawn a new process that starts executing in the `init_request/2` function, and handle the stream through the callback module. In the `accept` call, we specify a timeout to keep from blocking infinitely so we can yield control back to the main loop (not shown in the example) every second, ensuring we can handle system messages and the EXIT signal from the parent process.

²

```
-module(tcp_wrapper).
-export([start_link/2, cast/3]).
-export([init/3, system_continue/3, system_terminate/4, init_request/2]).

callback init_request() -> {'ok', Reply :: term()}.
callback get_request(Data :: term(), LoopData :: term()) -> {'ok', Reply :: term()} | 
{'stop', Reason :: atom(),
LoopData :: term()}.

callback stop_request(Reason :: term(), LoopData :: term()) -> term().

start_link(Mod, Port) ->
  proc_lib:start_link(?MODULE, init, [Mod, Port, self()]). 

cast(Host, Port, Data) ->
  {ok, Socket} = gen_tcp:connect(Host, Port, [binary, {active, false}, {reuseaddr, true}]),
  send(Socket, Data),
  ok = gen_tcp:close(Socket).

send(Socket, <<Chunk:1/binary,Rest/binary>>) ->
  gen_tcp:send(Socket, [Chunk]),
  send(Socket, Rest);
send(Socket, <<Rest/binary>>) ->
  gen_tcp:send(Socket, Rest).

init(Mod, Port, Parent) ->
  {ok, Listener} = gen_tcp:listen(Port, [{active, false}]),
  proc_lib:init_ack({ok, self()}),
  loop(Mod, Listener, Parent, sys:debug_options([])).
```

2. An alternative to this timeout approach is to use the `prim_inet:async_accept/2` function, which sends the calling process a message when a new connection is accepted, but that function is intended to be private to Erlang/OTP and so is not part of its documented and supported set of API functions.

```

loop(Mod, Listener, Parent, Debug) ->
    receive {system, From, Msg} -> sys:handle_system_msg(Msg, From, Parent, ?MODULE, Debug, [_]);
    {'EXIT', Parent, Reason} -> terminate(Reason, Listener, Debug);
    {'EXIT', Child, _Reason} -> NewDebug = sys:handle_debug(Debug, fun debug/3, stop_request, Child),
        after 0 -> accept(Mod, Listener, Parent, Debug)
    end.

accept(Mod, Listener, Parent, Debug) ->
    case gen_tcp:accept(Listener, 1000) of {ok, Socket} -> Pid = proc_lib:spawn_link(?MODULE,
        [Mod, Socket], init_request),
    end.

system_continue(Parent, Debug, {Listener, Mod}) ->
    loop(Mod, Listener, Parent, Debug).

system_terminate(Reason, _Parent, Debug, {Listener, _Mod}) ->
    terminate(Reason, Listener, Debug).

terminate(Reason, Listener, Debug) ->
    sys:handle_debug(Debug, fun debug/3, terminating, Reason),
    gen_tcp:close(Listener),
    exit(Reason).

debug(Dev, Event, Data) ->
    io:format(Dev, "Listener ~w:~w~n", [Event, Data]).

init_request(Mod, Socket) ->
    {ok, LoopData} = Mod:init_request(),
    get_request(Mod, Socket, LoopData).

get_request(Mod, Socket, LoopData) ->
    case gen_tcp:recv(Socket, 0) of {ok, Data} -> case Mod:get_request(Data, LoopData) of
    end.

stop_request(Mod, Reason, LoopData) ->
    Mod:stop_request(Reason, LoopData).

```

The generic code handling the TCP stream is straightforward. It is a process loop that initializes the stream state, receives the packets, and terminates when the callback module returns a `stop` tuple, or when the TCP client decides to close its side of the connection. For initialization, receiving packets, and termination, appropriate callback functions in the `Mod` callback module are called.

One item that stands out in our behaviour implementation—probably the most important one alongside the calling of the callback functions—is the callback specification. It lists the callback functions that need to be exported in the callback module, following the directives set out in the Erlang type and function specifications. The callback specifications are mapped to the `behaviour_info(callbacks)` function which returns a list of the form `{Function, Arity}`. You can bypass the callback specifications altogether, directly implementing and exporting the `behaviour_info/1` call in your generic behaviour module (which is how behaviours were required to be implemented with older

releases of Erlang/OTP prior to R15B). Compare the callback specifications to the callback functions in the `tcp_print` module. Do they match?

```
-module(tcp_wrapper).  
...  
-export([behaviour_info/1]).  
  
behaviour_info(callbacks) ->  
    [{init_request, 0}, {get_request, 2}, {stop_request, 2}].  
...
```

The advantages of using callback specifications over the `behaviour_info/1` function is that the dialyzer tool will find discrepancies between your callback modules and the specs, a welcome addition to the undefined callback function compiler warnings. The dialyzer enables behaviour callback warnings by default. Remember to compile your generic behaviour module and make it available in the code search path before compiling your callback module, or else you will get an *undefined behaviour* warning.

You might have noticed that we are using the British spelling when adding the behaviour directive in the callback module. American chums, don't despair. When defining your behaviour directives, both the American "behavior" and British "behaviour" spellings are honoured:

```
-behavior(tcp_wrapper).  
-behaviour(tcp_wrapper).
```

The same applies when defining your `behaviour_info/1` callback function. Many moons ago, if you do not stick to the British spelling, swallowing your pride and forcing yourself to type in that extra letter, you would get an unknown behaviour warning when compiling your callback module. Many have been caught out and spent endless hours trying to figure out the problem and resolve it.

Summing Up

In this chapter, we've introduced you to the ins and outs of implementing special processes, making them OTP compliant and including them as part of OTP supervision trees. We've taken special processes a step further, allowing you to split the code into generic and specific modules, turning them into behaviours complete with callback modules, behaviour directives and associated compiler warnings.

When starting and hibernating special processes, instead of the standard BIFs, you must use the following functions in the `proc_lib` module:

Table 10-1. starting special process with the proc_lib module

function call	callback function
proc_lib:spawn_link/1,2,3,4	none
proc_lib:spawn_opt/2,3,4,5	none
proc_lib:start/3,4,5	proc_lib:init_ack(Parent, Reply), proc_lib:init_ack(Reply)
proc_lib:start_link/3,4,5	proc_lib:init_ack(Parent, Reply), proc_lib:init_ack(Reply)
proc_lib:hibernate/3	none

The following system message calls and their respective callbacks need to be managed by your process, by either responding directly to the process sending the request or by using the `sys` module.

Table 10-2. system requests and messages

message	callback function or action
{system, From, Request}	Mod:system_continue(Parent, Debug, LoopData) Mod:system_terminate(Reason, Parent, Debug, LoopData)
{'EXIT', Parent, Reason}	exit(Reason)
{get_modules, From}	From ! {modules, ModuleList}

You can read more about the `sys` and `proc_lib` modules in their respective manual pages. There is an example covering special processes and user-defined behaviours in the `sys` and `proc lib` section of the OTP Design Principles User's Guide. And finally, you can find more information on type and function specifications used in defining your own callback definitions in the Erlang Reference Manual and User's Guide.

If you feel like coding, we suggest you download the `mutex` example from the book's code repository and implement some of the edge cases which can occur in concurrent applications. In your client function, when requesting the mutex, add references guaranteeing the validity of your reply together with optional timeouts. You will also want to monitor the mutex in case it terminates abnormally whilst you are executing in the critical section.

What's Next?

Special processes and user-defined behaviours are the foundations used to build existing and new behaviours allowing us to glue them together in a supervision tree and package them in an application. In the next chapter on release handling and system principles, we group applications in a release, and see how we can configure, start and stop an Erlang node as a whole.

System Principles and Release Handling

Now that we know how to implement and use existing OTP behaviours, organize them in supervision trees with special processes, and package them in applications, the time has come to group these applications together into an Erlang node, starting it as one unit. We do this by creating a *release*, where a system consists of one or more possibly different releases. The underlying Erlang runtime system does not differentiate between user-defined applications and applications that come as part as the Erlang/OTP distribution, treating them in the same manner. Because of this, it should not come as a surprise that any Erlang releases you start with the `erl` command are created with the same tools, structure, and principles you use when defining your own releases. An Erlang node consists of a set of loosely coupled applications. What differs between the standard release and the ones you create yourself are the applications that are loaded and started together, along with their configuration parameters. In this chapter, we walk you through the creation of a target release, explaining how it all hangs together.

System Principles

A release in Erlang is defined as a standalone node consisting of:

- A set of OTP applications written or reused as part of the project, typically containing the system business logic. The applications can be proprietary, open source, or a combination thereof.
- The OTP applications from the standard distribution that the aforementioned applications depend on.
- A set of configuration and boot files, together with a start script.
- The Erlang runtime system, including a copy of the virtual machine.

There are tools which help you create and package a standalone node, but before introducing them, we will cover all of the components in detail and step you through a

build manually. It will help you better understand how it a release is structured and works, and what options you have available.

The simplest way to start an Erlang node is using the `erl` command. You could start your program from the Erlang shell itself by typing in the module and function name or by passing the `-s` flag to `erl`:

```
$ erl -s module function arg1 arg2 ...
```

The *function* and arguments are optional. If only the module is listed, the command will invoke `module:start()`. If the module and function are listed, the command will invoke `module:function()`. We refer to this method of starting your node as a *basic target system*, where you create a Unix shell script that initializes your state and calls the `erl -s` command. This approach should be used only when coding, for basic proofs of concepts or quick hacks. Using basic target systems in production is not recommended, as you lose a lot of the benefits that come with OTP. There are better alternatives.

The next best step to starting your node is a *simple target system*. It makes use of a boot script and tools shipped with the sasl application, facilitating controlled software upgrades at run-time. To understand how a simple target systems work, let's start by examining your Erlang installation and investigating its directory structure and all the files and scripts associated with it. You need to create some of these files yourself when generating the release, using tools such as `systools`, `reltool`, or `relx`, while you can just copy other files from a repository or the installation in your target environment.

Start by finding the top-level directory, often called the Erlang root directory. It is the location where you (or the scripts you used) installed Erlang. If you don't know that location, start an Erlang node and call `code:root_dir()`.

```
$ erl
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V6.1  (abort with ^G)
1> code:root_dir().
"/usr/local/lib/erlang"
2> q().
ok
$ ls
EslErlangUpdater.app          erts-5.7.5           man
Install                      erts-5.8.1           misc
bin                          erts-6.1            releases
doc                          esl_otp_version    usr
erts-5.10.1                  lib
```

The contents of the directory are the output of creating the release. They vary depending on how (and from where) you installed Erlang, the number of upgrades you've done throughout the years, and the customizations made by those who built the release. There is, however, a set of basic files and directories that are required and will always be there, appearing with your first installation.

Release Directory Structure

In this section we explore the files needed for a release. Your own releases will have the same directories and file structures as the Erlang root, so we spend sometime looking at that. The only difference between the root and your own releases are the applications that are loaded and started, their versions, and the version of the runtime system. This becomes evident in the next few sections, where we create our own base station controller release that follows these very principles.

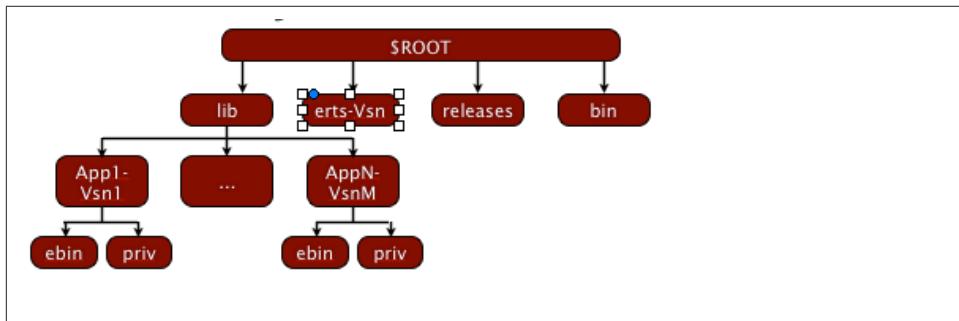


Figure 11-1. Release Directory Structure

Four directories are mandatory in every OTP release, shown in [Figure 11-1](#). We have already looked at *lib*, which contains all of the applications with their version numbers appended to the directory name. You rummaged through it in “[The Application Structure](#)” on page 198 when reading about applications and their directory structures. After upgrades, you could end up with multiple versions of a single application, differentiated by the version number of the application directory name. With multiple instances, the code search path defined when creating the release usually points to the *ebin* directory of the latest version of the application.

The *erts* directory contains binaries for the Erlang runtime system. Even here, if you have at some point upgraded your installation, you might find multiple instances of the directory, distinguished by the *erts* version number appended to the directory name. In *erts* the most interesting subdirectory is *bin*. It contains executables and shell scripts related not only to the virtual machine, but also to all the tools that can be invoked from the shell. Look around in the directory and you will find the following:

erl

A script or program (depending on the target environment) that starts the runtime system and provides an interactive shell.

erlexec

The binary executable called by the *erl* script.

erlc

A common way to run Erlang-specific compilers. The compiler chosen depends on the extension of the file you are trying to compile.

epmd

The Erlang port mapper daemon. It acts as a name server in distributed Erlang environments, mapping Erlang nodes to IP addresses and port numbers.

escript

Allows you to execute short Erlang programs as if they were scripts, without having to compile them.

start

Starts an embedded Erlang target system in Unix environments. This kind of release runs as a daemon job without a shell window. We will look at embedded target systems in “[Creating a Release Package](#)” on page 269

run_erl

The binary called by *start* to start Unix-based embedded systems, where I/O is streamed to pipes.

to_erl

Connects to the Erlang I/O streams with nodes started by *run_erl* in an embedded target system.

werl

Starts the runtime systems in Windows environments in a separate window from the console.

start_erl

Part of the chain of commands to start embedded target systems, setting the boot and config files in Unix systems. In Windows environments, this is similar to the Unix *start* command previously described.

erlsrv

Similar to *run_erl* but for Windows environments, allowing Erlang to be started without the need for the user to log in.

heart

Monitors the heartbeat of the Erlang runtime system and calls a script if the heartbeat is not acknowledged.

dialyzer

A static analysis tool for beam files and Erlang source code. It finds, among other things, type discrepancies and dead or unreachable code. It should be part of everyone’s build process.

type

Infers variable types in Erlang programs based on how the variables are used. It adds type specifications derived from your source code and provides input data to *dialyzer*.

Programmers use several of the executables listed in the *bin* directory when creating and starting an Erlang release. The ones we list are the most important and most relevant to what we cover in more detail later in this chapter. But the list is nowhere near complete, as the full contents depend on the Erlang/OTP version and operating system you are running.

These contents of the *erts-version/bin* directory are similar to those of *bin* in the Erlang root directory. The version-specific directory contains links and copies to the scripts and executables of the *bin* directory of the Erlang runtime version you start by default. This directory is needed because you might have several versions of a release installed and running at any one time. Although typing *erl* would point to the script in the *bin* directory, environment variables would redirect it to the *erts-version/bin* version you are using. Let's have a look at the contents of the *erl* script. With release 17.1 on a Mac running OS X Mavericks, it looks like this:

```
#!/bin/sh
#
# %CopyrightBegin%
#
# Copyright Ericsson AB 1996-2012. All Rights Reserved.
#
# The contents of this file are subject to the Erlang Public License,
# Version 1.1, (the "License"); you may not use this file except in
# compliance with the License. You should have received a copy of the
# Erlang Public License along with this software. If not, it can be
# retrieved online at http://www.erlang.org/.
#
# Software distributed under the License is distributed on an "AS IS"
# basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
# the License for the specific language governing rights and limitations
# under the License.
#
# %CopyrightEnd%
#
ROOTDIR="/usr/local/lib/erlang"
BINDIR=$ROOTDIR/erts-6.1/bin
EMU=beam
PROGNAME=`echo $0 | sed 's/.*/\///'`
export EMU
export ROOTDIR
export BINDIR
export PROGNAME
exec "$BINDIR/erlexec" ${1+"$@"}
```

`ROOTDIR` and `BINDIR`, along with other environment variables, are set when installing or upgrading Erlang. Note how `BINDIR` points to the `$ROOTDIR/erts-6.1/bin` directory we inspected at the beginning of this section and ends up executing `erlexec`. Look in the `erts-6.1/bin` directory for `erl.src` and you will find the file used to create the `erl` script. Similar src files exist for `start_erl` and `start`. We cover `src` files in “[Creating a Release](#)” on [page 259](#) later in this chapter.

If you enter the `releases` directory, also located in the Erlang root directory, you will find a subdirectory for every release you’ve installed on your machine. There should be a one-to-one mapping to the `erts` directories, because most new releases come with a new version of the runtime system. Inspect the contents of any of these `start_erl.data` files and you will see two numbers, the first referring to the emulator version used in the current installation of Erlang and the second referring to the directory of the OTP release being used. If you enter in that directory (we’ve picked version 17 in our example), you will find files with `rel`, `script` and `boot` extensions. Files with the `rel` suffix list the versions of the applications and runtime system for a particular release. The boot file is a binary representation of the `script` file, which contains commands to load and start applications when the system is first started. Enter the subdirectory of the latest release and look at any of the `rel` and `script` files to get a feel for what they might do. We create our own scripts in an upcoming section.

```
$ cd releases/
$ ls
17 R14B RELEASES start_erl.data
R13B04 R16B RELEASES.src
$ more start_erl.data
6.1 17
$ ls 17/
OTP_VERSION          start_all_example.rel
installed_application_versions start_clean.boot
no_dot_erlang.boot    start_clean.rel
no_dot_erlang.rel     start_clean.script
no_dot_erlang.script  start_sasl.boot
start.boot            start_sasl.rel
start.script          start_sasl.script
```

You can override the default location of the `releases` directory specifying the SASL application environment variable `releases_dir` or the OS environment variable `RELDIR`. The Erlang runtime system must have write permissions to this directory for upgrades to work, as it updates the `RELEASES` file in conjunction with upgrades.

Release Resource Files

All your project’s OTP applications, including those that come as part of the standard distribution, and including any proprietary applications as well as open source ones, are bundled up in a release specification containing their versions. This specification also includes the system release version and name, together with the version of the

runtime system. The build system uses this information to do sanity checks, create the boot files, and create the target directory structure.

The minimal (and default) release consists of the *kernel* and *stdlib* applications, but most releases also include and start *sasl* because it contains all of the tools required for a software upgrade. You might not think about upgrades when creating your first release, but you'll probably need to do so at a later date. You are given the option of including *sasl* by default when installing Erlang from source, but if you are using third-party binaries, this choice will have already been made for you.

Let's look at the *rel* files more closely. If, from the Erlang root directory, you enter into the *releases* directory and from there to any of the subdirectories, you will find at least one file with the *rel* suffix. As an example, we've picked the *releases/17/start_sasl.rel* file, stripping out the comments:

```
{release, {"Erlang/OTP", "17"}, {erts, "6.1"},  
 [{kernel, "3.0.1"},  
  {stdlib, "2.1"},  
  {sasl, "2.4"}]}.
```

As we can see, this release will run emulator version 6.1, starting kernel version 3.0.1, *stdlib* version 2.1, and *sasl* version 2.4. The name of the release is “Erlang/OTP” and its version is “17”. Other examples and versions of the *rel* files and corresponding boot and script files in the directory specify how other systems are grouped together.

Let's create a release file named *basestation.rel* to use in our base station controller example. The release name is “*basestation*” and we've given it version “1.0”. Along with the standard included applications, we'll include version 1.0 of *bsc*. It is fairly straightforward and differs very little from the previous example:

```
{release,  
 {"basestation", "1.0"},  
 {erts, "6.1"},  
 [{kernel, "3.0.1"},  
  {stdlib, "2.1"},  
  {sasl, "2.4"},  
  {bsc, "1.0"}]}.
```

The resource file is by convention named *ReleaseName.rel*. Following this convention is not mandatory but doing so makes life easier for those supporting and maintaining your code. The resource file contains a tuple with four elements: the *release* atom, a tuple of the format *{ReleaseName, RelVersion}*, a tuple of the format *{erts, ErtsVersion}*, and a list of tuples containing information about the applications and their versions. The application tuples we've seen so far were of the format *{Application, AppVersion}*, but as the following shows, other formats exist as well.

```
{release,  
 {ReleaseName, RelVersion},  
 {erts, ErtsVersion},
```

```
[{Application, AppVersion},  
 {Application, AppVersion, Type},  
 {Application, AppVersion, IncludedAppList},  
 {Application, AppVersion, Type, IncludedAppList}]  
].
```

All of the versions for the various elements in the tuple are strings. In your application tuple, you can also add a Type as one of the following values:

load

Load the application but do not start it.

none

Load the modules in the application, but not the application itself.

permanent

Shuts the node down when the top level supervisor terminates. When the application terminates, all other applications are cleanly taken down with it. This is the default chosen if no restart type is specified.

transient

Shuts down the node when the top level supervisor terminates with a non-normal reason. This is useful only for library applications that do not start their own supervision trees, because top-level supervisors will always terminate with the non-normal reason shutdown, yielding the same outcome as a permanent application.

temporary

Applications that terminate, normally or abnormally, are reported in the SASL logs, but do not affect other applications in the release.

Finally, you can specify a list of included applications in `IncludedAppList`. The list must be a subset of the applications specified in the application app file.

Release and Application Versions

An OTP version is a set of specific application versions listed in the rel file which have been tested together with an emulator version. This does not go to say you can not swap and change application and emulator versions, all it says is that they have not been tested together. As the test cases for OTP releases that are part of the repository, there is nothing stopping you from running them yourself with your proprietary applications as part of your development process. An application version is a set of module versions and resources, listed in the *app* file or contained in the *priv* directory.

Starting with OTP R17, Application and OTP versions share the same numbering scheme. They consist of three integers of the format *<Major>. <Minor>. <Patch>* Where major releases include substantial, possibly non-backwards compatible changes. Minor releases are incremented when new functionality is added and Patch is incremented as a result of bug fixes. Incrementing the version of a major release will set the minor and

patch levels to 0, whilst incrementing a minor release will reset the patch level. Tailing 0s are usually removed from the version number, so a version *17.1.0* is equivalent to version *17.1*.

Higher versions, starting with the most significant digit, tend to include features and bug fixes from the less significant versions. Void from backwards incompatible changes and features which have been removed, you can assume that higher versions contain all of the bug fixes and enhancements of the lower versions. Versions can have more than three parts. This refers to branches of a particular release created in order to deliver compatible patches in older releases. There is no limit to how many branched versions you can have. As an example, fixes in application or release version *17.1.3.1* are not guaranteed to be included in *17.2*. They might if the bug was fixed before release *17.2* came out, but you have no guarantee. Pre-releases, also known as release candidates, will have the *-rcVsn* suffix, e.g. *R17-rc1*.

If you are not sure what OTP release you are using, use the `erlang:system_info(otp_release)` BIF. In the releases directory for the release you are running, you will find the `OTP_VERSION` file which contains the OTP version number. You will find this file only in your development environment. If you look for it in your target installation, you will not find it unless you have put it there yourself.

Creating a Release

Having defined what is included in our release, the time has come to create it in a few simple steps, shown in [Figure 11-2](#):

1. Start by creating a binary boot file, which contains the commands required to load modules and start applications.
2. With your boot scripts in place, create a directory structure that includes all application directories, release directories, and if required, the emulator. This packet is target independent, but could be OS and hardware specific. Your directory structure must follow the directives described in [“Release Directory Structure” on page 253](#), making it compatible with the boot file you created.
3. Create a start script defining your configurations, system limits, code search paths, and other system-specific environment variables, including a pointer to the boot file. Your script will be based on the `.src` files you saw in the `bin` directory of the emulator. The scripts will depend on the directory structure you have created and how you want your target system to behave.
4. With the start scripts in place, create a deployment package specific to your target environment. It could be a tar file, a Debian or Solaris package, a Docker container, or any other instance that you can configure and deploy with tools of your choice.

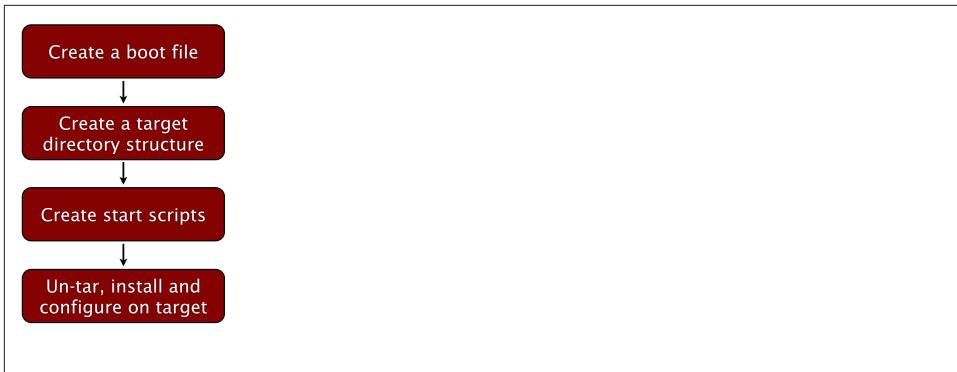


Figure 11-2. Creating an OTP Release

In our example, we create and deploy a tar file using the `systools` that come as part of the OTP distribution. Your typical target directory structure includes all of the applications listed in the release file, and in the majority of cases, the Erlang runtime system. Once we've created our tar file, we will want to untar it and fix scripts, configuration files, and other target specific environment variables before creating the final package. This step could be done manually or as part of your automated build process. It could be done locally on your computer or in your target environment. How you do it depends on the development and target environments as well as and the tools you pick. There never has been, and never will be, a “one size fits all” approach.

Creating the boot file

Let's start by creating our boot file. To do it, we need the `systools:make_script/2` library function. This function creates a binary boot file used by a start script to boot Erlang and your system. To get the `start_script/2` function to work, we need to copy the `bsc` application example, ensuring it follows the directory structure we covered in “[The Application Structure](#)” on page 198. The structure is available in this chapter’s directory of the GitHub repository. If you download it and recreate the example on your computer, don’t forget to compile the Erlang files, placing them into the `ebin` directory.

The script starts off by looking for the application versions specified in the `basestation.rel` file. It does so using the code search path, and any other paths you might have included in your `{path, PathList}` environment variable. In our example, assuming we started Erlang in the same directory as the `bsc` directory, we would use the `[{path, ["bsc/ebin"]}]` option or start Erlang using `erl -pa bsc/ebin`. Remember, `PathList` is a list of lists, so even if you have only one directory, the directory must be defined in a list: `[Dir]`. Let's try it out:

```

1> systools:make_script("basestation", [{path, ["bsc/ebin"]}]). 
Duplicated register names:
    overload registered in sasl and bsc

```

```
error  
ok
```

Oops, `systools` detected a problem when building the script. Remember how, in your app file, you specified a list of registered processes? Apparently, there is another process defined in the `sasl` app file with the name `overload`. We actually introduced `freq_overload` in [Chapter 7](#), and before changing it, had it registered with the name `overload` in [Chapter 8](#). When creating the first app file, we ended up using the wrong name.

If you are running the script on your laptop, you might get errors informing you that the script was unable to find a certain version of the app file, an error that is easily reproducible if you change any of the versions in `basestation.rel`. This is where version control becomes important. You need to know exactly which module, application, and release versions you are running in production, because your system may be running for years on end and is likely to be managed by other people. Should you get called in to support someone else's mess, at least you'll know what version of the mess you have to deal with.

When creating your boot file, sanity checks are run to:

- Check the consistency and dependencies of all applications defined in the `rel` files. Do all the applications exist, and are there no circular dependencies? Ensure that the versions defined in the `app` files match those specified in the `rel` files.
- Ensure that the `kernel` and `stdlib` applications of type `permanent` are part of the release. Warnings will be raised if `sasl` is not part of the release, but the script and boot file generation will not fail. You can suppress these warnings passing `no_warn_sasl` as one of the options when creating the boot file.
- Detect clashes in the registered process names defined in the application `app` files, ensuring that no two processes are registered with the same name.
- Ensure that all modules defined in the `app` files have a corresponding beam file in the `ebin` directory. While doing so, the sanity check detects any module name clashes, where the same module (or module name) is included in more than one application. If you want to ensure that the beam files match the latest version of the source code, include `src_tests` in the options.

As we look at our release, we see that the registered process name clash arises as an error in our app file. Let's change `overload` to `freq_overload` in the registered process names of the `bsc.app` file and try it again. It worked (at least for us!).

When viewing the resulting contents of the directory as shown in the following example, we discover two new files, `basestation.script` and `basestation.boot`. Before investigating them further, let's use the boot file to start the basestation release.

```
$ erl  
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]
```

```

Eshell V6.1 (abort with ^G)
1> systools:make_script("basestation", [{path, ["bsc/ebin"]}]). 
ok
2> q().
ok
$ ls
basestation.boot      basestation.rel  basestation.script    bsc-1.0
$ erl -pa bsc/ebin/ -boot basestation
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

=PROGRESS REPORT==== 20-Jul-2014::23:02:01 ===
  supervisor: {local,sasl_safe_sup}
    started: [{pid,<0.35.0>},
               {name,alarm_handler},
               {mfargs,{alarm_handler,start_link,[]}},
               {restart_type,permanent},
               {shutdown,2000},
               {child_type,worker}]

...<snip>...

=PROGRESS REPORT==== 20-Jul-2014::23:02:01 ===
  supervisor: {local,bsc}
    started: [{pid,<0.43.0>},
               {name,freq_overload},
               {mfargs,{freq_overload,start_link,[]}},
               {restart_type,permanent},
               {shutdown,2000},
               {child_type,worker}]

=PROGRESS REPORT==== 20-Jul-2014::23:02:01 ===
  supervisor: {local,bsc}
    started: [{pid,<0.44.0>},
               {name,frequency},
               {mfargs,{frequency,start_link,[]}},
               {restart_type,permanent},
               {shutdown,2000},
               {child_type,worker}]

=PROGRESS REPORT==== 20-Jul-2014::23:02:01 ===
  supervisor: {local,bsc}
    started: [{pid,<0.45.0>},
               {name,simple_phone_sup},
               {mfargs,{simple_phone_sup,start_link,[]}},
               {restart_type,permanent},
               {shutdown,2000},
               {child_type,worker}]

=PROGRESS REPORT==== 20-Jul-2014::23:02:01 ===

```

```

application: bsc
started_at: nonode@nohost
Eshell V6.1 (abort with ^G)
1> observer:start().
ok

```

Because the `bsc` application was not placed in the `lib` directory, we have to provide the code search path to the `app` and beam files using the `-pa` directive to the `erl` command. Note all the progress reports that start appearing as soon as `sasl` is started (we've removed a few in our example). Just to be completely sure that the supervision tree has started, start the `observer` tool and look at the Applications tab ([Figure 11-3](#)) and have a look at the `bsc` supervision tree.

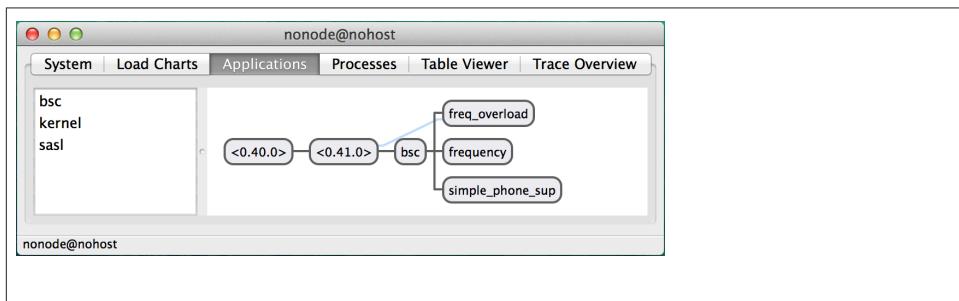


Figure 11-3. The Observer Applications tab

This is how we start OTP compliant simple target systems. This approach is used by several popular open source projects, is more robust than basic target systems, and represents a step in the right direction. But we can (and will) do better! Before discovering how, let's review in more detail the contents of the files we've generated and the parameters we can pass to `systools:make_script/2`.

Script Files

[Figure 11-4](#) shows the basic relationships between files used to build a release. The `basestation.boot` file is a binary file containing all of the commands executed by the Erlang runtime system and needed to start the release. Unlike other files we will look at, the boot file has to be a binary because it contains the commands that load the modules that allow the runtime system to parse and interpret text files. You can find the textual representation of the boot file's commands in `basestation.script`. And even better, for those of you who like to tinker, you can edit or write your own. (Do this while sparing a thought to those using OTP R1 back in 1996, when `make_script/2` still had not been written.)

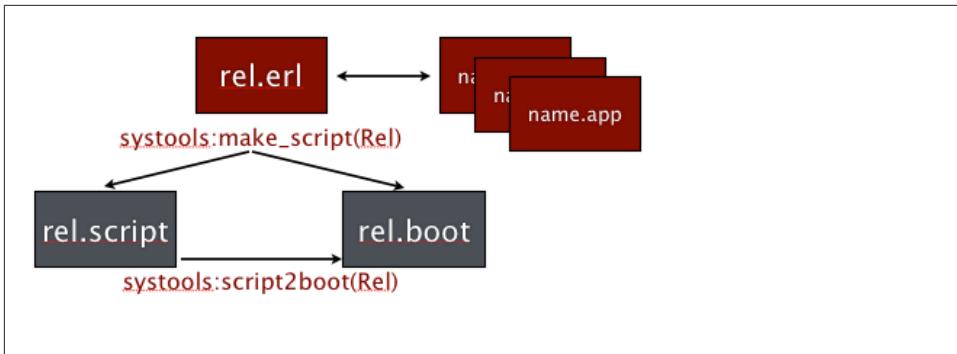


Figure 11-4. Creating Boot and Release Files

Have a look at the contents of a script file. It is a file containing an Erlang term of the format {script, {ReleaseName, ReleaseVsn}, Actions}:

```

{script,
  {"basestation","1.0"},
  [{preLoaded,
    [erl_prim_loader,erlang,erts_internal,init,otp_ring0,prim_eval,
     prim_file,prim_inet,prim_zip,zlib]},
   {progress,preloaded},
   {path,[{"$ROOT/lib/kernel-3.0.1/ebin","$ROOT/lib/stdlib-2.1/ebin"]},
   {primLoad,[error_handler]},
   {kernel_load_completed},
   {progress,kernel_load_completed},
   {path,[{"$ROOT/lib/kernel-3.0.1/ebin"]},
   {primLoad,KernelModuleList}, %%%
   {path,[{"$ROOT/lib/stdlib-2.1/ebin"]},
   {primLoad,StdLibModuleList},
   {path,[{"$ROOT/lib/sasl-2.4/ebin"]},
   {primLoad,SASLModuleList},
   {path,[{"$ROOT/lib/bsc-1.0/ebin"]},
   {primLoad,BscModuleList},
   {progress,modules_loaded},
   {path,
     [{"$ROOT/lib/kernel-3.0.1/ebin","$ROOT/lib/stdlib-2.1/ebin",
      "$ROOT/lib/sasl-2.4/ebin","$ROOT/lib/bsc-1.0/ebin"]},
     {kernelProcess,heart,{heart,start,[]}},
     {kernelProcess,error_logger,{error_logger,start_link,[]}},
     {kernelProcess,application_controller,
       {application_controller,start,KernelAppFile}}}
   {progress,init_kernel_started},
   {apply, {application,load, StdLibAppFile}},
   {apply, {application,load, SASLAppFile}},
   {apply, {application,load, BscAppFile}},
   {progress,applications_loaded},
   {apply,{application,start_boot,[kernel,permanent]}},
   {apply,{application,start_boot,[stdlib,permanent]}}},

```

```
{apply,{application,start_boot,[sasl,permanent]}},  
{apply,{application,start_boot,[bsc,permanent]}},  
{apply,{c,erlangrc,[]}},  
{progress,started]}].
```

We replaced the *kernel*, *stdlib*, *sasl*, and *bsc* applications' module lists and app file contents with variables shown in italics to make the file more book friendly and readable. The script file starts off by defining any modules that have to be preloaded before any processes are spawned. Let's step through these commands one at a time. Although you need not understand what they all mean if all you need to do is get a system up and running, having knowledge of the various steps helps when you have to dive into the internals of the kernel or need to troubleshoot why your system is not starting, or even more worrisome, not restarting.

preLoaded

Contains the list of Erlang modules that have to be loaded before any processes are allowed to start. You can find them in the *erts* application located in the *lib* directory. Of relevance to this section are the *init* module, which contains the code that interprets your boot file, and the *erl_prim_loader* module, which contains information on how to fetch and load the modules.

progress

Lets you report the progress state of your initialization program. The progress state can be retrieved at any time by calling the function *init:get_status/0*. The function returns a tuple of the format *{InternalState, ProgressState}*, where *InternalState* is *starting*, *started* or *stopping*. *ProgressState* is set to the last value executed by the script. In our example, the only progress state that matters to the startup procedure is the last one, *{progress, started}*, which changes *InternalState* from *starting* to *started*. All other phases have no use other than for debugging purposes.

kernel_load_completed

Indicates a successful load of all the modules that are required before starting any processes. This variable is ignored in embedded mode, where loading of the modules happens before starting the system. We discuss the *embedded* and *interactive* modes in more detail later in this chapter.

path

A list of directories, represented as strings. They can be absolute paths or start with the *\$ROOT* environment variable. These directories are added to the code search path (together with directories supplied as command line arguments using *-pa*, *-pz* and *-path*) and used to load modules defined in *primLoad* entries. Note how the generated paths—specifically, the one in the *bsc* application—assume that the beam files of the target environment are located in *\$ROOT/lib/bsc-1.0/ebin* and not *bsc/ebin*. Note also how the application version number in the start scripts has been added to the path, assuming a standard OTP directory structure.

`primLoad`

Provides a list of modules loaded by calling the function `erl_prim_load.erl:get_file/1`. If loading a module fails, the start script terminates and the node is not started. Modules may fail to be loaded when the beam files are missing, are corrupt, or are compiled by a wrong version of the compiler, or when the code search path is incorrect, e.g., if you forgot to add your application to the `lib` directory or have omitted the directory version number. We explain how to troubleshoot startup errors in various places throughout this chapter.

`{kernelProcess, Name, {M, F, A}}`

Starts a kernel process by calling `apply(M, F, A)`. You already know what the error logger and the application controller do. We will look at `heart` in “[Heart on page 281](#)”. The kernel process is monitored, and if anything abnormal happens to it, the node is shut down.

`{apply, {M, F, A}}`

Causes the process initializing the system to execute the `apply(M,F,A)` BIF, where the first argument is the module, the second is the function, and the third is a tuple of arguments for the function. If this function exits abnormally, the startup procedure is aborted and the system terminates. It may not hang and has to return, because starting the node is a synchronous procedure. If an `apply` does not return, the next command will not be executed.

Now that we are enlightened about each line of the script file, we can follow what is happening in our start script.

1. We start off by pre-loading all of the modules in the `erts` application, together with the `error_handler` in the `kernel` application. Once they load, we inform the script interpreter with the `{kernel_load_completed}` and issue a progress report.
2. For all applications listed in the release file, we add the path to the end of the code search path and use `primLoad` to load all of the modules listed in the respective application app files. We then issue a `modules_loaded` progress report.
3. We start all of the kernel processes, starting with `heart`, the `error_logger`, and the `application_controller` (you already know about the latter two). We issue an `init_kernel_started` progress report.
4. All applications that are part of this release are loaded by calling `application:load(AppFile)`. We load the four applications listed in our `rel` file, `kernel`, `stdlib`, `sasl` and `bsc`. When completed, we issue an `applications_loaded` progress report.
5. Now that we've started the kernel processes and loaded all of the applications, it is time to start them. Note how, instead of calling `application:start/1` in the `{apply, {M, F, A}}` tuple, we are calling `application:start_boot/2`. This is an

undocumented function that, unlike `application:start/2`, assumes that the application has already been loaded and asks the application controller to start it.

6. Before issuing the final started progress report, we call `c:erlangrc()`. This function is not documented, but reads and executes the `.erlang` file in your home or Erlang root directory. This is a useful place to set code paths and execute other functions.

Be very careful of the code search paths in your target environment. The only reason our example can start the `bsc` application is that we provide the path to the beam files using `-pa` in the command-line prompt when starting Erlang. Our base station script expects it to be in `$ROOT/lib/bsc-1.0/ebin`. When generating the start script for the target environment, all applications are assumed to be in the directory `AppName-version` within the root directory `$ROOT/lib/`. This will become evident when we generate the target directory structure and files.

The `make_script` Parameters

Let's look at all the options we can pass to the `make_script/2` call in more detail. We already know that `Name` is the name of the release file:

`systools:make_script(Name, OptionsList).`

Options include:

`no_module_tests`

By default, when source code is available, systools ensures that the beam file are up to date and represent the latest version of the source code. This flag will omit these tests.

`{path,DirList}`

Adds paths listed in `DirList` to the code search path. This option can be used along with passing the `-pa` and `-pz` parameters when starting the erlang VM that executes the systools functions. You can include wildcards in your path, so "`lib/*/ebin`" will expand to contain all of the subdirectories in `lib` containing an `ebin` directory.

`local`

Places local paths instead of absolute paths in the start script. This flag is ideal for testing boot scripts using your code and the Erlang runtime system on your local machine.

`{variables,[{Prefix, Var}]}]`

Replaces path-prefixes with variables. It allows you to specify alternative target paths for some or all of your applications. Defining a prefix such as `{"$BSC", "/usr/basestation/"}` results in the path `$BSC/lib/bsc-1.0/ebin`, if the `app` and beam files are found in `/usr/basestation/lib/bsc/ebin`. Similarly, it results in the path `$BSC/ernie/lib/bsc-1.0/ebin`, if the local path is `/usr/basestation/ernie/lib/bsc/ebin`.

`{outdir, Dir}`

Puts the boot and script files in `Dir`.

`exref and {exref, AppList}`

Tests the release with the `Xref` cross reference tool, which looks for calls to undefined and deprecated functions.

`src_tests`

Issues a warning if there is a discrepancy between the source code and the beam file.

`silent`

Returns a tuple of the format `{ok, ReleaseScript, Module, Warnings}` or `{error, Module, Error}` instead of printing results to I/O. Use this option when calling systools functions from scripts or integrating the call in your build process where you need to handle errors.

`no_dot_erlang`

Removes the instructions that load and execute the expressions in the `.erlang` file.

`no_warn_sasl`

Can be used if you are not including `sasl` as one of your default applications and are not interested in the warnings that are generated.

`warnings_as_errors`

Treats warnings as errors, and refuses to generate the script and boot files if warnings occur.

Alternative boot files

If you look in the releases directory of your standard erlang/OTP distribution you are currently running, you will find four boot files and three rel files. They start and load different applications. They include:

`start_clean.boot`

Starts the `kernel` and `stdlib` applications as defined in the `start_clean.rel` file.

`start_sasl.boot`

Starts the `kernel`, `stdlib`, and `sasl` applications as defined in the `start_sasl.rel` file.

`no_dot_erlang.boot`

Starts the `kernel` and `stdlib` applications but does not execute commands in the `.erlang` file. This is useful when determinism is important, because it does not allow the code search paths to be manipulated and other user preferences to be modified.

The fourth file, `start.boot`, is a copy of whichever of the preceding files was selected as the default when installing Erlang. You can rename any of the three files in the list to `start.boot` yourself in the `releases` directory, should you wish to try them out.

You can write your own script files, generate them with `systools:make_script/2`, or change existing ones. If you need to generate a release boot file from a script file, use the `systools:script2boot(File)` function.

Changing script files was a necessity in the good old days when debugging startup issues. In order to pinpoint exactly where the problems occurred, we had to add progress reports after every operation. When working with projects with thousands of modules, if one of the beam files installed on the target machine got corrupted during the build or transfer process, the only way to find it was by adding progress reports after every `primLoad` command in the boot file. It told us in which application directory we had a problem, after which we loaded all of the modules individually, finding the culprit. Today, you can turn on the startup trace functionality by passing the `-init_debug` flag to the `erl` command. It makes the startup phases much more visible. When users are unaware of this option, debugging startup errors can end up being worse than looking for a needle in a haystack. But there are still reasons for manipulating and writing your own release files: to reduce startup times by loading only specific modules and starting specific applications, or to change their start order.

Creating a Release Package

Now that we know the ins and outs of creating and starting a simple target system and have a boot file at hand, let's have a look at how the pros package, deploy, and start their releases. The most solid and flexible way of deploying an Erlang node is an *embedded target system*. Unfortunately Erlang/OTP uses the term “embedded” in several contexts, which we explain in this chapter, so please don't assume it means the same thing each time we use it. Here, by *embedded* we mean our target system becomes part of a larger package running on the underlying operating system and hardware. It is capable of executing as a daemon job in the background, without the need to start an interactive shell or keep it open all the time, and it typically starts when the operating system is booted. To communicate without a shell, an embedded target system streams all I/O through pipes.

Because target environments differ based on design and operational choices, there is no one size that fits all. The basic steps when creating a release package are as follows, but they are often tweaked based on the details of what you are trying to achieve:

1. Create a target directory and release file.
2. Create the `lib` directory with the application versions specified in the `rel` file.

3. Create the release directory with the boot scripts and the application configuration file.
4. Copy the *erts* executable and binaries to the target directory.
5. Create a *bin* directory and copy the configuration files and the start scripts to it.

These steps are, at least in part, usually integrated in the automated build system and the install scripts executed on the target machine or run by one of the many available tools. Because OTP originally did not ship with tools to create target releases, and eventually included a complex tool focused on batch handling, the boundary for what is done by the build environment and what is done by the installation scripts on the target host varies among users. What also varies are the manual versus the automated steps. If you are doing your build on the same hardware and operating system as your target environment, you might be better off getting everything ready in one place. If you do not have this luxury, do not know where (or on what target machine) your deployment will be running, or need other target-specific configuration files created on the fly, parts of the procedure may have to be performed on the target environment.

Now we make our way through the steps required to create a target system manually, assuming that our development and target environments are the same. Based on how you are used to building, deploying, and configuring your target systems, it should be straightforward to understand where you should be drawing the boundary between what you do in your build process and what you do on the target host. We also cover some tools that can be used to automate this process.

We start off by creating the target directory, which we are going to call *ernie*,¹ and adding the *releases* and *lib* directories to it. Along with standard Unix commands, we use the `systools:make_tar/2` library function. We start in the same directory as the *bsc* application directory. The `make_tar` call also expects the system *release* and boot files to be located here, alongside a *config* file.

The configuration file is optional at this stage. You might want to generate target-specific values at install time, overriding those specified in the *app* files. If you choose to omit it at this stage, you must not forget to add it when installing the system, else your system will not start. The configuration file must be named *sys.config*, although you can change the name by tweaking the arguments you pass to the emulator when starting it.

We create our *ernie* target directory, rename the configuration file *sys.config*, and place it in the same directory as the *bsc* application and the *rel* and boot files. When done, we can create our tar file:

1. Ernie is the user name of the account where the AXD301 ATM switch runs its Erlang nodes. A tip down memory lane for those who contributed to Erlang in some shape or form in the early days, including many of the reviewers of this book.

```

$ mkdir ernie
$ cp bsc.config sys.config
$ erl
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V6.1 (abort with ^G)
1> systools:make_tar("basestation", [{erts, "/usr/local/lib/erlang/"},
                                         {path, ["bsc/ebin"]}], {outdir, "ernie"}).
ok
2> q().
ok
3>
$ cd ernie
$ ls
basestation.tar.gz
$ tar xf basestation.tar.gz
$ ls
basestation.tar.gz      lib
erts-6.1                releases
$ ls lib/
bsc-1.0      kernel-3.0.1    sasl-2.4      stdlib-2.1
$ ls releases/
1.0          basestation.rel
$ ls releases/1.0/
basestation.rel start.boot    sys.config
$ ls erts-6.1/bin/
beam          dialyzer       erl.src        heart        start.src
beam.smp      dyn_erl        erlc          inet_gethost  start_ertl.src
child_setup   epmd           erlexec       run_erl     to_ertl
ct_run        erl            escript       start        typer
$ rm basestation.tar.gz

```

Our call to `systools:make_tar(Name, OptionsList)` generates the `basestation.tar.gz` package. `Name` is the name of the release and `OptionsList` accepts all of the options `make_script` takes, together with the `{erts, Dir}` directive. We give this directive if we wish to include the runtime system binaries, resulting in the `erts-6.1` directory. That is not always the case, because the runtime system binaries might already be installed on the target machine, or a single version of Erlang might be used to run multiple nodes. Also note that the `sys.config` file is included in the `releases/1.0` directory. If it is in a different directory from the `rel` file, you have to copy it in a later stage of the installation.

You could deploy `basestation.tar.gz` to your target machine and run your local configuration scripts when you install the node, or do it in your build environment and create a single tar file for all deployments of this particular node. Keep in mind that your node might run in tens of thousands of independent installations, one for every base station controller your company sells, or if hosted, in multiple occurrences of the node, all in a single installation. Your configuration parameters will depend on your needs and type of installation; they might be the same configuration parameters across all tens

of thousands of deployments, or may have to be individually customized when installing the software on each target environment.

In our example, we untar the `basestation.tar.gz` packet. The remaining steps could run either on the target or in our build environment. When untarring the file, we find three new directories: the `lib` directory, containing all of the application directories (including their version numbers), the `releases` directory, and the `erts` directory. The `erts` directory is there because we included the `{erts, Dir}` directive in the `sys_tools:make_tar/2` call.

We already know that `Name` is the name of the release file:

```
systools:make_tar(Name, OptionsList).
```

`OptionsList` is a list that can be empty or can contain some combination of the following elements:

`{dirs, IncDirList}`

Copies the specified directories (in addition to the defaults `priv` and `ebin`) to the application subdirectories. Thus, to add `tests`, `src`, and `examples` to the release, set the `IncDirList` to `[tests, src, examples]`.

`{path, DirList}`

Adds paths to the code search path. This option can be used along with the `-pa` and `-pz` parameters passed when starting the Erlang VM that runs the system. You can include wildcards in your path. For instance, `["lib/*/ebin"]` will expand to contain all of the subdirectories in `lib` that contain an `ebin` directory.

`{erts, Dir}`

Includes the binaries of the Erlang runtime system found in directory `Dir` in the target tar file. The version of the runtime system is extracted from the `rel` file. Make sure that the binaries have been compiled and tested on your target operating system and hardware platform.

`{outdir, Dir}`

Puts the tar file in directory `Dir`. If omitted, the default directory is the same directory as that of the `.rel` file.

`exref and {exref, AppList}`

Tests the release with the `exref` cross reference tool, which looks for calls to undefined and deprecated functions. This is the same test executed by the `sys_tools:make_script/2` call when passing the same option.

`src_tests`

Issues a warning if there are discrepancies between the source code and the beam files. This is the same test executed by the `systools:make_script/2` call when passing the same option.

`silent`

Returns a tuple of the format `{ok, ReleaseScript, Module, Warnings}` or `{error, Module, Error}` instead of printing the results to I/O. You can get formatted errors and warnings by calling `Module:format_error(Error)` and `Module:format(Warning)` respectively. Use this option if you are integrating `systools` in your build process; it works in the same way as for the `systools:make_script/2` call.

Two additional options, `{variables, [{Prefix, Var}]}]` and `{var_tar, VarTar}`, allow you to change and manipulate the way target libraries and packages are created. Use them when deviating from the standard Erlang way of doing things; for example, if you prefer to deploy your release as deb, pkg, rpm (and other) packages or containers. They allow you to override the application installation directory (by default set to `lib`) and influence where and how the packages are stored. We do not cover these options in this chapter, but for more information and some examples, read the `systools` reference manual page.

Start Scripts and Configuring on Target

Now that we have our target files in place, we need to configure our start scripts. Here we go through these steps manually, introducing tools that automate the process later.

1. In the target directory (`ernie`, in our case), create a `bin` directory in which we place and edit the start scripts that will boot our system.
2. Create the `log` directory, to which all debug output from the start scripts is sent. It will be one of the first points of call when the system fails to start.
3. Create a file called `start_erl.data` in the `releases` directory containing the versions of the Erlang runtime system and its release.
4. If our original tar file did not contain a `sys.config` file, we create one (possibly empty) and place it in the release version directory.

At this point, fingers crossed, everything will start. Let's go through these steps in more detail, adding and editing files as we go along:

```
$ mkdir bin  
$ cp erts-6.1/bin/start.src bin/start  
$ cp erts-6.1/bin/start_erl.src bin/start_erl  
$ cp erts-6.1/bin/run_erl bin/.  
$ cp erts-6.1/bin/to_erl bin/.  
$ mkdir log
```

In our example, we create the `bin` directory and copy `start.src` and `start_erl.src` to it, renaming them `start` and `start_erl` respectively. We also copy over `run_erl`, which the start scripts expect to be available locally, and `to_erl`, which we will use to connect to an

embedded Erlang shell. The start script initializes the environment for the embedded system, after which it calls *start_erl*, which in turn starts Erlang via the *run_erl* script.

Think of *start_erl* as an embedded version of *erl* and *start* as a script you can use and customize as you please. Depending on your needs and requirements, you might also want your own version of the *erl* and *heart* scripts and, if running distributed Erlang, the *epmd* binary. All of these can be copied from the *bin* directory of the runtime system.

Now that the files and binaries are in place, we need to edit them accordingly. We modify the *start* file, replacing %FINAL_ROOTDIR% with the absolute path to the new Erlang root directory. In our case, this directory is *ernie*, so to find its absolute path, we type the *pwd* command in the shell. We change the file using the editor *vim*, showing you the before and after versions using the *diff* command.

```
$ pwd
/Users/francescoc/OTPBOOK/current/code/ch11/ernie
$ vim bin/start
$ diff erts-6.1/bin/start.src bin/start
27c27
< ROOTDIR=%FINAL_ROOTDIR%
---
> ROOTDIR=/Users/francescoc/OTPBOOK/current/code/ch11/ernie
$ echo '6.1 1.0' > releases/start_erl.data
$ bin/start
$ bin/to_erl /tmp/
Attaching to /tmp/erlang.pipe.1 (^D to exit)

1> application:which_applications().
[{'bsc', "Base Station Controller", "1.0"}, 
 {sasl, "SASL CXC 138 11", "2.4"}, 
 {stdlib, "ERTS CXC 138 10", "2.1"}, 
 {kernel, "ERTS CXC 138 10", "3.0.1"}]
2> [Quit]
$ ls /tmp/erlang.*
/tmp/erlang.pipe.1.r /tmp/erlang.pipe.1.w
```

Having edited the *start* file, we create the *start_erl.data* file in the *releases* directory. It contains the version of the Erlang runtime system and the release directory containing all the boot scripts and configuration files for the release. These two items, in our example both numbers, are separated by a space.

We are now able to boot our system with the *start* command. Notice how, unlike the *erl* command, this release starts as a background job. To connect to the Erlang shell, we use the *to_erl* command, passing it the */tmp* directory where the read and write pipes reside.



When running an embedded Erlang system, you might out of habit exit the shell using CTRL-C a. The CTRL-C invokes the virtual machine break handler, after which you can execute one of the following commands:

```
(a)abort (c)continue (p)roc info (i)nfo  
(l)oaded (v)ersion (k)ill (D)b-tables  
(d)istribution.
```

Thus, the **a** terminates the Erlang node.

To avoid termination, be careful to exit the shell using CTRL-D. If you type `q()`, `halt()`, or CTRL-C a out of habit, you will kill the whole background job. By using CTRL-D, you exit the `to_erl` shell while keeping the Erlang VM alive running in the background.

If you are trying to connect to the pipes on your computer and get an error of the form *No running Erlang on pipe /tmp/erlang.pipe: No such file or directory*, look in the log directory to find out why your Erlang node failed to start. All start errors in your scripts will be recorded there. Problems might include wrong paths, missing `sys.config` files, a corrupt boot file, or an incorrectly named binary.

It is good practice to always include the `erl` command in the `bin` directory of your target system. This will come as a blessing when, after a failure of some sort, you are unable to restart your node. Your first point of call in these situations will be the SASL report logs, where crash and error reports will in most cases tell you what triggered the chain of errors that caused the node to fail. The last thing you want to do is to have to move the SASL logs to a remote computer every time you want to view them just because your Erlang nodes will not start. Be safe and always generate a second boot file similar to `start_sasl.boot` that contains the same application versions of the `kernel`, `stdlib`, and `sasl` as your system.

In our example, we used the `/tmp` directory for the read and write pipes. If you plan on running multiple embedded nodes on the same machine, though, this will cause a problem. A good practice is to redirect your pipes to a subdirectory of your Erlang root directory in your target structure. This allows multiple node instances to run on the same computer, a common practice in many systems. If you look at the last line of the start script, you will see where to replace `/tmp` with the absolute path of your new pipes in the root directory. You can also redirect all of the logs elsewhere:

```
$ROOTDIR/bin/run_erl -daemon /tmp/ $ROOTDIR/log "exec $ROOTDIR/bin/start_erl ..."
```

Arguments and Flags

So far, so good. But what if we want to start a distributed Erlang node or add a `patch es` directory to the code search path? Or maybe you have developed a dislike toward the `sys.config` filename and want to retain the original `bsc.config` file? Or even more impor-

tantly, are there flags we can pass to the emulator that will disable the ability to kill the node via CTRL-C a?

When starting Erlang, we can pass three different types of arguments to the runtime system. They are *emulator flags*, *flags*, and *plain arguments*. You can recognize emulator flags by their initial + character. They control the behaviour of the virtual machine, allowing you to configure system limits, memory management, scheduler options, and other items specific to the emulator.

Flags start with - and are passed to the Erlang part of the runtime system. They include code search paths, configuration files, environment variables, distributed Erlang settings, and more.

Plain arguments are user-defined and not interpreted by the runtime system. You first came across them in “[Environment Variables](#)” on page 211 to override application environment variables in *app* and configuration files from the command line. You can use plain arguments in your application business logic.

The following sample command uses several different arguments:

```
erl -pa patches -boot basestation -config bsc -init_debug +Bc
```

This starts Erlang with the *patches* directory added to the beginning of the code search path. It also uses the *basestation.boot* and *bsc.config* files to start the system, and sets the *init_debug* flag, increasing the number of debug messages at startup. The *+Bc* emulator flag disables the shell break handler, so when you press the sequence CTRL-C a, instead of terminating the virtual machine you terminate just the shell process and restart it.

Let’s look at some of the emulator flags in more detail. We’ve picked high-level flags and system-limit flags that do not deal with memory management, multi-core, ports and sockets, low-level tracing, or other internal optimizations. The internal optimizations are outside the scope of this book and should be used with care, only if you are sure of what you are doing. You can read more about the arguments we cover (and those we don’t) in the manual pages for *erl*. The ones we list are those we have used ourselves in some shape or form before the need to optimize our target systems.

+Bc

It is dangerous to keep the break handler enabled in live systems, as your fingers are often faster than your mind (especially if this is a support call in the middle of the night when your mind is still fast asleep). If you are used to terminating the shell that way, you will be inclined to do it on production systems as well. Using the *+Bc* flag makes CTRL-C a terminate the current shell and start a new one without affecting your system. This is the option to enable for all your live systems.

+Bd

This allows you to terminate the Erlang node using simply CTRL-C, bypassing the break handler altogether.

+Bi

This makes the emulator ignore CTRL-C, in which case the only way to terminate your Erlang virtual machine is using the shell command `q()` or the `halt()` BIFs. This option is dangerous because it does not allow you to recover should an interactive call fail to return, thereby hanging the shell.

+e Num

This sets the maximum number of ETS tables, which defaults to 2053.

+P Num

This changes the system limit on the maximum number of processes allowed to exist simultaneously. The limit by default is 262,144, but can range from 1024 to 134,217,727.

+Q Num

This changes the maximum number of ports allowed in the system, set by default to 65,536. The allowable range is 1024 to 134,217,727.

+t Num

This allows you to change the maximum number of allowed atoms, set by default to 1,048,576. These limits are specific to Erlang 17 and to Unix-based OSes. Default values might differ on other operating systems.

+R Rel

This allows your Erlang node to connect using distributed Erlang to other nodes running an older, potentially non-backward-compatible version of the distribution protocol.

Regular flags are defined at startup, retrieved in the Erlang side of the runtime system, and used by standard and user-defined OTP applications alike. Remember that large parts of the Erlang kernel and runtime system are written in Erlang, so how you define and retrieve flags in your application is identical to how Erlang defines and retrieves them to its runtime. Here are the main flags:

-Application Key Value

Sets Application's environment variable **Key** to **Value**. We covered this option in “[Environment Variables](#)” on page 211.

-args_file FileName

Allows you to list all of the flags, emulator flags, and plain arguments in a separate configuration file named **FileName**, which is read at startup. This is the recommended approach so as to avoid the need to mess with the start scripts.

-async_shell_start

Avoids processing what you type in the shell until the system has been completely booted. This is useful when you want to be sure no one is allowed to fiddle with the

applications when starting the system. It is not useful, however, when you are trying to debug startup issues or figure out where timeouts are occurring.

-boot *filename*

Sets the name of the boot file to *filename.boot*. If you do not include an absolute path, the emulator assumes the boot file is in the *\$ROOT/bin/* directory.

-config *filename*

Sets the location and name of the configuration file to *filename.config*.

-connect_all false

Stops the global subsystem from maintaining a fully connected network of distributed Erlang nodes, in effect disabling the subsystem.

-detached

Starts the Erlang runtime system in a manner detached from the system console. You need this option when running daemons and background processes. The detached option implies **-noinput**, which basically starts the Erlang node but not the shell process that runs the read-evaluate loop interpreting all the commands you type. The **-noinput** option in turn implies the **-noshell** command, which starts the Erlang runtime system without a shell, potentially making it a component in a series of Unix pipes.

-emu_args

Prints, at startup, all of the arguments passed to the emulator. Keep this on all the time, as you will never know when you need access to the information.

-init_debug

Provides you with detailed debug information at startup, outlining every step executed in the boot script. The overheads of using **-init_debug** and **-emu_args** are negligible, but the information they provide is priceless when troubleshooting.

-env *Variable Value*

An alternative (and convenient) way to set host operating system environment variables. It is mainly used for testing, but is also useful when dealing with Erlang-specific values.

-eval

Parses and executes an Erlang expression as part of the node's initialization procedure. If the parsing or execution fails, the node shuts down.

-hidden

When using distributed Erlang, starts the Erlang runtime system as a hidden node, publishing neither its existence nor the existence of the nodes to which it is connected.

-heart

Starts the external monitoring of the Erlang runtime system. If the monitored virtual machine terminates, a script that can restart it is invoked. We cover heart in detail in “Heart” on page 281.

-mode Mode

Establishes how code is loaded in the system. If Mode is `interactive`, calls to modules that have not been loaded are automatically searched for in the code search path. Your target systems should run in `embedded` mode, where all modules should be loaded at startup by the boot file, and calls to nonexisting modules should result in a crash. You can still load modules in embedded mode using the `l(Module)` or `code:load_file(Module)` calls from the shell.

Running in `embedded` mode is recommended for all production systems. It ensures that in the middle of a critical call, you do not pause the process while traversing the code search path looking for a module that has not been loaded.

-nostick

Disables a feature that prevents loading and overriding modules located in sticky directories. By default, the `ebin` directories of the `kernel`, `compiler`, and `stdlib` applications are sticky, a measure intended to prevent key elements of the system from being accidentally corrupted.

-pa and -pz

Adds directories containing beam files to the beginning and end of the code search path, respectively. One common use is to add `-pa patches` to point to a directory used to store temporary patches in between releases.

-remsh node

Starts a shell connected to a remote `node` using distributed Erlang. This is useful when running nodes with no shells or when you need to remotely connect to a node.

-shutdown_time Time

Specifies the number of milliseconds the system is allowed to spend shutting down the supervision trees. It is by default set to infinity. Use this option with care, though, because it overrides the shutdown values specified in the behaviour child specifications.

-name name and -sname name

When working with distributed Erlang, these start distributed nodes with long or short names respectively. If nodes are to communicate with each other, they must share a cookie, which can be set using the `-setcookie` directive, and all have either long or short names. Nodes with short and long names can not communicate with each other.

`-s module`, `-s module function`, and `-s module function args`

The first executes, at startup, `module:start()`. The second executes `module:function()`. The third is like the second but includes the argument list to the function. All Args are passed as atoms. The `-run` option works similarly, except that if arguments are defined, they are passed as a list of strings to `module:function/1`. Functions executed by `-run` and `-s` must return, or otherwise the startup procedure will hang. If they terminate abnormally, they will cause the node to terminate as well, aborting the startup procedure.

Let's try using `-s`, `-eval` and `-run` in the shell to get a feel for them:

```
$ erl -s observer
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V6.1 (abort with ^G)
1> q().
ok
$ erl -eval 'Average = (1+2+3)/3, io:format("~p~n",[Average]), erlang:halt()' -noshell
2.0
$ erl -run io format 1 2 3
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

123Eshell V6.1 (abort with ^G)
1> q().
ok

$ erl -s io format 1 2 3
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

>{"init terminating in do_boot",{badarg,[{io,format,[<0.24.0>,['1','2','3'],[],[]]}, {init,start_it,
Crash dump was written to: erl_crash.dump
init terminating in do_boot ()}
```

We successfully use `erl -s observer` to call `observer:start()`. You do not see it in the shell, but it opens up the observer WX widgets window. It is an efficient way to start debugging tools when starting the emulator. We then use the `-eval` flag to calculate the average of three integers, print out the result and stop the emulator, all without starting the erlang shell. In our third and fourth examples, we use `-run io format 1 2 3` to call `io:format(["1","2","3"])` and `-s io format 1 2 3` to call `io:format(['1','2','3'])`. The latter crashes because it attempts to call `io:format/1` with a list of atoms, when it is expecting a string.

When using the `-run` and `-s` flags, beware of calling functions such as `spawn_link` and `start_link` that link themselves to the initialization process, because the process is there to initialize the system and not act as a parent. Although the process currently continues running after executing the initialization calls, you should not depend on that behaviour because it is not documented and might change in a future release.



Do not ship basic target systems unless they are proof of concepts or quick hacks. If your program is started by a script that invokes `erl -s myprojectsup -noshell`, you lose all of the benefits gained by OTP applications and their startup, supervision, and upgrade procedures. You have everything to gain from using boot files and shipping your systems as embedded target systems.

Applications can use the functions `init:get_arguments()` and `init:get_argument(Flag)` to retrieve flags. `Flag` can be one of the predefined flags `root`, `progname`, and `home`, together with all other command line user-defined flags.

Plain arguments include all arguments specified before emulator flags and regular flags, after the `-extra` flag, and in between the `--` directive and the next flag. We can retrieve plain arguments using the `init:get_plain_arguments/0` call:

```
$ erl one -two three -pa bin/bsc -- four five -extra 6 7 eight
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V6.1 (abort with ^G)
1> init:get_plain_arguments().
[{"one","four","five","6","7","eight"]
2> init:get_argument(two).
{ok,[["three"]]}
3> init:get_argument(pa).
{ok,[["bin/bsc"]]}
4> init:get_argument(progname).
{ok,[["erl"]]}
5> init:get_argument(root).
{ok,[["/usr/local/lib/erlang"]]}
6> init:get_argument(home).
{ok,[["/Users/francescoc"]]}
```

Heart

It is customary to run your embedded Erlang systems as daemon jobs, automatically starting them when the computer they are supposed to run on is booted. So if there is a power outage or any other failure or maintenance procedure that requires a reboot, your system will start automatically. But what happens if only the Erlang node itself crashes or stops responding? It could be an unexpected memory spike, a top-level supervisor terminating, a dodgy NIF causing a segmentation fault in the virtual machine, or lo and behold, a bug in the virtual machine that causes the system to hang. This is where you need to enable `heart`.

`Heart` is an external program that monitors the virtual machine, receiving regular heartbeats sent by an Erlang process through a port. If the external program fails to receive a heartbeat within a predefined interval, it attempts to terminate the virtual machine and invokes a user-defined command to restart the runtime system.

Let's write a very simple script, *bsc_heart*, that simply calls the *bin/start* command. We could just set *start* as the heart command, but real world scenarios tend to be too complex for a blind restart and so a restart script is typically used. We could, after failed restart attempts, come to the conclusion that this is a cyclic restart from which we cannot recover, and opt to not restart the node. Or after a certain number of restart attempts, allowed only at variable (but increasing) time intervals, we could reboot the operating system. Or trigger other auto-diagnostic scripts that would run sanity tests on the surrounding environment. The options are many, typically depending on your deployment environment and monitoring/alerting facilities, so restart scripts can be as simple or as complex as you want them to be. Let's use the following script, which we place in the *bin* directory of our target installation:

```
#!/bin/sh
#Basic Heart Script for the Base Station Controller

ROOTDIR=/Users/francescoc/OTPBOOK/current/code/ch11/ernie

$ROOTDIR/bin/start
```

We set the *HEART_COMMAND* environment variable, edit the *start_erl* script to include *heart*, and start the base station controller, killing it in a variety of different ways. Despite that, every time we connect to the I/O pipes, the system is up and running:

```
$ export HEART_COMMAND=/Users/francescoc/OTPBOOK/current/code/ch11/ernie/bin/bsc_heart
$ vim bin/start_erl
$ diff bin/start_erl erts-6.1/bin/start_erl.src
46c46
< exec $BINDIR/erlexec -boot $RELDIR/$VSN/start -config $RELDIR/$VSN/sys ${1+"$@"} -heart
$ bin/start
$ bin/to_erl /tmp/
Attaching to /tmp/erlang.pipe.5 (^D to exit)

1> halt().
heart: Sat Aug 23 12:49:47 2014: Erlang has closed.
[End]
$ bin/to_erl /tmp/
Attaching to /tmp/erlang.pipe.5 (^D to exit)

1>
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
      (v)ersion (k)ill (D)b-tables (d)istribution
a
[End]
$ bin/to_erl /tmp/
Attaching to /tmp/erlang.pipe.5 (^D to exit)

1>
```

We see that using *halt()* or CTRL-C a kills the node, because every time we connect, the command prompt is 1 again. The heart is immediately restarting the process.

The following OS environment variables, all optional, can be set either in the start scripts, when booting your system using the `-env` flag, or wherever else you might choose to configure such variables.

`HEART_COMMAND`

The name of the script triggered when the timeout occurs. If this variable is not set, a timeout will trigger a warning and the system will not be restarted.

`HEART_BEAT_TIMEOUT`

The number of seconds heart waits for a heartbeat before terminating the virtual machine and invoking the heart command. In Erlang 17, it can be a value greater than 10 and less than or equal to 65,535. Omitting this setting defaults the timeout to 60 seconds.

`ERL_CRASH_DUMP_SECONDS`

Specifies how long the virtual machine is allowed to spend writing the crash dump file before being killed and restarted. Because crash dump files can be substantial, the virtual machine can take its time writing it to disk. The default setting when using heart and not setting this variable is 0, meaning that no crash dump file is written; the virtual machine is immediately killed and the heart command is immediately invoked. Setting the value to -1 (or any other negative number) allows the virtual machine to complete writing the crash dump file no matter how long it takes. Any other positive integer denotes the number of seconds allowed to the virtual machine to write the crash dump file before it terminates and is restarted.

In our example, we decided to set the environment variables in the Unix shell, but we could just as easily have edited the `start_erl` file or passed it as a flag to `erl` using the `-env variable value` argument:

```
erl -heart -env HEART_BEAT_TIMEOUT 10  
-env HEART_COMMAND boot_bsc
```



Race conditions between heart, heartbeats, and restarts can occur. If you do not anticipate and check for these race conditions, they will leave you scratching your head when you are trying to figure out what went wrong. There have been cases where an Erlang virtual machine was chugging away under extreme heavy load, but the heartbeat never reached heart because of underlying OS issues, perhaps as a result of I/O starvation together with a low HEART_BEAT_TIMEOUT value. The lack of the heartbeat caused heart to terminate the Erlang VM and restart it. No crash dump was generated because heart, at least on Unix-like systems, terminates its target with extreme prejudice via SIGKILL, which the target cannot catch. Killing the Erlang VM (and possibly rebooting the OS itself) might have been the solution to the problem, but it was not of any help to the poor developers who were looking for an Erlang-related VM crash, trying to figure out why there was no crash dump file.

Heart works on most operating systems. Discussing how it executes on Windows and other non-Unix based OSes is beyond the scope of this book, as is the ability to connect and configure it to work with the Solaris hardware watchdog timer. For more information, read the manual page for `heart` that comes with the standard Erlang distribution.

How Does Yaws Use Heart?

As an example of heart usage, let's consider the Yaws web server, originally developed by Claes "Klacke" Wikström and available from both [the Yaws web site](#) and [GitHub](#). Yaws includes the ability to use heart in an interesting way: to get around heart's stubborn habit of attempting endlessly to restart its target, the Yaws restart script keeps track of how many times it has been restarted within a specified time period, much like supervisor child restart counts in OTP. To accomplish this, Yaws sets HEART_COMMAND as shown here:

```
HEART_COMMAND="$ENV_PGM \
    HEART=true \
    YAWS_HEART_RESTARTS=$restarts \
    YAWS_HEART_START=$starttime \
    $program"
```

As you can see, the Yaws HEART_COMMAND includes the setting of several other environment variables that its restart shell script examines when it executes due to a heart restart:

HEART environment variable

Set to true so that Yaws knows heart is controlling it.

YAWS_HEART_RESTARTS environment variable

Tracks how many times Yaws has been restarted.

YAWS_HEART_START environment variable

Tracks the start time based on the Unix epoch (the number of seconds since January 1, 1970).

\$restarts and \$starttime shell variables

Help Yaws calculate new settings for HEART_COMMAND based on the values of the YAWS_HEART_RESTARTS and YAWS_HEART_START set for the previous restart.

When you run Yaws, you specify via command-line arguments the maximum number of restarts allowed in a given period. If the Yaws shell script detects through these environment variables that it has restarted too many times in the specified period, it emits an error message and refuses to restart. For more details, see [the source code for the Yaws start script](#).

The Erlang Loader

You might sometimes run a release on embedded devices with little or no disk space and want to change the method the runtime system uses to load modules. Instead of reading them from a file, you might want to load them from a database or from another node across the network. The `-loader` argument specifies how the `erl_prim_loader` fetches the modules. The default loader, `efile`, retrieves the modules from the local file system. If you want to use the boot server on another machine, you must specify the `inet` loader. When using `inet`, you must include the name of the remote node where the boot server is running through the `-id name` argument, where `name` comes from the `-name` or `-sname` flags issued when starting the remote node. You must also include the IP address of that machine, using the `-hosts address` flag, where `address` is a string IP address, such as one consisting of four integers separated by a period. An example is `-id foo -hosts "127.0.0.1"`, which specifies that the boot server is running in the `foo` Erlang virtual machine on the local host.

To see loading in action, we first generate a `basestation.boot` file using the `local` option to `systools:make_script/2`. The `local` option is critical, as it ensures that our local copy of `bsc` beam files can be found without us having to install them into the `lib` directory of the official release. It basically adds the local path to the `bsc` application in the boot server's load path so that generating the `basestation.boot` file succeeds:

```
$ erl -pa bsc/ebin
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V6.1 (abort with ^G)
1> systools:make_script("basestation", [local]).
ok
```

Next, we start the boot server:

```
$ erl -name foo@127.0.0.1 -setcookie cookie
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]
```

```
Eshell V6.1 (abort with ^G)
(foo@127.0.0.1)1> erl_boot_server:start([{127,0,0,1}]).
```

```
{ok,<0.42.0>}
```

With the boot server started and ready to serve requests, we can start our node:

```
$ erl -name bar@127.0.0.1 -id foo -hosts 127.0.0.1 -loader inet -setcookie cookie
  -init_debug -emu_args -boot basestation
Executing: /usr/local/lib/erlang/erts-6.1/bin/beam.smp  /usr/local/lib/erlang/erts-6.1/bin/beam.smp
  -root  /usr/local/lib/erlang -programe erl -- -home /Users/francescoc -- -name bar@127.0.0.1
  127.0.0.1 -loader inet -setcookie cookie -init_debug -boot basestation
{progress,preloaded}
{progress,kernel_load_completed}
{progress,modules_loaded}
{start,heart}
{start,error_logger}
{start,application_controller}
{progress,init_kernel_started}

...<snip>....
```

=PROGRESS REPORT==== 18-Aug-2014::13:20:54 ===

```
supervisor: {local,bsc}
  started: [{pid,<0.50.0>},
             {name,simple_phone_sup},
             {mfargs,{simple_phone_sup,start_link,[]}},
             {restart_type,permanent},
             {shutdown,2000},
             {child_type,worker}]
{apply,{c,erlangrc,[]}}
```

=PROGRESS REPORT==== 18-Aug-2014::13:20:54 ===

```
application: bsc
  started_at: 'bar@127.0.0.1'
{progress,started}
```

Eshell V6.1 (abort with ^G)

```
(bar@127.0.0.1)1> application:which_applications().
[{bsc,"Base Station Controller","1.0"},{sasl,"SASL CXC 138 11","2.4"},{stdlib,"ERTS CXC 138 10","2.1"},{kernel,"ERTS CXC 138 10","3.0.1"}]
```

As the output shows, our node was able to boot by loading from the remote boot server. Although our example uses `localhost` (127.0.0.1), thus making one wonder whether loading is occurring over the network or from the local file system, you can try this on your own network on two different hosts and see for yourself that the necessary files are loaded from the remote boot server.

The init module

The init module is preloaded in the Erlang runtime system. It manages arguments and the startup and shutdown procedures of your release. At startup, it executes all the commands in the boot file. Of interest to us is the ability the module gives us to restart the system, cleanly shut down all applications, and stop the node, as well as the ability to reboot the virtual machine. Here is a list of common uses for the module.

`init:restart/0`

Restarts the system in the Erlang node without restarting the emulator. Applications are taken down smoothly, modules are unloaded, and the ports are closed, after which the boot file is executed again, using the same boot arguments originally provided. You can use the `-shutdown_time` flag to limit the amount of time spent taking down the applications.

`init:reboot/0`

Like `restart`, with the difference that the emulator is also shut down and restarted. Heart, if used, will attempt to restart the system, causing a potential race condition that will resolve itself when it kills the emulator and restarts it. Timeout values set with the `-shutdown_time` flag will be followed.

`init:stop/0`

Takes the system down smoothly and stops the emulator. If running, heart is also stopped before any attempts to restart the node are made. This is the correct way to stop running nodes, because it allows the applications to terminate and clean up after themselves and properly shut down. Calling `init:stop(Status)` has the same effect as calling the shell command `halt(Status)`. Timeout values set with the `-shutdown_time` flag will be followed.

`init:get_status()`

Determines whether the system is being started, is stopped, or is currently running. It returns a tuple of the format `{InternalStatus, ProvidedStatus}`, where `InternalStatus` is one of `starting`, `started`, or `stopping`. When starting the system, `ProvidedStatus` indicates what part of the boot script `init` is currently running. It gets `Info` status from the last `{progress, Info}` term interpreted by the boot.

We have already covered other useful functions in the `init` module, including `get_arguments/0`, `get_argument/1`, and `get_plain_arguments/0`, in “[Arguments and Flags](#)” on page 275.

Other Tools

Many of the manual tasks we have gone through in this chapter are automated by various tools. Automation is required to generate templates, build the release, and generate the target structure. Because there have been no standards or comprehensive ways of ship-

ping releases developed to date—just preferred or recommended approaches—tools that are now shipped with Erlang/OTP and tools developed by the community are both complementary and overlapping in functionality. In the remainder of this chapter, we cover three tools: *rebar*, *relx*, and *reltool*, helping you evaluate which of them best suits your needs.

Rebar is a general build tool that also manages releases and dependencies. *Relx* is a release and target system builder, assembling systems based on configurations you pass on to it. *Reltool* consists of a graphical front-end that allows you to examine application dependencies and customize your build. The front-end creates a configuration file used by the back-end to build your customized target systems. The batch processing capabilities of the back-end and the completeness of the configuration file have made *reltool* the tool users integrate into more complex build systems.

Rebar

The *rebar* tool is one of the most widely used Erlang build tools. It's a comprehensive tool that addresses a number of project management needs, including dependency management, compilation, and release generation. You can also enhance or extend its functionality via plug-ins.

To obtain *rebar*, you can either download a prebuilt version from its source repository website:

```
$ curl -L0 https://github.com/rebar/rebar/wiki/rebar
```

or clone the *rebar* git repository and build it from source:

```
$ git clone https://github.com/rebar/rebar
$ cd rebar
$ ./bootstrap
Recompile: src/rebar
Recompile: src/rebar_abnf_compiler
...
Recompile: src/rebar_xref
==> rebar (compile)
==> rebar (escriptize)
Congratulations! You now have a self-contained script called "rebar" in
your current working directory. Place this script anywhere in your path
and you can use rebar to build OTP-compliant apps.
```

Some Erlang projects that have been around for a few years include their own *rebar* executables in their source repositories. This was originally done to make it easier for users to build projects without forcing them to first build *rebar*, but given how widespread *rebar* has become, including your own copy of it in your project is no longer necessary. A user need only place a copy somewhere in their shell path, such as */usr/local/bin*, and use it from there.

Running *rebar* with no arguments provides information about how to use it. Here is part of its output:

```
$ rebar
No command to run specified!
Usage: rebar [-h] [-c] [-v <verbose>] [-q <quiet>] [-V] [-f]
           [-D <defines>] [-j <jobs>] [-C <config>] [-p] [-k]
           [-r <recursive>] [var=value,...] <command,...>

-h, --help      Show the program options
-c, --commands  Show available commands
-v, --verbose   Verbosity level (-v, -vv)
-q, --quiet     Quiet, only print error messages
-V, --version   Show version information
-f, --force     Force
-D              Define compiler macro
-j, --jobs      Number of concurrent workers a command may use.
                Default: 3
-C, --config    Rebar config file to use
-p, --profile   Profile this run of rebar
-k, --keep-going Keep running after a command fails
-r, --recursive Apply commands to subdirs and dependencies
var=value       rebar global variables (e.g. force=1)
command         Command to run (e.g. compile)
```

To see a list of built-in commands, execute *rebar -c*.

Type '*rebar help <CMD1> <CMD2>*' for help on specific commands.

...

As suggested in its default output, running *rebar -c* shows its list of commands. That list is too long to show in its entirety, but in general, *rebar* commands fall into the following categories:

Build commands

Support compilation of Erlang and non-Erlang sources and cleaning of build artifacts.

Project creation commands

Generate skeleton projects based on templates.

Dependency management commands

Support the retrieval, building, updating, cleaning, and removal of project dependencies.

Release generation commands

Support the creation of releases and upgrades.

Test commands

Support running unit tests, *common_test* suites, and property-based tests.

Rebar also provides other miscellaneous commands that support project activities such as documentation, generating escript archives, and starting an Erlang shell with all project files and dependencies on the load path.

Generating a rebar Project

You can use *rebar* to generate a project skeleton for a system like our base station controller example using the appropriate project template. Although our example uses only a single user-defined application, *bsc*, we use an approach that can accommodate multiple apps, since that is typical of most projects.

First, let's create an *ernie* directory and within it, an *apps* directory holding our *bsc* directory:²

```
$ mkdir -p ernie/apps/bsc
```

If our system used other applications in addition to *bsc*, they too would reside under the *apps* directory. This differs from the directory structure we described earlier for standard Erlang/OTP applications, primarily because *rebar* supports the development of applications in addition to release generation, and it would be highly unusual for the storage area for the development version of an application to also be used as its deployment area.

Because *bsc* is an application, let's use *rebar* to create an application skeleton for it:

```
$ cd ernie/apps/bsc
$ rebar create-app appid=bsc
==> bsc (create-app)
Writing src/bsc.app.src
Writing src/bsc_app.erl
Writing src/bsc_sup.erl
$ ls
src
```

Here, we can see that *rebar* created a *src* directory which it then populated with skeletons for an application resource file skeleton, an application source file, and a supervisor source file. Take special note of the application resource file skeleton *src/bsc.app.src*; *rebar* generates this file rather than an actual application resource file because later, as part of its compilation process, it takes the *src/bsc.app.src* skeleton, automatically fills in its **modules** definition with the names of all the application source modules, and generates the *ebin/bsc.app* application resource file from that. We can see this by compiling our newly-generated files:

```
$ rebar compile
==> bsc (compile)
```

2. For systems where `mkdir` does not support the `-p` option, make three separate `mkdir` calls instead.

```
Compiled src/bsc_app.erl
Compiled src/bsc_sup.erl
```

and then look at the *ebin/bsc.app* file generated by the compilation process:

```
$ cat ebin/bsc.app
{application, bsc,
 [{}description, []],
 [{}vsn, "1"],
 [{}registered, []],
 [{}applications, [kernel, stdlib]],
 [{}mod, {bsc_app, []}],
 [{}env, []],
 [{}modules, [bsc_app, bsc_sup]]]}.
```

As the last line shows, *rebar* created the **modules** definition for us based on the Erlang modules present in the *src* directory. When we add more modules, *rebar* automatically adds them to the application resource file for us during its compilation phase, which is much easier than manually editing the resource file ourselves. The only tricky part is that if you want to modify other fields of the application resource file, you have to remember to edit the *src/bsc.app.src* skeleton rather than the generated *ebin/bsc.app* file.

To run the skeleton application, we can just start a *rebar* shell, which ensures that all the appropriate project paths are on the Erlang load path. When the shell starts, we then start the application:

```
$ rebar shell
==> bsc (shell)
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V6.1 (abort with ^G)
1> application:start(bsc).
ok
```

Our generated skeleton application contains no actual code, but still, it starts and runs correctly. We can populate it more fully by copying our *bsc* example code, which is available in this chapter's directory of the GitHub repository:

```
$ rm -r src
$ cp -R <path-to-bsc-example-directory>/*
$ ls
ebin priv src
$ mv ebin/bsc.app src/bsc.app.src
```

Note that we rename the original manually-maintained *ebin/bsc.app* file to *src/bsc.app.src* so that, as we explained earlier, *rebar* can use it as a skeleton from which to generate the application resource file. Before proceeding, though, we must remember to edit *src/bsc.app.src* to remove the **modules** definition, because *rebar* automatically generates it for us.

When that's complete, we can use *rebar* to clean and compile the project:

```
$ rebar clean compile
==> bsc (clean)
==> bsc (compile)
Compiled src/phone_test.erl
Compiled src/phone_sup.erl
Compiled src/simple_phone_sup.erl
src/logger.erl:3: Warning: undefined callback function code_change/3 (behaviour 'gen_event')
src/logger.erl:3: Warning: undefined callback function handle_call/2 (behaviour 'gen_event')
Compiled src/logger.erl
Compiled src/phone.erl
Compiled src/frequency_sup.erl
Compiled src/hlr.erl
Compiled src/freq_overload.erl
src/frequency.erl:7: Warning: undefined callback function code_change/3 (behaviour 'gen_server')
Compiled src/frequency.erl
src/counters.erl:3: Warning: undefined callback function code_change/3 (behaviour 'gen_event')
Compiled src/counters.erl
src/phone_fsm.erl:11: Warning: undefined callback function code_change/4 (behaviour 'gen_fsm')
src/phone_fsm.erl:11: Warning: undefined callback function handle_event/3 (behaviour 'gen_fsm')
src/phone_fsm.erl:11: Warning: undefined callback function handle_info/3 (behaviour 'gen_fsm')
Compiled src/phone_fsm.erl
Compiled src/bsc_sup.erl
Compiled src/bsc.erl
```

If we again start a *rebar* shell, we can start our application and see that it runs as expected:

```
$ rebar shell
==> bsc (shell)
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V6.1 (abort with ^G)
1> application:ensure_all_started(bsc).

=PROGRESS REPORT==== 6-Aug-2014::16:37:59 ===
supervisor: {local,sasl_safe_sup}
    started: [{pid,<0.48.0>},
               {name,alarm_handler},
               {mfargs,{alarm_handler,start_link,[]}},
               {restart_type,permanent},
               {shutdown,2000},
               {child_type,worker}]

=PROGRESS REPORT==== 6-Aug-2014::16:37:59 ===
supervisor: {local,sasl_safe_sup}
    started: [{pid,<0.49.0>},
               {name,overload},
               {mfargs,{overload,start_link,[]}},
               {restart_type,permanent},
               {shutdown,2000},
               {child_type,worker}]

...
```

```
=PROGRESS REPORT==== 6-Aug-2014::16:37:59 ===
  supervisor: {local,bsc}
    started: [{pid,<0.58.0>},
               {name,simple_phone_sup},
               {mfargs,{simple_phone_sup,start_link,[]}},
               {restart_type,permanent},
               {shutdown,2000},
               {child_type,worker}]

=PROGRESS REPORT==== 6-Aug-2014::16:37:59 ===
  application: bsc
    started_at: nonode@nohost
{ok,[sasl,bsc]}
```

Creating a Release with rebar

The *rebar* tool wraps *reltool* in an effort to make it easier for developers to create releases. With *rebar*, creating a release is a two-step process: first creating a node, and then generating a release. Creating a node is straightforward:

```
$ cd ../..
$ mkdir rel
$ cd rel
$ rebar create-node nodeid=basestation
==> rel (create-node)
Writing reltool.config
Writing files/erl
Writing files/nodetool
Writing files/basestation
Writing files/sys.config
Writing files/vm.args
Writing files/basestation.cmd
Writing files/start_erl.cmd
Writing files/install_upgrade.escript
```

First, we create a *rel* directory in the top level of our *ernie* project directory. This directory is where we generate our release. Within the *rel* directory, we run the *create-node* command of *rebar* to create a node. This generates a number of files, including a *reltool.config* file that we need to modify to make it suitable for generating a release. Specifically, the following line must be changed:

```
{lib_dirs, []},
```

to add the *apps* directory, like this:

```
{lib_dirs, ["../apps"]},
```

Next, we want our release to contain only the applications needed for correct execution. To achieve this, we modify *reltool.config* to add the following line just after the *{incl_cond, derived}*, declaration:

```
{mod_cond, derived},
```

After adding that, you must change the `basestation` atom to `bsc` in the list of applications for the `rel` element, which also includes `kernel`, `stdlib`, and `sasl`. And finally, the `app` element shown here must be deleted:

```
{app, basestation, [{mod_cond, app}, {incl_cond, include}]}]
```

In case the descriptions for these changes aren't clear, here's the output from `diff` showing the differences from the original `reltool.config` file:

```
$ diff reltool.config.orig reltool.config
4c4
<     {lib_dirs, []},
---
>     {lib_dirs, ["../apps"]},
12c12
<         basestation
---
>         bsc
21a22
>     {mod_cond, derived},
27c28
<     {app, basestation, [{mod_cond, app}, {incl_cond, include}]}
---
>     {app, hipe, [{incl_cond, exclude}]}]
```

Depending on how your Erlang/OTP installation was built, you might find that you also need to add `{app, hipe, [{incl_cond, exclude}]}` to exclude the `hipe` application, as shown in the diff, in order to ensure your release starts up without errors.³ With these changes to `reltool.config` in place, we can proceed to the next step, which is to create a `rebar.config` file containing the following directive in the project's top-level directory:

```
{sub_dirs, ["apps/bsc", "rel"]}.
```

Until now, we haven't needed a `rebar.config` file because we relied only on the `rebar` default settings, but now we need one to inform `rebar` that the `apps/bsc` and `rel` directories are part of the project.

After the `rebar.config` file is in place, the final steps are to ensure that everything is compiled using the `rebar compile` command and then generate a release using `rebar generate`:

```
$ cd ..
$ rebar compile
==> bsc (compile)
==> rel (compile)
```

3. Both authors ran into this problem independently while writing this chapter.

```
==> ernie (compile)
$ rebar generate
==> rel (generate)
WARN: 'generate' command does not apply to directory /tmp/ernie
```

You can ignore the unfortunate warning message at the end of the *rebar generate* output; it looks scary but just reports that the *ernie* directory itself, apart from its subdirectories, contains nothing the *generate* command needs to consider. Once we've generated the release, we can verify that it works as expected:

```
$ rel/basestation/bin/basestation console
Exec: ernie/rel/basestation/erts-6.1/bin/erlexec -boot ernie/rel/basestation/releases/1/basestation
Root: ernie/rel/basestation
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V6.1 (abort with ^G)
(bsc@127.0.0.1)1> application:which_applications().
[{bsc,"Base Station Controller","1.0"},{sasl,"SASL CXC 138 11","2.4"},{stdlib,"ERTS CXC 138 10","2.1"},{kernel,"ERTS CXC 138 10","3.0.1"}]
```

Instead of the **console** argument to *rel/basestation/bin/basestation* shown in this example, which starts the application and gives us an Erlang shell, you can alternatively specify **start** to start the release in the background, **attach** to get a shell attached to an already-started release, or **stop** to stop an already-started release. Run the command *rel/basestation/bin/basestation* with no arguments to see a list of all its arguments and options.

Rebar Releases with Project Dependencies

So far our *rebar* example has been limited to including only a single application, *apps/bsc*, which has no dependencies, but in practice Erlang applications often depend on other applications. Fortunately, *rebar* is able to fetch such dependencies and compile them along with the application that depends on them. However, as we show next, ensuring that *rebar* also includes such dependencies when generating a release requires a few changes.

Let's assume we decide to change *bsc* logging use the popular open source *lager* framework so our logfiles can work with existing log rotation tools, and so that we can count on *lager* to protect our application from running out of memory should it attempt to emit a storm of log messages because of some unexpected persistent error condition. Adding a dependency for *lager* to the *apps/bsc* application is easy: we just specify it in the *apps/bsc/rebar.config* file:

```
{deps, [{lager, ".*", {git, "git://github.com/basho/lager.git", "master"}}]}.
```

This directive tells *rebar* that `lager` is a dependency, with the `".*"` indicating we don't care what version of `lager` we use, and the `{git, ...}` tuple telling *rebar* the location from which it can fetch the `lager` source code.

With this directive in place, we make sure we're in the `ernie` top-level directory and we then ask *rebar* to fetch all dependencies:

```
$ rebar get-deps
==> bsc (get-deps)
Pulling lager from {git,"git://github.com/basho/lager.git","master"}
Cloning into 'lager'...
==> lager (get-deps)
Pulling goldrush from {git,"git://github.com/DeadZen/goldrush.git",
                      {tag,"0.1.6"}}
Cloning into 'goldrush'...
==> goldrush (get-deps)
==> rel (get-deps)
==> ernie (get-deps)
```

As the *rebar* output shows, fetching the `lager` dependency resulted in `lager`'s own dependencies being fetched as well. All the dependencies are pulled into the `deps` directory:

```
$ ls deps
goldrush lager
$ rebar compile
==> goldrush (compile)
...
==> lager (compile)
...
==> bsc (compile)
==> rel (compile)
==> ernie (compile)
```

Issuing a *rebar compile* after fetching the dependencies compiles them along with the applications under the `apps` directory. Note that individual applications under `apps` might also specify dependencies in their *rebar.config* files; these too will be stored under the top-level `deps` directory.

Including the dependencies in a release requires changes to the `rel/reltool.config` file. First, the `deps` directory must be added to the `lib_dirs` definition:

```
{lib_dirs, ["../apps", "../deps"]},
```

Next, the `lager` application must be added into the `rel` definition:

```
{rel, "basestation", "1",
[
  kernel,
  stdlib,
  sasl,
  lager,
```

```
    bsc  
  ]},
```

With these changes in place, we can generate a new release and then run it:

```
$ rel/basestation/bin/basestation console  
Exec: /tmp/ernie/rel/basestation/erts-6.3/bin/erlexec -boot /private/tmp/ernie/rel/basestation/re  
Root: /tmp/ernie/rel/basestation  
Erlang/OTP 17 [erts-6.3] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10] [kernel-poll:f  
20:32:35.545 [info] Application lager started on node 'basestation@127.0.0.1'  
20:32:35.545 [info] Application bsc started on node 'basestation@127.0.0.1'  
Eshell V6.3 (abort with ^G)  
(basestation@127.0.0.1)1> application:which_applications().  
[{"bsc", "Base Station Controller", "1.0"},  
 {"lager", "Erlang logging framework", "2.0.3"},  
 {"goldrush", "Erlang event stream processor", "0.1.6"},  
 {"compiler", "ERTS CXC 138 10", "5.0.3"},  
 {"syntax_tools", "Syntax tools", "1.6.17"},  
 {"sasl", "SASL CXC 138 11", "2.4.1"},  
 {"stdlib", "ERTS CXC 138 10", "2.3"},  
 {"kernel", "ERTS CXC 138 10", "3.1"}]
```

As the console output shows, starting the `lager` application in our release results in its dependencies, such as `goldrush`, starting as well.

Relx

The `relx` tool generates releases. Like `rebar`, its goal is to simplify the process of putting a release together, but unlike `rebar`, it focuses entirely on releases and is not intended as a general Erlang build tool.

You can obtain `relx` by cloning its git repository and building it, like this:

```
$ git clone https://github.com/erlware/relx.git  
Cloning into 'relx'...  
...<snip>....  
  
Checking connectivity... done.  
$ cd relx  
$ make  
/usr/local/bin/rebar get-deps  
==> relx (get-deps)  
Pulling rebar_vsn_plugin from {git, "https://github.com/erlware/rebar\_vsn\_plugin.git",  
{branch, "master"}}  
Cloning into 'rebar_vsn_plugin'...  
...<snip>....  
==> relx (post_compile)  
/usr/local/bin/rebar skip_deps=true compile
```

```

==> relx (compile)
==> relx (post_compile)
/usr/local/bin/rebar skip_deps=true escriptize
==> relx (escriptize)

```

Once *relx* finishes building, you can copy the executable to a convenient spot in your shell's path, such as */usr/local/bin*.

Now that *relx* is available in the path, we can use it to generate a **basestation** release. To instruct *relx* what to do, our first step is to construct a *relx.config* file in the *ernie* directory (preferably right next to the *apps* directory and its contents we created in “[Rebar](#)” on page 288). Our *relx.config* file for basestation looks as follows:

```

{release, {basestation, "1.0"},  
 [bsc,  
  sasl]}.  
{extended_start_script, true}.

```

The contents of our *relx.config* file are straightforward and largely self explanatory. First, we declare our release with name “*basestation*” and version “*1.0*”. We then declare that the *basestation* release comprises the *bsc* and *sasl* applications (*kernel* and *stdlib* are also included by default). The final element of *relx.config*, *{extended_start_script, true}*, declares that we want *relx* to generate an extended start script, like the one we generated in “[Creating a Release with rebar](#)” on page 293, that supports *console*, *start*, *stop*, and other command arguments.

Once our *relx.config* file is in place, and assuming all the *bsc* code is already compiled, we run *relx* to generate a release:

```

$ relx
==> Starting relx build process ...
==> Resolving OTP Applications from directories:
    /tmp/ernie/apps
    /usr/local/lib/erlang/lib
==> Resolving available OTP Releases from directories:
    /tmp/ernie/apps
    /usr/local/lib/erlang/lib
==> Resolved basestation-1.0
==> Including Erts from /usr/local/lib/erlang
==> release successfully created!

```

We then execute the generated *basestation* start script to see that it runs properly:

```

$ _rel/basestation/bin/basestation console
Exec: /tmp/ernie/_rel/basestation/erts-6.1/bin/erlexec -boot /tmp/ernie/_rel/basestation/releases/
Root: /tmp/ernie/_rel/basestation
/tmp/ernie/_rel/basestation
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:4:4] [async-threads:10] [kernel-poll:false]

=PROGRESS REPORT==== 1-Sep-2014::22:04:25 ===

```

```

supervisor: [{local,sasl_safe_sup}]
  started: [{pid,<0.41.0>},
             {name,alarm_handler},
             {mfargs,{alarm_handler,start_link,[]}},
             {restart_type,permanent},
             {shutdown,2000},
             {child_type,worker}]

...<snip>....
```

=PROGRESS REPORT==== 1-Sep-2014::22:04:25 ===

```

supervisor: {local,bsc}
  started: [{pid,<0.51.0>},
             {name,simple_phone_sup},
             {mfargs,{simple_phone_sup,start_link,[]}},
             {restart_type,permanent},
             {shutdown,2000},
             {child_type,worker}]
```

=PROGRESS REPORT==== 1-Sep-2014::22:04:25 ===

```

application: bsc
  started_at: 'basestation@127.0.0.1'
Eshell V6.1 (abort with ^G)
(basestation@127.0.0.1)1> application:which_applications().
[{bsc,"Base Station Controller","1.0"},
 {sasl,"SASL CXC 138 11","2.4"},
 {stdlib,"ERTS CXC 138 10","2.1"},
 {kernel,"ERTS CXC 138 10","3.0.1"}]
```

As the output shows, the Erlang shell starts as expected, and we can see that the *bsc* and *sasl* applications are executing properly. But we also see that *relx* created an alternative directory named *_rel* to hold the release. If you prefer to use the *rel* directory, same as *rebar*, you can specify `{output_dir, "rel"}` in your *relx.config* file.

If you compare the approach described in “[Creating a Release with rebar](#)” on page 293 to what’s shown here, you can see that using *relx* to create a release is easier than doing it with *rebar* and *reltool*. With *relx*, we had no need to edit a generated *reltool.config* file or make any other manual changes. But interestingly, it’s also possible to use *relx* instead of *reltool* with *rebar*. To do so, modify the *rebar.config* file we created in “[Creating a Release with rebar](#)” on page 293 to add the following line:

```
{post_hooks,[{compile, "relx"}]}.
```

This directive tells *rebar* to run the *relx* tool after a successful compile. To see it in action, we first remove the *_rel* directory created earlier and then, assuming we still have the same *relx.config* file in place, run the *rebar -v compile* command:

```
$ rm -rf _rel
$ rebar -v compile
==> bsc (compile)
INFO: Skipped src/simple_phone_sup.erl
```

```

INFO: Skipped src/phone_test.erl
INFO: Skipped src/phone_sup.erl
INFO: Skipped src/phone_fsm.erl
INFO: Skipped src/phone.erl
INFO: Skipped src/logger.erl
INFO: Skipped src/hlr.erl
INFO: Skipped src/frequency_sup.erl
INFO: Skipped src/frequency.erl
INFO: Skipped src/freq_overload.erl
INFO: Skipped src/counters.erl
INFO: Skipped src/bsc_sup.erl
INFO: Skipped src/bsc.erl
INFO: No app_vars_file defined.
==> rel (compile)
==> ernie (compile)
INFO: sh info:
      cwd: "/tmp/ernie"
      cmd: relx
==> Starting relx build process ...
==> Resolving OTP Applications from directories:
      /tmp/ernie/apps
      /usr/local/lib/erlang/lib
==> Resolving available OTP Releases from directories:
      /tmp/ernie/apps
      /usr/local/lib/erlang/lib
==> Including Erts from /usr/local/lib/erlang
==> release successfully created!

```

First, *rebar* tries to recompile the applications under the *apps* directory, but performs no compilation because everything is already compiled. Once it finishes the compilation step, *rebar* then invokes *relx*, which creates a release just as in our first example where we invoked *relx* directly from the command line.

Reltool

Reltool is a release management tool shipped as part of the OTP distribution. It has a back-end that provides a lot of flexibility, likely much more than you'll ever require. It provides a graphical user interface as well. Unfortunately, its flexibility also brings complexity. You should use *reltool* only if you need a release management system you can reconfigure on the fly, capable of shipping releases in all shapes and sizes, or are writing a wrapper around your build system and exposing only the functionality specific to your needs. Fortunately, few people fall into this category; everyone else is better off using *rebar* and *relx* to create their releases.

Although it's impractical to try to cover all the details of *reltool* here, it's instructive to look at the *sys* tuple in the *reltool.config* file that *rebar* generates. The generated *sys* tuple, which we had to modify in “[Creating a Release with rebar](#)” on page 293 to get our release working, is shown as follows without any changes:

```

{sys, [
    {lib_dirs, []},
    {erts, [{mod_cond, derived}, {app_file, strip}]},
    {app_file, strip},
    {rel, "basestation", "1",
        [
            kernel,
            stdlib,
            sasl,
            basestation
        ]},
    {rel, "start_clean", "",
        [
            kernel,
            stdlib
        ]},
    {boot_rel, "basestation"},
    {profile, embedded},
    {incl_cond, derived},
    {excl_archive_filters, [".*"]}, %% Do not archive built libs
    {excl_sys_filters, ["^bin/(?!start_clean.boot)",
                      "^erts.*/bin/(dialyzer|typer)",
                      "^erts.*/(doc|info|include|lib|man|src)"]},
    {excl_app_filters, [".gitignore"]},
    {app, basestation, [{mod_cond, app}, {incl_cond, include}]}
]}.

```

Each element of the list within the `sys` tuple controls some aspect of the configuration of a generated release. Given that we had to modify the `reltool.config` file `rebar` generated for us to get our `basestation` system working, it's worthwhile knowing something about these elements, so the following list describes the purpose of some of them:

`lib_dirs`

Specifies a list of directories where applications to be included in the release may be found. By default, only the root directory of the Erlang system generating the release (the one returned by `code:root_dir/0`) is searched for applications. For example, in “[Creating a Release with rebar](#)” on page 293 we modified this element to include the `“..../apps”` directory to ensure that our `bsc` application, located under the `ernie/apps` directory, would be included in our `basestation` release.

`erts`

Specifies configuration options for the Erlang runtime system for the release. Here, the `{mod_cond, derived}` element tells `reltool` to include only the runtime modules that other applications use, while the `{app_file, strip}` element instructs `reltool` to modify runtime app files included in the release to strip out names of modules that are not included due to the `mod_cond` directive. You'll notice there's also an `{app_file, strip}` element following the `erts` element; it has the same effect but affects all app files, not just those of the runtime.

rel

The two `rel` elements specify two different releases. the first for our normal `base station` system and the second “clean” one specifying just `kernel` and `stdlib` to use for debugging in case problems arise when running the first one. Each `rel` element specifies a release name, a version, and a set of applications to include. You might recall that in “[Creating a Release with rebar](#)” on page 293 we modified the first `rel` element to change the last application in the list from `basestation`, which doesn’t exist, to `bsc`.

boot_rel

Specifies which `rel` is the default to use when booting the target system.

profile

Controls which file filters are applied to the generated specification of the target system. The `incl_sys_filters`, `excl_sys_filters`, `incl_app_filters`, and `excl_app_filters` settings allow you to control which system and app files are included in the release. Different profiles set these values in different ways. Further below this element, our generated `reltool.config` file explicitly sets the `excl_sys_filters` and `excl_app_filters` elements to filter out additional files not needed for our release. For example, if your software development team uses git for source code version control, adding the `.gitignore` file to `excl_app_filters` is a good idea as it’s useful only for git and is not needed by any Erlang application.

incl_cond

Controls which application modules are included. The value specified here, `derived`, which is the default, instructs `reltool` to include only those application modules used by other included applications.

app

Provides application-specific options. This was also an element we modified in “[Creating a Release with rebar](#)” on page 293 because, as explained earlier, the name of the application we want to be part of our release is `bsc` and not `basestation`. Multiple `app` elements can appear in the `sys` tuple, although for this particular `reltool.config` file, the `app` element appears only once and provides two options: `{mod_cond, app}` and `{incl_cond, include}`. Both of these override, for the named application, the defaults of the same option at the system level. We saw `mod_cond` earlier in the `erts` element; here it serves a similar purpose of limiting module inclusion but only for this application. The `incl_cond` option with a value of `include` means that this application should unconditionally be included in the target system whether or not other applications refer to it. This is required because `bsc` is effectively our top-level application in the `basestation` system.

For larger systems that include more applications, `reltool.config` files can be much larger and more complex. There are also quite a few other options not covered here; you should

check the `reltool` man page included in the Erlang/OTP documentation if you’re interested in those details. But in general, you’re best off using `relx` to make releases if you’re able to do so, or if not, using `rebar` to help hide the many details of `reltool`.

Wrapping Up

You’ve got to agree that everything we present here is a mouthful, and probably more detail than what you had originally bargained for when you started reading this chapter! Having said that, the steps involved in bundling up your OTP applications in a release and starting them as one unit are not many and are relatively straightforward. The reason we’ve gone into so much detail is that we want to explain not just how, but also why. You will thank us when you need to integrate Erlang/OTP releases in your build system or troubleshoot why a node that was running for years on end is refusing to start. You can’t even begin to imagine how many systems we’ve reviewed that, despite being responsible for tens of thousands of transactions per day, hour, minutes (and in many cases even seconds!) are started using `erl -s Module`, are not OTP compliant, do not have heart configured, or are not set up as embedded target systems running as daemons. Start by creating a proper OTP release, integrating the process in your build system, and the rest will follow.

The preferred way to deploy your Erlang nodes that must run years on end and be available 24/7 are as embedded target systems. You must be strict with revision control and be aware of the exact versions of your modules, applications, and configuration files. You will want to access the Erlang shell through I/O streams sent to a directory in your Erlang root directory (not `/tmp/`), allowing you to run multiple embedded nodes on that host.

Start your Erlang system as a daemon job, ensuring it is automatically started every time your computer or image is rebooted. Always ensure that you have the `erl` command at hand with a boot file that starts kernel, stdlib, and sasl, giving you access to the SASL logs on your local machine when your nodes have crashed and are refusing to start. And don’t forget to set your emulator flags, normal flags, and plain arguments adapting them to your internal operational requirements. Do you want to disable the break handler using `+Bc` but still allow the user to kill the shell? What about printing out the arguments passed to the emulator using `-emu_args` and printing startup trace reports using the `-init_debug` flag? And how do you want to implement and configure your heart script to handle emulator crashes? The combinations are many, and getting the right configuration in place that works for you and your organization can take years of operational experience and firefighting. You will eventually get there, but hopefully, taking into consideration all that we have covered in this chapter, the pager calls will be few and far apart, and never in the middle of the night.

Having said that, we know that not all systems are mission critical and require this level of supervision, complexity, and professionalism. Simple target systems can be both ac-

ceptable (and respectable) if they do their job and fulfill your requirements. If running many nodes on a single machine sharing the same Erlang installation works for you, there is no need to ship every release with its own Erlang virtual machine. You will not be able to individually upgrade applications and emulators, but then again, you might not care! The type of release that works for you, your organization, and the types of systems you are deploying is for you to judge. It can be as simple or as complicated as you need it to be. What is important is that you understand the trade-offs involved in your choices, and do things without cutting corners, because you will end up paying for it at a later date.

The process we have covered in this chapter, automated using libraries or tools, include the following steps:

1. Create a release resource file for your node, defining what will be included in your release. The *rel* file will contain all of the applications and their respective versions, together with the version of the emulator to be used in the target deployment.
2. Create a boot file containing all of the commands required to start your node.
3. Create the file structure you will deploy to your target system. It will contain the *lib*, *releases* and *bin* directories, and if you plan to ship it with its own emulator, the *erts* directory.
4. Specific to your deployment (and possibly on the target host), configure your start scripts, not forgetting your *start_erl.data* file and config file containing deployment specific configurations.

And if you are too lazy to do the above chores every time (we are), use existing tools and libraries for the bulk of the work and automate the rest.

So far in this book, you have come across the many different file types, all held together in a release. We've listed them here, as there is no better time than now to review them.

Table 11-1. Erlang/OTP file types

File Type	File Extension	Description
Erlang Module	erl	File containing the Erlang source code
Compiled Module	beam	Compiled Erlang Source code file for the Beam emulator
Application Resource File	app	File containing application resource and configuration data
Application Upgrade File	appup	File containing application upgrade data
Release File	rel	Contains release specific application and emulator versions
Release Upgrade File	relup	Contains release upgrade information
Start Script	script	Text based version of the script used to boot the system
Binary Start Script	boot	Binary version of the script used to boot the system
Configuration File	config	Contains application specific environment variables

We cover *appup* and *relup* files in [Chapter to Come]. They are used for live upgrades of the applications and regular upgrades of the emulator.

If you haven't had enough and want to read more about creating releases, head straight to the documentation that ships with Erlang/OTP. The *OTP Design Principles User's Guide* will tell you more about releases and release handling, going as far as creating the first release package ready for deployment in your target environment. The *OTP System Principles User's Guide* has sections that cover the starting, restarting and stopping of systems, as well as describing in more detail the difference between the embedded versus interactive code loading strategies. It overlaps with the *OTP Design Principles Guide*, which also covers the creation and configuration of target systems. In doing so, the user's guide introduces the *target_system.erl* module shipped in the SASL application's examples directory as well as in this chapter's GitHub repository. It is an example that automates many of the steps we covered manually when explaining how to do a release and target system, a necessity prior to the existence of *rebar*, *relx*, and *reltool*. Have a look at it, as it has for many been a good source of inspiration for those integrating Erlang into their existing build systems.

The user's guides are complemented by reference manual pages, of which the following are relevant to what we have just covered and worth mentioning:

- If you need more information on the *rel* file, look up the *rel* reference manual page. Given a *rel* file, *systools* describes the functions you need to create script, boot, and target tar files. The contents of the binary boot files and its script text counterpart are described in more detail in the *script* reference manual page. To find out more on how they are executed, review the *init* user manual pages.
- If you need to load code remotely and the example in this chapter is not enough, the *erl_boot_server*, *erl_prim_loader*, and *init* user manual pages will help you.
- The *erl* and *init* manual pages describe most the emulator flags and command line flags, some of which we have not covered in this chapter. For plain arguments, you will have to refer to the user manual pages of the modules and applications using those arguments.
- The *heart* manual page is the place to look for more information on automated restarts, including configuration details and required environment variables when implementing your script. You will find the environment variables described in the *erl* manual pages.
- If you are running on Windows, read the *start_erl* manual page. It is the equivalent of the start command we have been using in this chapter, allowing you to start your embedded system in Windows environments.

Reltool has both a user's guide and reference manual page you will have to study in detail if your system requires the configuration complexity not handled by *rebar*, which you

can find at <https://github.com/rebar/rebar> or *relx*, available from <https://github.com/erlware/relx>.

What's Next?

Erlang has been called the language of the system. It is not just a language suitable for solving a particular type of problem, but rather a language and a set of tools that allow you to develop, deploy, and monitor predictable and maintainable systems. Whilst in this chapter we have covered how to package and deploy your first target systems, that is just the beginning of your adventure. What we cover next is how to manage bug fixes and deploy new functionality by doing live upgrades. We do so by introducing the upgrade tools and functionality that come as part of OTP and its behaviours. You've heard about Erlang achieving five nines availability, software maintenance and upgrades included? Continue on to find out how we do it.

CHAPTER 12

Release Upgrades

After your system goes live, it churns away in the background handling requests day in and day out. It self-heals when issues occur and restarts automatically after power outages or system reboots. But as with any piece of software, you are bound to continue optimizing it, fixing bugs as they are reported and adding new features. Irrespective of having thousands of instances of your coffee machine running on dedicated hardware monitored through a wireless link, or any other system whose requirements state that it must service its requests with 100% availability, upgrades included, then Erlang/OTP's software upgrade capabilities are something to study carefully. Imagine you not being able to have your morning coffee because of an ongoing firmware upgrade of your office coffee machine!

The built-in functionality in the Erlang VM that allows dynamic module loading might work for simple patches where the upgrade is backward-compatible. But have you thought of the cases where you've changed the functional API? Or where a process running a call to completion with an old version of the code cannot communicate with a process running a new version because of a change in the protocol? How do you handle state changes in your loop data between releases, database schema changes, and even more importantly, what if an upgrade fails and you need to downgrade?

Complex systems need to be upgraded in a coordinated and controlled manner. The built-in functionality used to dynamically load new modules, like everything else Erlang and OTP, provides the foundations used to build the tools which coordinate and control these upgrades, greatly reducing and even hiding their complexity. But before introducing the tools themselves, let's review the semantics, terminology, and most commonly used functions relevant to our example to ensure we are all on the same page.

Software Upgrades

We covered module upgrades in [Link to Come]. You might recall that you can load a new module in the Erlang runtime environment using the shell command `l(Module)`, by calling `code:load_file(Module)`, and by compiling the source code using `c(Module)` or `make:files(ModuleList, [load])`. At any one time, your runtime environment can have two versions of the code loaded at the same time. We refer to them as the *old* and *current* versions. A process running the old module version will continue doing so until it issues a fully qualified function call, i.e., a call of the format `Module:Function(...)`, where the module name is used as a prefix to the function.

When a fully qualified function call occurs, the runtime checks to ensure that the process is running the current version of the code. If it is, the call continues using the current code. But if the process is still running the old version, the pointer to the code is switched to the current version before the call is made.

Calls to library modules have to be fully qualified, because you are calling another module, so such a call will automatically use the current version. Recursive calls controlling process receive-evaluate loops, however, tend to recurse locally without a fully qualified call. We need to either change these local calls to fully qualified ones, or add a new message that triggers the fully qualified function in one place in the receive-evaluate loop. Depending on the complexity of the upgrade, this function could either call the loop function in the new module or call a hook in the new module that handles any change of the process state, including loop data, ets tables, and database schemas, before returning into the loop.

When not executing a fully qualified call, a process running the current version of a module will continue running it even after a new version is loaded in the system. If a process is already running the old version of a module—not the current version—when yet an even newer version is loaded, the process will be unconditionally terminated. Processes will also be unconditionally terminated if they are running an old module version forcefully removed using the `code:purge(Module)` call.

The two-module version limit is legacy debt from a design decision taken to simplify the JAM virtual machine (the most used VM at the time) and to preserve memory in an architecture where memory was scarce. Today, the right design decision would be to allow an unlimited number of modules in the runtime, and garbage collect them when they're no longer in use. In the JAM, in order to garbage collect code, you would have had to go through the stack of each process and look at the return addresses of each function call to work out what module version a process was using. This would have been a very time consuming activity they preferred to avoid, simplifying it slightly with the two module approach.

With two versions of the code allowed in the runtime system, we need a way to determine the current version of the module. The `-vsn(Version)`. attribute helps us achieve exactly that. `Version` can be any Erlang term, but it is most commonly a string, number, or atom. More often than not, it is set by a script triggered by the revision control system when committing the code to the repository. Placing the `vsn` attribute at the beginning of the module with the other attributes gives us the ability to determine the version of the code we are upgrading from, using it to control changes to the state, database schemas, protocols, and other non-backward-compatible internal data formats. You can find the version of the current module using the `Mod:module_info/0,1` call.

The `vsn` attribute is not mandatory. If omitted, the compiler generates it at compile time using the `beam_lib:md5/1` call to generate a 128-bit *md5 digest* of the module. The md5 digest is based on properties of the module, but excludes compile date and other attributes that are irrelevant to the code, since they may change without the code itself changing. This guarantees that a version will be tagged with the same 128-bit key regardless of compilation time, spaces, carriage returns, or comments in the code.

Remember the coffee finite state machine we looked at in [Link to Come]? Let's dust off the Erlang version and compile it to better understand how the `vsn` module attribute works. If you are using modules from the book repository, the module we are using is under `ch12/erlang/coffee.erl.original`. Don't forget to change its filename to `coffee.erl`:

```
1> c(coffee).
{ok,coffee}
2> coffee:module_info(attributes).
[{vsn,[293551046745957884913825426256179654413]}]
3> {ok, {coffee, MD5Digest}} = beam_lib:md5(coffee).
{ok,{coffee,<<220,215,224,7,110,247,231,148,86,224,44,74,197,2,111,13>>}}
4> <<Int:128/integer>> = MD5Digest, Int.
293551046745957884913825426256179654413
```

In shell command 2, a call to `coffee:module_info/1` returns the md5 digest in the `vsn` module attributes, something we confirm in shell commands 3 and 4 by getting the digest from the module and reversing the digest process. Let's now add the `vsn` directive manually in our module and recompile:

```
-module(coffee).
-export(...).

-vsn(1.0).

...
```

This will ensure the compiler will not override the version with the md5 digest and sets it instead to 1.0:

```
5> c(coffee).
{ok,coffee}
```

```
6> coffee:module_info(attributes).
[{{vsn,[1.0]}}
```

Let's continue working with the Erlang version of the coffee machine fsm, adding a new upgrade message that triggers a fully qualified function call. This will allow us to upgrade the server in a controlled way, understanding the how and why of all the steps involved in the process. When done, we'll explore how it is done using OTP.

The First Version of the Coffee FSM

You might recall that the Erlang version of the coffee finite state machine consisted of three states, *selection*, *payment*, and *remove* (Figure 12-1). In our software upgrade example, we add a new state called *service*, which allows us to open the cabinet door and service the coffee-maker. But before going there, let's add some generic code that executes the fully qualified call, giving us a baseline we can use to perform the upgrade itself. We can do this either by making every call to the receive-evaluate loop fully qualified, or by sending the process a message that triggers the fully qualified call.

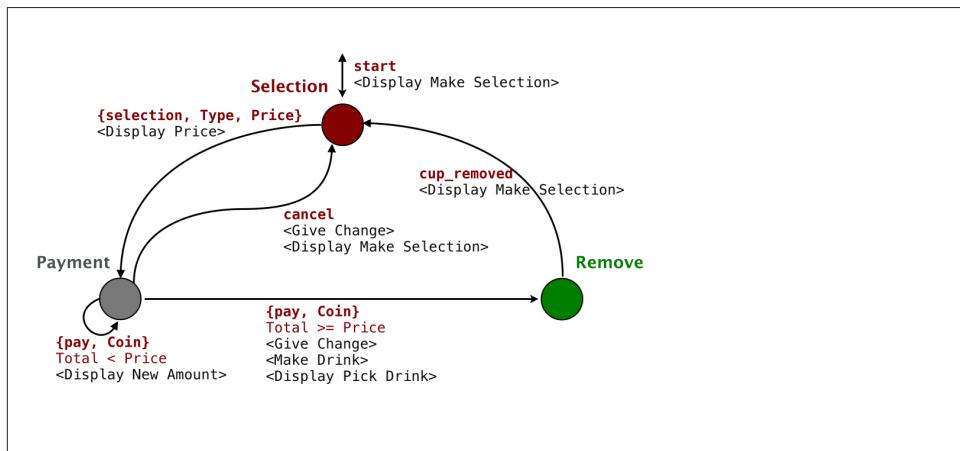


Figure 12-1. Coffee FSM

The recommended approach to upgrading your code is to separate the loading of the new module from each process's trigger of the upgrade. In our generic upgrade code, we load the module using `module:load_file/2`. We then inform the processes that have to trigger an upgrade through a fully qualified call by sending them the `{upgrade, Data}` message.

Data is an opaque datatype containing upgrade-specific information used by the new module. It is there to act as a placeholder and to future-proof the code, allowing us to manipulate the process state in conjunction with the transition to the new module. As an example, pretend we are upgrading our frequency server and want to add more

frequencies. We could use Data to pass the new frequencies to the server during the upgrade. A process that receives the upgrade message and its data then issues a fully qualified function call to `code_change/2`, where the first argument is the process state and the second is Data. In this function, we could append the new frequencies to the list of available ones, entering the receive-evaluate loop in the new module with the newly updated loop data.

Let's have a look at what the generic upgrade code looks like. And notice that we have added a version number to the module:

```
-module(coffee).
-export(...).
-export([..., code_change/2]).
-vsn(1.0).

...

%% State: drink selection

selection() ->
    receive ... {upgrade, Extra} ->      ?MODULE:code_change(selection, Extra); ...
end.

%% State: payment
payment(Type, Price, Paid) ->
    receive ... {upgrade, Extra} ->      ?MODULE:code_change({payment, Type, Price, Paid}, Extra);
end.

%% State: remove cup

remove() ->
    receive ...
        {upgrade, Data} ->
            ?MODULE:code_change(remove, Extra);
        ...
    end.

code_change({payment, Type, Price, Paid}, _Extra) ->
    payment(Type, Price, Paid);
code_change(State, _Extra) ->
    State().
```

Note how we need to handle the `{upgrade, Extra}` message in all states. Upon receiving it, we do a fully qualified function call to `code_change/2`, where the first argument is the fsm state and loop data, and the second is Extra, which we transparently pass to the call. The `code_change/2` function in the new module provides a place to change the old process state to one compatible with the new code base, possibly using Extra. Changes in the process state could include adaptations to the loop data format and contents, database schema changes, synchronization with other processes, changing process flags, or even going as far as manipulating messages in the mailbox.

Once done, `code_change/2` yields control by calling the tail recursive function returning the process to its new receive-evaluate loop. In our example, these functions are the fsm state functions `selection/0`, `payment/3` and `remove/0`. This is the first version of the module, so we do not expect the `code_change/2` clauses we've added to do anything; they simply return to the state from which the call originated. Adding these clauses avoids the *undefined function* runtime error that we explained will result if you attempt an upgrade and a process is running an old version of the `coffee` module.

This is our baseline code. Let's compile it, start the Erlang VM, and get our coffee FSM up and running, making sure it works before creating a new version of the module and doing a software upgrade.

```
$ cd erlang/
$ cp coffee.erl.1.0 coffee.erl
$ erl -make
Recompile: coffee
Recompile: hw
$ erl -pa patches
Erlang/OTP 17 [erts-6.1] [source] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V6.1  (abort with ^G)
1> coffee:start_link().
Machine:Rebooted Hardware
Display:Make Your Selection
{ok,<0.36.0>}
2> coffee:module_info(attributes).
[{vsn,[1.0]}]
3> coffee ! {upgrade, {}}.
{upgrade, {}}
4> coffee:module_info(attributes).
[{vsn,[1.0]}]
```

Note how in shell command 3 we trigger an upgrade without having loaded a new version of the fsm. This results in an execution of the `code_change/2` call in the current version of the module.

Adding a state

Let's add a state used for servicing the coffee finite state machine. It gets triggered when the coffee FSM is in the *selection* state and the cabinet door is opened. In any other state, the open door event is ignored. As we can see in [Figure 12-2](#), closing the cabinet door triggers a reboot of the hardware and a transition back to the *selection* state. The closing door event is ignored in all other states.

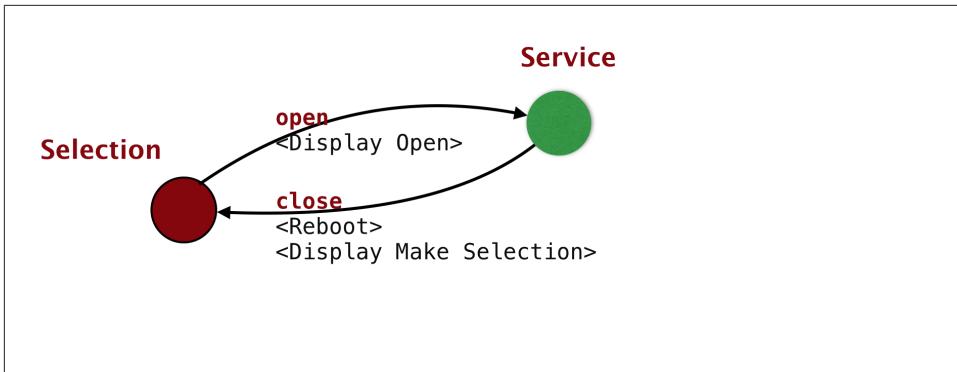


Figure 12-2. Service State

We've opted to keep the example simple, but could have easily inserted locks in the hardware by upgrading *hw.erl* to add the functions `hw:lock()` and `hw:unlock()`. These would represent coffee-maker safeguards that would ensure that the coffee machine door could be opened only in the *selection* state and would keep it locked when the machine is in other states.

Let's look at the new module, where we've highlighted the changes from version 1.0. The major differences are the addition of the *service* state, the *open* and *close* events, and actions executed in the `code_change/2` function clauses.

First we see the client functions `open/0` and `close/0`, which respectively generate an event when the coffee machine door is opened and closed. In state selection, upon receiving the *open* event, we show *Open* in the display and transit to the *service* state.

The *service* state ignores all events except for users inserting coins and the closing of the coffee machine door. Upon closing the door, the hardware is rebooted and the display instructs the customer to make a selection. The *open* and *close* events are ignored in all other states.

```

-module(coffee).
-export([tea/0, espresso/0, americano/0, cappuccino/0,
        pay/1, cup_removed/0, cancel/0, open/0, close/0]).
-export([start_link/0, init/0, code_change/2]).
-vsn(1.1).

start_link() ->
    ...

open() -> ?MODULE ! open.
close() -> ?MODULE ! close.

...

selection() ->

```

```

receive {selection, Type, Price} ->      hw:display("Please pay:~w",[Price]), payment(Type)
end.

...
service() ->
receive close ->    hw:reboot(),    hw:display("Make Your Selection", []), service();
end.

...
code_change({payment, _Type, _Price, Paid}, _Extra) ->
hw:return_change(Paid),
hw:display("Make Your Selection", []),
selection();
code_change(State, _Extra) ->
State().

```

In our code change function, if a user has selected a drink and is paying for it, we return whatever amount they have paid and transit to the *selection* state. For all other states, we transition back to the state we were in prior to the upgrade. In our example, we don't need *Extra*, but as you are preparing the code for potential upgrades without knowing what these upgrades will be, the argument is worth including to future proof your code and allow you to pass the variable and use it to change the process state in a later upgrade.

We place version 1.1 of the source code in the patches directory and compile it. Note how we started the Erlang runtime system with the `-pa patches` directive. When we first start the coffee fsm, this directory is empty. As we find and fix bugs, we place the new beam files here. Because this directory appears first in the code search path, beam files we put here will override beam files of the same module further down in the code search path.

```

$ cd patches/
$ erl -make
Recompile: coffee

```

Using the same Erlang node where we started version 1.0 of the coffee fsm, we load the new version of the module by calling `code:load_file/1`. The code server looks for the first version of the coffee beam file in its code search path, and because the patches directory is at the top of list, the version we just compiled is chosen. The success of the operation is confirmed in shell command 6, showing us that the *version* attribute is now set to 1.1. At this point, we have two versions of the coffee module loaded in the runtime system: the current one we just loaded and the old one used by the fsm process. When we order an espresso in shell command 7 and start paying for it in the subsequent command, the shell does a fully qualified call using the *current* version of the code, namely the one we just loaded. The fsm process, however, is still using the *old* version of the coffee module.

If we were to load another version of the coffee module at this point, even 1.0, the coffee fsm process would be terminated because it is running the now deleted old version of the code. The current version would become the old version, while the newly loaded module would become the current one. We are not doing it in our example, but try it out yourself if you've compiled the code and are following along.

```
5> code:load_file(coffee).
{module,coffee}
6> coffee:module_info(attributes).
[{{vsn,[1.1]}}
7> coffee:espresso().
Display:Please pay:150
{selection,espresso,150}
8> coffee:pay(100).
Display:Please pay:50
{pay,100}
9> coffee ! {upgrade, {}}.
Machine:Returned 100 in change
Display:Make Your Selection
{upgrade,{}}
10> coffee:open().
Display:Open
open
11> coffee:espresso().
{selection,espresso,150}
12> coffee:close().
Machine:Rebooted Hardware
Display:Make Your Selection
close
```

In shell command 9, we trigger an upgrade. This causes the coffee machine fsm, currently in state `pay`, to call `code_change/2` in the new module. It returns the change and, thanks to the new state service, now allows us to open and close the machine door so we can service it.

So this is how basic Erlang can handle upgrades. The generic code is the handling of the `{upgrade, Extra}` message and the calling of `code_change/2`, which does a fully qualified call back to the receive-evaluate loop. This will be the same across all processes. What will differ among processes is what we do in `code_change/2` depending on the loop data, the process state, and the contents of `Extra` itself. Using these foundations, let's read on and see how we do it with OTP.

Creating a Release Upgrade

To upgrade releases using the tools and design principles provided by OTP, we have to start with a baseline consisting of a properly packaged and deployed OTP release following the principles covered in [Link to Come]. We also need:

- One or more new versions of existing applications
- Zero or more new applications
- An application upgrade file for each application that has been changed
- Release resource and release upgrade files

The modules containing the bug fixes and new features are packaged into new or existing applications, where their version numbers are bumped up. *Application upgrade files* contain commands that tell us how to upgrade or downgrade from one application version to another. The release resource file, covered in [Link to Come], is the file containing the emulator and application versions that make the new release. Together with the application upgrade files and the release file of the baseline system we are upgrading from, the new release file is used to generate the *release upgrade file*. This file contains all the commands that have to be executed during the upgrade itself. After having installed the new code on the target machine, we run the instructions in the release upgrade file. If anything fails, the system is restarted using the old release. Through tests and observations, you have to determine if the system is stable. If so, it is made permanent. Restarting the system prior to it being made permanent will result in the old release being restarted. Let's do an upgrade and see how the different steps and components all work together.

In the section of this chapter's code repository, you will find the files used to create our first deployment. They are the output of the exercises we left you to do in [Link to Come]. We've taken the *coffee_fsm.erl* example and created an OTP application out of it, supervisor and application behaviour files included. We also created the *coffee.app* file and placed it in the *ebin* directory. Download it, compile it and make sure you can get it up and running:

```
$ cd coffee-1.0/src/ ; erl -make ; mv *.beam ..ebin/ ; cd ../../
Recompile: coffee_app
Recompile: coffee_fsm
coffee_fsm.erl:2: Warning: undefined callback function code_change/4 (behaviour 'gen_fsm')
coffee_fsm.erl:2: Warning: undefined callback function handle_event/3 (behaviour 'gen_fsm')
coffee_fsm.erl:2: Warning: undefined callback function handle_info/3 (behaviour 'gen_fsm')
coffee_fsm.erl:2: Warning: undefined callback function handle_sync_event/4 (behaviour 'gen_fsm')
Recompile: coffee_sup
Recompile: hw
$ erl -pa coffee-1.0/ebin/
Erlang/OTP 17 [erts-6.1] [source] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V6.1  (abort with ^G)
1> application:start(sasl), application:start(coffee).

...<snip>...

=PROGRESS REPORT==== 2-Nov-2014::21:11:24 ===
application: coffee
```

```

started_at: nonode@nohost
ok

2> coffee_fsm:module_info(attributes).
[{behaviour,[gen_fsm]},{vsn,['1.0']]}
```

Even if the *coffee* application directory is not in the *lib* directory (yet), we've given it a version number for the sake of clarity. Note how, when compiling the code, we get the following warning:

```
Warning: undefined callback function code_change/4 (behaviour 'gen_fsm')
```

Up to now, we asked you to patiently bear with us and ignore this warning message, but no more. You should by now have understood what it is for and figured out how we are going to use it when we upgrade the *coffee_fsm* module. Note also how, when retrieving the module attributes in shell command 2, we get both the behaviour type and the current module version number.

With our application running, let's create the boot file, a release file, and the target directory structure. We use the empty *sys.config* and *coffee-1.0.rel* files in the book code repository. If you are typing along as you are reading this, getting your own version up and running, don't forget to update the standard OTP application and erts versions in the rel file to the Erlang release you are currently using. If you are not typing along, or do not have access to the code, we've included the contents of the *sys.config* and *coffee-1.0.rel* for your convenience:

```

$ cat sys.config
[].

$ cat coffee-1.0.rel
{release,
 {"coffee","1.0"},
 {erts, "6.2"},
 [{kernel, "3.0.3"},
  {stdlib, "2.2"},
  {sasl, "2.4.1"},
  {coffee, "1.0"}]}.

$ mkdir ernie
$ erl
Erlang/OTP 17 [erts-6.1] [source] [smp:4:4] [async-threads:10] [kernel-poll:false]

Eshell V6.1 (abort with ^G)
1> systools:make_script("coffee-1.0", [{path, ["coffee-1.0/ebin"]}]). 
ok
2> systools:make_tar("coffee-1.0", [{erts, "/usr/local/lib/erlang/"}, {path, ["coffee-1.0/ebin"]}], 
                     {outdir, "ernie"}).
ok
3> halt().

$ cd ernie; tar xf coffee-1.0.tar.gz; rm coffee-1.0.tar.gz; mkdir bin; mkdir log
$ cp erts-6.1/bin/run_erl bin/.; cp erts-6.1/bin/to_erl bin/.
$ cp erts-6.1/bin/start.src bin/start; cp erts-6.1/bin/start_erl.src bin/start_erl
```

```

$ vim bin/start
$ diff erts-6.1/bin/start.src bin/start
27c27,28
< ROOTDIR=%FINAL_ROOTDIR%
---
> ROOTDIR=/Users/francescoc/OTPBOOK/current/code/ch12/ernie
$ echo '6.1 1.0' > releases/start_erl.data

```

Hello Joe, coffee machine working? Seems to be. We now need to create the *releases/RELEASES* file, required for upgrading and downgrading releases. We got away without it in the previous chapter, as it is only really required when downgrading to this release after a failed upgrade. When we do an upgrade and this file is not present, a new one is created, but it only contains information of the upgraded release. This is fine if the upgrade is successful, because when we upgrade a second time, we should be able to downgrade to the first upgraded version. The downside is that if the first upgrade fails, we are unable to downgrade back to the original version once we've made the upgrade permanent, and we'll have to reinstall the node instead.

```

$ bin/start
$ bin/to_erl /tmp/
Attaching to /tmp/erlang.pipe.1 (^D to exit)

1> application:which_applications().
[{coffee,[],"1.0"},{sasl,"SASL CXC 138 11","2.4"},{stdlib,"ERTS CXC 138 10","2.1"},{kernel,"ERTS CXC 138 10","3.0.1"}]
2> RootDir = code:root_dir().
"/Users/francescoc/OTPBOOK/current/code/ch12/ernie"
3> Releases = RootDir ++ "/releases".
"/Users/francescoc/OTPBOOK/current/code/ch12/ernie/releases"
4> RelFile = Releases ++ "/coffee-1.0.rel".
"/Users/francescoc/OTPBOOK/current/code/ch12/ernie/releases/coffee-1.0.rel"
5> release_handler:create_RELEASES(RootDir, Releases, RelFile, []).
ok

```

The *RELEASES* file contains a list with an entry for every release that has been installed. Every entry has information similar to that found in the rel file, including release and erts versions. Together with the application names and versions, however, an absolute path to the application directory is also included. While the first version of the *RELEASES* file will contain a single entry on the first release, subsequent upgrades will result in multiple entries:

```

%% File:RELEASES
[{release,"coffee","1.0","6.1",
 [{kernel,"3.0.1",
   "/Users/francescoc/OTPBOOK/current/code/ch12/ernie/lib/kernel-3.0.1"},{stdlib,"2.1",
   "/Users/francescoc/OTPBOOK/current/code/ch12/ernie/lib/stdlib-2.1"},{sasl,"2.4",
   "/Users/francescoc/OTPBOOK/current/code/ch12/ernie/lib/sasl-2.4"}]}

```

```

"/Users/francescoc/OTPBOOK/current/code/ch12/ernie/lib/sasl-2.4"],
{coffee,"1.0",
 "/Users/francescoc/OTPBOOK/current/code/ch12/ernie/lib/coffee-1.0"]},
permanent]}.

```

The Code To Upgrade

Now that we have our first OTP compliant release up and running, let's create the new version of the `coffee_fsm` module, adding the new *service* state and its client functions. We start by bumping up the version attribute to 1.1. It might not mean much now, but if you have kept the discipline of bumping up the version (or doing it automatically through a script when checking in your code or building your release), payback time will come many upgrades later, in the early hours of the morning, when you are figuring out why the version of the code you think is running in production is actually not the one that should be running.¹

We export the state functions `service/2` and `service/3` (you might recall that the `gen_fsm` callbacks `State/2` handles asynchronous events and `State/3` handles synchronous ones). We also export two client functions, `open/0` and `close/0`, which asynchronously send the coffee machine door `open` and `close` events to the FSM. And finally, we export `code_change/4`, a behaviour callback used to update the state of the behaviour. All these should be familiar from reading “[Adding a state](#)” on page 312.

```

-module(coffee_fsm).
-behaviour(gen_fsm).
-vsn('1.1').
-export([start_link/0, init/1]).
-export([selection/2, payment/2, remove/2, service/2]).
-export([americano/0, cappuccino/0, tea/0, espresso/0,
        pay/1, cancel/0, cup_removed/0, open/0, close/0]).
-export([stop/0, selection/3, payment/3, remove/3, service/3]).
-export([terminate/3, code_change/4]).

start_link() ->
    gen_fsm:start_link({local, ?MODULE}, ?MODULE, [], []).

...
cup_removed() -> gen_fsm:send_event(?MODULE,cup_removed).
open()       -> gen_fsm:send_event(?MODULE, open).
close()      -> gen_fsm:send_event(?MODULE, close).

...

```

In state selection, we handle the `open` event. This is the only state-event combination in which the transition to our new *service* state is allowed. In the *service* state, upon receiving the `close` event, we transition back to the *selection* state. In all other states, `open`

1. Please don't ask us about this one!

and *close* events are ignored. The `service/3` state callback function also handles the synchronous *stop* event, which stops the finite state machine and triggers a call to `terminate/3`.

```
%% State: drink selection
selection({selection, Type, Price}, LoopData) ->
    hw:display("Please pay:~w",[Price]),
    {next_state, payment, {Type, Price, 0}};
selection({pay, Coin}, LoopData) ->
    hw:return_change(Coin),
    {next_state, selection, LoopData};
selection(open, LoopData) ->
    hw:display("Open", [ ]),
    {next_state, service, LoopData};
selection(_Other, LoopData) ->
    {next_state, selection, LoopData}.

%% State: service
service(close, LoopData) ->
    hw:reboot(),
    hw:display("Make Your Selection", []),
    {next_state, selection, LoopData};
service({pay, Coin}, LoopData) ->
    hw:return_change(Coin),
    {next_state, service, LoopData};
service(_Other, LoopData) ->
    {next_state, service, LoopData}.

...
service(stop, _From, LoopData) ->
    {stop, normal, ok, LoopData}.

...
```

We now need to implement our new `code_change/4` callback function. This callback takes three arguments when called within an event handler or a generic server, and four when called from within a finite state machine:

```
Mod:code_change(Vsn, State, LoopData, Extra) -> {ok, NewState, NewLoopData} |
                                                {error, Reason} %Finite State Machines
Mod:code_change(Vsn, LoopData, Extra) -> {ok, NewLoopData} | %Generic Servers
                                                {error, Reason}
Mod:code_change(Vsn, LoopData, Extra) -> {ok, NewLoopData} | %Event Handler
```

The first argument, `Vsn`, is the version of the old module you are upgrading from, or the version you're going to when downgrading back to the old module. In this example it is `1.0`, and could also be `{down, 1.0}` when downgrading to a previous version. When a module does not have a version directive, use the `md5` module checksum, and when versions do not matter at all, use wildcards.

`State` is passed only to finite state machines, and contains the state the fsm was in when the upgrade was triggered.

The final two arguments include the loop data and any extra arguments passed in the upgrade instructions specific for this module. In our example, we don't do anything with the `_Extra` arguments, nor do we manipulate the loop data.

The `code_change/4` callback, when successful, has to return `{ok, NewState, NewLoopData}`. Returning `{error, Reason}` will cause the upgrade to fail and the node to restart the previous version when dealing with generic servers or finite state machines. In the case of event handlers, returning anything other than `{ok, NewLoopData}` or terminating abnormally will cause the handler to be removed from the event manager, but the node will not revert to its previous version and be restarted.

This is how our coffee machine's `code_change/4` OTP callback function looks like:

```
code_change('1.0', State, LoopData, _Extra) ->
    {ok, State, LoopData};
code_change({down, '1.0'}, service, LoopData, _Extra) ->
    hw:reboot(),
    hw:display("Make Your Selection", []),
    {ok, selection, LoopData};
code_change({down, '1.0'}, payment, {_Type, _Price, Paid}, _Extra) ->
    hw:return_change(Paid),
    hw:display("Make Your Selection", []),
    {ok, selection, {}};
code_change({down, '1.0'}, State, LoopData, _Extra) ->
    {ok, State, LoopData}.
```

We've changed the behaviour slightly from the Erlang example. Regardless of the state we are in, payment included, we do not change the loop data and remain in the state we were originally in. This is normal in cases where we simply add functionality or a state. If we were to change the state or loop data as part of the upgrade, it would occur here.

If an upgrade failure triggers a downgrade and we are in the `service` state, we reboot the hardware and return to the `selection` state, because the `service` state does not exist in version 1.0. If the user is in the process of paying for a coffee, we return whatever amount they have paid and move back to the `selection` state. Downgrades, as we will see, will cause the system to reboot and start the old version from scratch. So if your old version is dependent on some persistent values that were set at startup and later changed, make sure your `code_change` reverts to the correct values.

When we are done implementing the new modules, we package them in an application, bumping up the version. In our case, our new coffee application version is "1.1", whereas the versions of the `hw`, `coffee_app` and `coffee_sup` modules are the same as in the application version "1.0", and the version of the `coffee_fsm` module is now 1.1.

Application Upgrade Files

Now that we have the new version of our coffee machine up and running, we need an application upgrade file containing a set of actions to be executed when upgrading or downgrading to other versions of the same application. Application upgrade files are similar in concept to *app* files, because they are used by *systools* to create the upgrade script. They have the name of the application with the *appup* suffix and are placed in the *ebin* directory, alongside the *app* file.

Go into the Erlang root directory of your installation and type **ls lib/*/ebin/*.appup**. The call will return all application upgrade files installed as part of your Erlang release. Starting with Erlang/OTP version 17, *appup* files are included in every application. Prior to that, you could upgrade only some core applications, as not all applications provided an *appup* file. Let's have a look at the *sasl.appup* file for version 2.4 of the applications:

```
{"2.4",
  %% Up from - max one major revision back
  [{<<"2"\.\.4(\\".[0-9]+)*">,[restart_new_emulator]},           %% 17
   {<<"2"\.\.3(\\".[0-9]+)*">,[restart_new_emulator]}],           %% R16
  %% Down to - max one major revision back
  [{<<"2"\.\.4(\\".[0-9]+)*">,[restart_new_emulator]},           %% 17
   {<<"2"\.\.3(\\".[0-9]+)*">,[restart_new_emulator]}]           %% R16
}.
```

Based on its contents, we should be able to figure out what happens when application version 2.4 is upgrading or downgrading between OTP versions R16 and 17. When upgrading from application version 2.4.X or 2.3.X or downgrading to 2.4.X or 2.3.X (where X is the patch release number), we need to restart the emulator. Notice how regular expressions, placed in binaries, create a range of sub releases and point to a list of upgrade and downgrade instructions. Instead of regular expressions, you can also use strings defining specific versions, e.g. "2.4.5".

Inspect any other *appup* files in the release you have installed and you will notice they all follow the following format:

```
{Vsn,
  [{UpFromVsn1, InstructionsU1}, ...,
   {UpFromVsnK, InstructionsUK}],
  [{DownToVsn1, InstructionsD1}, ...,
   {DownToVsnK, InstructionsDK}]}.
```

Vsn is the application version you are upgrading to. *UpFromVsn* are the application versions you will be upgrading from. In case something goes wrong, *DownToVsn* are the application versions you will be able to downgrade *Vsn* back to. *Vsn* can be either a string with the exact version numbers, or a binary containing a regular expression allowing you to describe multiple application versions on which to execute upgrade and downgrade instructions. If you have installed version 17 or later, look at the various *appup*

files and you will notice that OTP standard applications usually allow you to upgrade from or downgrade to two revisions.

If you plan on using regular expressions, the following constructs will be more than enough to denote ranges of versions:

- A period (.) matches any character, so the expression 1.3 will match any combination of characters starting with 1 and ending with 3.
- An asterisk (*) matches the preceding element zero or more times.
- A plus sign (+) matches the preceding element one or more times.
- A question mark (?) matches the preceding element zero or one times.
- The range [0-9] matches the elements between 0 and 9.
- The sequence \\. returns a period. You need to escape the backslash because Erlang itself uses the backslash to escape characters.
- A caret (^) at the beginning of the regular expression anchors the match to the beginning of the version string.
- A dollar sign (\$) at the end of the regular expression anchors the match to the end of the version string.

As an example, <<"^1\\.\\.[0-9]+\$">> matches all versions of 1.X, <<"^1\\.\\..\\.[0-9]+ \$">> matches all versions of 1.0.X and <<"^1\\.\\.([0-9]*\\.)?\\..\\.[0-9]+\$">> will match versions 1.X or 1.X.X, where X is an integer.

If you are not sure of your regular expressions, test them using `re:run(Vsn, RegExp)`, which returns `nomatch` if the match fails and `{match, MatchData}` otherwise. You can read more about the format of regular expressions in the manual pages for the `re` module.

Browsing the `appup` files, you should have come across lists of actions associated with different versions. They include elements such as `restart_new_emulator` (used only when upgrading the erts, kernel, stdlib and sasl applications), `load_module`, `apply`, `restart_application`, and `update`. In some cases, when no actions have to be taken, you will find a tuple `{Vsn, [], []}` with two empty lists. Actions are divided into high-level instructions and low-level ones. High-level instructions are translated to low-level ones when creating the release upgrade script.

Let's go back to our example, where we are going to upgrade the coffee fsm application from version 1.0 to 1.1. It will not be a complicated upgrade because no drivers or NIFs are involved, no new applications or modules are added to the release, and there are no inter-process and inter-module dependencies to worry about, let alone internal state or loop data changes. Behind the scenes, all we need to do is suspend all behaviour processes with a dependency on the module `coffee_fsm`, load the new version of the module, purge the old one, call `code_change`, and resume the processes ([Figure 12-3](#)).

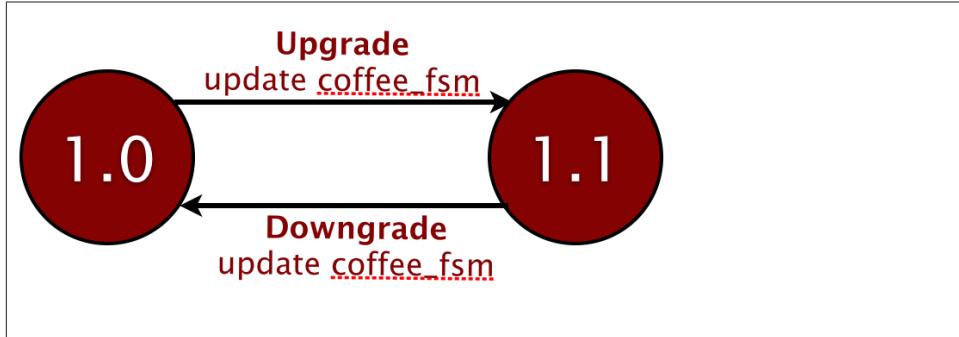


Figure 12-3. Coffee FSM

Our `coffee.appup` file contains a tuple containing the version we are upgrading to along with the high level upgrade and downgrade instructions. In our case, `update` loads the new module and `{advanced, []}` triggers the `code_change/4` call, passing `[]` as the last argument.

```

%% File:coffee.appup
{"1.1", % Current version
 [{"1.0", [{update, coffee_fsm, {advanced, []}}]}], % Upgrade from
 [{"1.0", [{update, coffee_fsm, {advanced, []}}]}] % Downgrade to
}.

```

During both an upgrade and a downgrade, the `update` high-level instruction will translate to the following set of low-level instructions:

1. Search for the object code for the module, load it from file, and cache it. This ensures that time-consuming file operations are done prior to suspending the processes.
2. Suspend any process that specified the module as a dependency in its child specification, using `sys:suspend/1`.
3. Purge any old version of the module being upgraded.
4. Load the new version of the module, making the current version the old one.
5. Purge any old version of the module, which prior to 4 was the current version.
6. Call `Mod:code_change/4`.
7. Resume the suspended processes with `sys:resume/1`, allowing them to continue handling new requests.

So far, so good, but how do we associate a module dependency with a behaviour process? Remember that in the supervisor child specification, you had to list the modules that implement the behaviour:

```
{coffee_fsm, {coffee_fsm, start_link, []}, permanent, 5000, worker, [coffee_fsm]}
```

We have to list them because this is where, during an upgrade or downgrade, *systools* tells the supervisors to suspend a particular process when upgrading one or more of its core modules. In behaviours such as event handlers and other special processes where the modules are not known at compile time, we would replace the module list with the term **dynamic** and query the process prior to an upgrade.

OTP needs to distinguish between dynamic and static module sets for scalability reasons. There is no point in asking millions of behaviours what modules they are running every time we do a software upgrade, only to discover it is not the one being upgraded. Processes with dynamic modules are few and far apart, and rarely have impact on performance when doing an upgrade. If you have dynamic children where you know millions of instances will co-exist concurrently and the modules are not known at compile time, pick an upgrade strategy which scales or do not upgrade at all.

High-Level Instructions

Actions in our *appup* file are grouped into high-level and low-level instructions, with high-level instructions being mapped to low-level ones when the upgrade scripts are generated. For the sake of simplicity (and your sanity), you are encouraged to use high-level instructions and avoid low-level ones where possible, even if they can be mixed together. Let's look at the high-level instructions in more detail:

{update, Mod}

This instruction, and all of its variants, are used for *synchronized code replacements* where all process dependent on Mod have to be suspended before loading the new version of the module. When it is loaded and its old version is purged, the suspended processes are resumed. This is the simplest variant of a module update command, as the `code_change/3,4` behaviour callbacks are not invoked. You will want to synchronize and suspend all processes with a dependency to Mod when you want all processes to consistently display the same properties towards other processes which interface towards them. By not suspending them all prior to loading the new module, some processes might display the old behaviour whilst others the new one.

{update, Mod, supervisor}

You will want to use this high-level instruction if Mod is a supervisor callback module and you are changing the supervisor specification returned by the `init/1` callback function. Any change in the supervision tree needs to be handled using the `supervisor:start_child/2` if you are adding children. Use `supervisor:terminate_child/2` and `supervisor:delete_child/2` if you are removing them. We covered these functions in [Link to Come]. The update becomes even more complicated if you are changing the order in which you start the children because of `rest-for-one` dependencies. You will have to terminate children and restart them in the order specified in your `init/1` callback function.

```
{update, Mod, {advanced,Extra}}
```

```
{update, Mod, DepMods}
```

```
{update, Mod, {advanced,Extra}, DepMods}
```

If we include the `{advanced,Extra}` tuple, the upgrade scripts invokes the `Mod:code_change/3,4` callback function, passing `Extra` as the last argument. You will need this option when the upgrade requires a change of your behaviour state and loop data. For this and all other `update` instructions, you can omit `{advanced,Extra}` or replace it with `soft`, both of which result in `code_change` not being called. `DepMods` is a module list on which `Mod` depends. Behaviours using these modules will also be suspended.

```
{update, Mod, {advanced,Extra}, PrePurge, PostPurge, DepMods}
```

`PrePurge` and `PostPurge` are by default set to `brutal_purge`. Use this option when you want processes running the old version of `Mod` to be unconditionally terminated before the updated module is loaded and after the module upgrade when the release is made permanent. You can override this behaviour by setting `PrePurge` to `soft_purge`. If some processes are still running a version of the old code, `release_handler:install_release/1`, which triggered the execution of the `relup` file, returns `{error,{old_processes,Mod}}`. If `PostPurge` is set to `soft_purge`, the release handler will purge `Mod` only after the processes executing the old version have terminated their calls.

```
{update, Mod, Timeout, {advanced,Extra}, PrePurge, PostPurge, DepMods}
```

Remember that behaviours are implemented as callback functions, so for a purge to fail, they must be executing in a callback for an unusually long amount of time or have an unusually long message queue. The default timeout value when trying to suspend a process is 5 seconds, but this can be overridden by setting `Timeout` filed to an integer in milliseconds or the atom `infinity`. If a behaviour does not respond to the `sys:suspend/1` call and the timeout is triggered, the process is ignored. It might later be terminated if the module it is executing is purged, or as the result of a runtime error when it starts running the new version of the module without properly going through the upgrade procedure. Use the `Timeout` option when, after testing your upgrades under heavy load, you see there is a need to increase the value.

```
{update, Mod, ModType, Timeout, {advanced,Extra}, PrePurge, PostPurge, DepMods}
```

By default, the `code_change/3,4` callback function is executed after loading the new module. In the case of a downgrade, `code_change/3,4` is called before loading the module. You can override this by setting `ModType` to `static`, which loads the module and calls `code_change/3,4` before an upgrade or downgrade. If not specified, or if you want the default behaviour, set `ModType` to `dynamic`,

```
{load_module, Mod}  
{load_module, Mod, DepMods}  
{load_module, Mod, PrePurge, PostPurge, DepMods}
```

You want to use this low level instruction for upgrades where you do not need to suspend the process. We refer to these upgrades as *simple code replacements*. The same applies to the instructions used for adding and deleting modules. `DepMods` lists all of the modules that should be loaded before `Mod`. This argument is an empty list by default. `PrePurge` and `PostPurge` can be set to `soft_purge` or `brutal_purge` (the default). They work the same way as they do with the update command. Use this instruction when dealing with library modules or extending functionality that does not affect running processes.

```
{add_module, Mod}  
{delete_module, Mod}
```

These commands translate to low-level instructions that add and delete modules between releases.

```
{add_application, Application}  
{add_application, Application, Type}
```

This instruction will add a new application to a release, including loading all of the modules defined in the `app` file and where applicable, starting the supervision tree. Application types covered in [Link to Come] default to `permanent`, but `Type` can also be set to `transient`, `temporary`, `load`, or `none`.

```
{remove_application, Application}  
{restart_application, Application}
```

You will want to use these commands when removing or restarting an application. Removing an application shuts down the supervision tree, deletes the modules from memory, and stops the application. If the upgrade or downgrade requires an application restart, this high-level command will translate to commands that stop and start the application and its supervision tree. You usually find application restarts in `appup` files belonging to non core OTP applications such as tools and libraries which can be restarted without affecting traffic in the live system.

You can mix high and low level instructions in the same `appup` file, but for the vast majority of use cases, high level instructions will be enough as most of your actions can be completed with them. We'll cover low level instructions in the next section, as soon as we've done our first upgrade.

Release Upgrade Files

Now that we have our `coffee.appup` file and understand what the high-level instructions do, let's use this knowledge to generate an upgrade package. The first step is to create the boot file using `systools:make_script/2`. It is not used for the upgrade itself, but is part of the package we deploy in case the upgraded node has to be rebooted (for whatever

reason) after the upgrade. In the second shell command, we create a release upgrade file called *relup*, which is placed in the current working directory. This file is generated using the emulator and application versions specified in the *rel* and *appup* file, using them to retrieve and map high and low-level instructions in the *appup* files to a sequence of low level ones.

```
Eshell V6.1 (abort with ^G)
1> systools:make_script("coffee-1.1", [{path, ["coffee-1.1/ebin"]}]). 
ok
2> systools:make_relup("coffee-1.1", ["coffee-1.0"], ["coffee-1.0"], [{path, ["coffee*/ebin"]}]). 
ok
3> systools:make_tar("coffee-1.1", [{path, ["coffee-1.1/ebin"]}], {outdir, "ernie/releases"}]). 
ok
```

In our third shell command, we create the tar file *coffee-1.1.tar.gz*. It contains the lib and releases directory specified in *coffee-1.1.rel*. Calling *make_tar*/2 picks up the *relup*, *start.boot*, and *sys.config* files automatically and creates a version 1.1 directory under releases. Note that, unlike our first installation, we did not include the *erts* option. We are going to use the one already installed.

Let's look at the *relup* file more closely now that the low level instructions have been generated. We'll explain them all in “[Low-level instructions](#)” on page 330, but even without having covered them, you should get a good idea of what is going on.

```
{"1.1",
[{"1.0", [],
 [{load_object_code,{coffee,"1.1",[coffee_fsm]}},
  point_of_no_return,
  {suspend,[coffee_fsm]},
  {load,{coffee_fsm,brutal_purge,brutal_purge}},
  {code_change,up,[{coffee_fsm,{}]}},
  {resume,[coffee_fsm]}]],
[{"1.0", [],
 [{load_object_code,{coffee,"1.0",[coffee_fsm]}},
  point_of_no_return,
  {suspend,[coffee_fsm]},
  {code_change,down,[{coffee_fsm,{}]}],
  {load,{coffee_fsm,brutal_purge,brutal_purge}},
  {resume,[coffee_fsm]}]]}].
```

Before covering the low level commands in more detail, let's look at the *systools:make_relup/3,4* call we used to generate the file itself:

```
systools:make_relup(RelName, UpFromList, DownToList, [Options]) -> ok | error
{ok, Relup, Mod, Warnings}
```

The call takes *RelName*, the name of a release we are upgrading or downgrading to. This points to the *RelName.rel* file, used to determine the version of the Erlang runtime system and the versions of the various applications. *RelName* can also be a tuple {Re

`{Name, Descr}`, where `Descr` is a term that is included in the upgrade and downgrade instructions, returned by the function installing the release on the target machine.

The second and third arguments, `UpFromList` and `DownList`, include the list of releases we want to upgrade from or downgrade to, respectively. They are all names that point to a specific version of a `rel` file used to figure which applications need to be added, removed, or upgraded. Using their respective `app` and `appup` files, the call also determines the sequence of commands that needs to be executed. The fourth, optional, argument is a list of options that may include:

`{path,DirList}`

Adds paths listed in `DirList` to the code search path. You can include wild cards in your path, so the asterisk in `"lib/*/ebin"` will expand to contain all of the sub-directories in `lib` containing an `ebin` directory. The code search path of the node creating the `relup` file must have paths to the old and the new versions of the `rel` and `app` files, as well as a path to the new `appup` and `beam` files.

`{outdir, Dir}`

Puts the `relup` file in `Dir` instead of the current working directory.

`restart_emulator`

Generates low-level instructions that reboot the node after an upgrade or downgrade.

`silent`

Returns a tuple of the format `{ok, Relup, Module, Warnings}` or `{error, Module, Error}` instead of printing results to I/O. Use this option when calling `systools` functions from scripts or integrating the call in your build process where you need to handle errors.

`noexec`

Returns the same values as the `silent` option, but without generating a `relup` file.

`warnings_as_errors`

Treats warnings as errors, and refuses to generate the `relup` script if warnings occur.

The format of the `relup` file itself is similar to the `appup` file:

```
{Vsn,  
 [{UpFromVsn1, Descr InstructionsU1}, ...,  
 {UpFromVsnK, Descr InstructionsUK}],  
 [{DownToVsn1, Descr InstructionsD1}, ...,  
 {DownToVsnK, Descr InstructionsDK}]}.
```

The `Descr` term contains a term passed in the `{RelName, Descr}` tuple of the `systools:make_relup/3,4` call. If `Descr` was omitted from the call, it defaults to an empty list. You will notice this in our example, as we left it out for the coffee machine `relup` example. `Descr` becomes relevant when automating the installation of the upgrade on

the target machine, as its values can be used by the programs or scripts installing the upgrade.

Low-level instructions

Relup files consist of low-level instruction sets generated from the *appup* files. For complex upgrades, you can write your files using low-level instructions or edited generated ones by hand. Low level instructions consist of the following:

`{load_object_code, {Application, VsN, ModuleList}}`

Reads all the modules from the `Application` ebin directory, but does not load them into the runtime system. This instruction is executed prior to suspending the behaviours and special processes. This differs from the high level instruction `load` which not only loads the module, but also makes it available to the runtime.

`point_of_no_return`

This instruction should appear once in the relup script and should be placed where the system cannot recover after failing to execute one or more of the instructions in the relup file. Crashes occurring after this instruction will result in the old version of the system being restarted. It is usually placed after the `load_object_code` instructions.

`{load, {Module, PrePurge, PostPurge}}`

Makes a module that has been loaded using `load_object_code` the current version. `PrePurge` and `PostPurge` can be set to `soft_purge` or `brutal_purge` (the default).

`{apply, {Mod, Func, ArgList}}`

Calls `apply(Mod, Func, ArgList)`. If the apply is executed before the point of no return and fails, returns (or throws) `{error,Error}`, `release_handler:install_release/1` returns `{error,['EXIT',Reason]}` or `{error,Error}` respectively. If executed after the point of no return and fails, the system is restarted with the old version of the release. This instruction could be used instead of the `code_change/3,4` callback function.

`{remove, {Module, PrePurge, PostPurge}}`

Used together with `load` and `purge`. This instruction makes the current version of `Module` old.

`{purge, ModuleList}`

Purges the old version of all modules in `ModuleList`. Behaviours and special processes executing the old version of the code being purged are terminated.

`{suspend, [Module | {Module, Timeout}]}`

Suspends behaviours that depend on the `Module` list. `Timeout` is an integer in milliseconds or the atoms `default` (set to 5 seconds) or `infinity`. If the `sys:suspend/1` call does not reply within `Timeout`, the process is ignored, but not terminated.

```
{resume, ModuleList}
```

Resumes suspended processes that depend on modules listed in `ModuleList`.

```
{code_change, [{Module, Extra}]}},
```

```
{code_change, Mode, [{Module, Extra}]}},
```

Triggers the `Module:code_change/3,4` call, passing `Extra` in all behaviour processes running `Module`. `Mode` is up or down, defining the call as either an upgrade or a downgrade. If omitted, `Mode` defaults to up.

```
{stop, ModuleList}
```

This instruction results in the `supervisor:terminate_child/2` call for all behaviours with a dependency for one of the modules specified in `ModuleList`.

```
{start, ModuleList}
```

Starts all stopped processes with a dependency to a module in `ModuleList` by calling `supervisor:restart_child/2`.

```
restart_new_emulator
```

Whenever upgrading the *emulator* or the *kernel*, *stdlib* and *sasl* core applications, the emulator needs to be restarted right after upgrading these applications, but before executing the remainder of the *relup* file. All other applications will be restarted with their old versions running in the new emulator and upgraded when running the remainder of the *relup* file in the new emulator. When different processes end up running different application versions in this manner, non backward compatibility clashes between them can occur, so ensure all possible scenarios in your upgrade procedure have been properly tested before using this technique. If you are worried about the order of your low-level instructions, use high-level ones and let `systools:make_relup/3,4` generate the *relup* file. This instruction should be executed only once during the upgrade.

```
restart_emulator
```

This instruction is used when an emulator restart is required as part of an upgrade that does not involve the core applications or an emulator upgrade. This instruction may appear only once in the *relup* file and has to be the last instruction.

Installing an Upgrade

Let's go back to the *coffee-1.1.tar.gz* file we generated and use it for our live upgrade. We assume that it has been placed in the *releases* directory of the target environment. From the *ernie* root directory, we connect to the *coffee_fsm* node that we left running version 1.0. If it is not running, start it with **bin/start**. We unpack the new release using the `release_handler:unpack_release/1` call, uncompressing all the files, adding the version of *coffee-1.1* application to the lib directory, and creating the version 1.1 in the releases directory. We can see in shell command 2 and 3 that after unpacking the new release, it resides alongside 1.0, and that 1.0 is still running.

```

$ bin/to_erl /tmp/
Attaching to /tmp/erlang.pipe.1 (^D to exit)

1> release_handler:unpack_release("coffee-1.1").
{ok, "1.1"}
2> release_handler:which_releases().
[{"coffee", "1.1",
  ["kernel-3.0.1", "stdlib-2.1", "sasl-2.4", "coffee-1.1"],
  unpacked},
 {"coffee", "1.0",
  ["kernel-3.0.1", "stdlib-2.1", "sasl-2.4", "coffee-1.0"],
  permanent}]
3> application:which_applications().
[{coffee, "Coffee Machine Controller", "1.0"},
 {sasl, "SASL CXC 138 11", "2.4"},
 {stdlib, "ERTS CXC 138 10", "2.1"},
 {kernel, "ERTS CXC 138 10", "3.0.1"}]
4> coffee_fsm:espresso().
Display:Please pay:150
ok
5> coffee_fsm:pay(100).
Display:Please pay:50
ok
6> release_handler:install_release("1.1").
{ok, "1.0", []}
7> coffee_fsm:cancel().
Display:Make Your Selection
ok
Machine:Returned 100 in change
8> coffee_fsm:open().
ok
Display:Open
9> coffee_fsm:close().
Machine:Rebooted Hardware
Display:Make Your Selection
ok
10> application:which_applications().
[{coffee, "Coffee Machine Controller", "1.1"},
 {sasl, "SASL CXC 138 11", "2.4"},
 {stdlib, "ERTS CXC 138 10", "2.1"},
 {kernel, "ERTS CXC 138 10", "3.0.1"}]
11> init:restart().
ok
12>
Erlang/OTP 17 [erts-6.1] [source-d2a4c20] [smp:4:4] [async-threads:10] [kernel-poll:false]
...<snip>...

Eshell V6.1 (abort with ^G)
1> application:which_applications().
[{coffee, "Coffee Machine Controller", "1.0"},
 {sasl, "SASL CXC 138 11", "2.4"},


```

```
{stdlib,"ERTS  CXC 138 10","2.1"},  
{kernel,"ERTS  CXC 138 10","3.0.1"}]
```

We upgrade the release by executing the `release_handler:install_release/1` call. If issues arise and a restart is triggered, the system will reboot and revert to the old version. If the system is stable, the current (new) version is made permanent by calling `release_handler:makePermanent/1`.

We then use the new client functions we've added to test the transition to and from state *service* before rebooting the node in shell command 11. Because we never made the release permanent, the node restarts version 1.0.

Next, in shell commands 2 and 3, we reinstall the release and make it permanent. At this point, we do not need files specific to 1.0 anymore. Unused releases can be removed from the system using the `release_handler:remove_release/1` call. The call removes the applications that are only part of that release the lib directory, removes the directory from *releases*, and updates the *RELEASES* file there. To revert back to the old version we have to reinstall it, covering all the steps we've just described, including creating *appup* files for the coffee application version 1.0, a relup file and a tar file.

```
2> release_handler:install_release("1.1").  
{ok,"1.0",[]}  
3> release_handler:makePermanent("1.1").  
ok  
4> release_handler:remove_release("1.0").  
ok  
5> release_handler:which_releases().  
[{"coffee","1.1",  
 ["kernel-3.0.1","stdlib-2.1","sasl-2.4","coffee-1.1"],  
 permanent}]  
6> halt().  
[End]  
$ ls lib/  
coffee-1.1      kernel-3.0.1      sasl-2.4          stdlib-2.1
```

That's it! A software upgrade during runtime, with the ability to fall back to old releases when issues occur or remove them when they are no longer needed.



The release handler is intended to work with embedded target systems. If you use it with simple target systems, you need to ensure the correct boot and config files are used in the case of a restart. How you do it is entirely up to you. You could replace existing files or have OS environment variables pointing to the correct ones.

The Release Handler

We introduced the SASL application in [Link to Come]. It is one of the core OTP applications that has to be part of every release because it contains tools required to build,

install and upgrade the release itself. If you looked at SASL's supervision tree ([Link to Come]), you might have noticed the release handler process. It is responsible for unpacking, installing and upgrading releases locally on each node. It also removes them and makes them permanent. We used the release handler and went through these phases in our example. [Figure_toCome]

The release handler assumes a release tar file, created using `systools:make_tar/1,2` and placed in the `releases` directory. Each release version can be in one of the following states: *unpacked*, *current*, *permanent* and *old*. State transitions occur when functions in the `release_handler` module are called or a release that has not been made permanent fails, triggering a system restart. At any one time, there is always a release that is either current or permanent. Let's look at the functions exported by the `release_handler` module, including those that trigger the transition more closely.

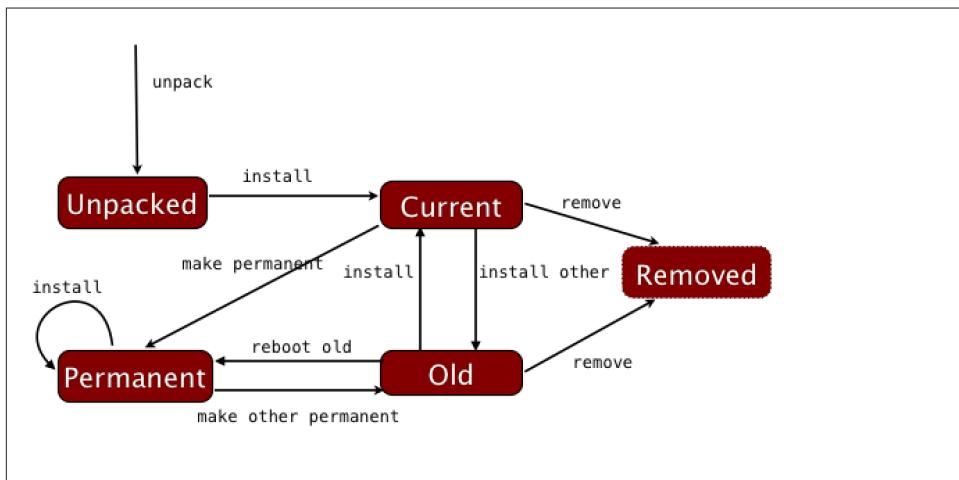


Figure 12-4. Upgrading a Release

When dealing with your first target installation, the release handler becomes relevant only if Erlang is already installed on the target machine. As it wasn't when we created the first `coffee_fsm` release, everything had to be done manually. If you follow the steps, you will notice that the first call we did once version 1.0 of the system was up and running was to create the `RELEASES` file.

```
release_handler:create_RELEASES(Root, RelDir, RelFile, AppDirs) -> ok | {error, Reason}
```

This call creates the first version of the `RELEASES` file, stored in the `releases` directory. It contains the persistent state of the release handler, which includes the release applications, their versions and absolute paths. The Erlang VM executing this function must have permission to write to the `releases` directory. `Root` is the Erlang root directory, whilst `RelDir` is the path pointing to the `releases` directory. The `releases` directory is

often located in the Erlang root directory, but you can over-ride this by setting the OS or OTP environment variables described in [Link to Come]. `RelFile` points to the release file located in the `releases` directory whilst `AppDirs` is a list of `{App, Vsn, Dir}` tuples used to override the applications stored in `lib`. It is most commonly used when distributing Erlang in OS specific packages and not OTP ones.

```
release_handler:unpack_release(Name) -> {ok, Vsn} | {error, Reason}
```

This function unpacks the `Name.tar.gz` file located in the `releases` directory. It checks that all mandatory files and directories are present, adding the applications in the `lib` and `release` directory under `releases`. It fails if the string `Name` is an existing release, or if there are issues unpacking or reading the mandatory files and directories.

```
release_handler:install_release(Vsn)
release_handler:install_release(Vsn, OptList) -> {ok, OtherVsn, Descr} | {error, Reason}
                                                {continue_after_restart, OtherVsn, Descr}
```

When we have unpacked the release, `install_release/1,2` triggers the software upgrade (or downgrade) executing the instructions specified in the `relup` file. `OptList` is a list of options which allow us to override some of the default settings. They include

- `{error_action, restart | reboot}` to specify if the runtime system is rebooted (`init:reboot()`) or restarted (`init:restart()`) as the result of an upgrade failure.
- `{suspend_timeout, Timeout}` to override the default (five second) timeout for the `sys:suspend/1` call, used to suspended a process prior to upgrading the code.
- `{code_change_timeout, Timeout}` to override the default (five second) timeout for the `sys:change_code/4` call, used to tell a suspended process to upgrade the code.
- `{update_paths, Bool}` is used when overriding the default `lib/App-Vsn` directory provided in the `AppDirs` argument in the `create_RELEASES/4` call. Setting `Bool` to `true` will cause all code paths of the applications in `AppDirs` to be changed, including applications which are not being upgraded. Setting it to its default value of `false` will only cause the paths of the upgraded applications to be changed.

You might recall that the `relup` file contains tuples of the format `{Vsn, Descr, Instructions}`. `Descr` is part of the return value when the upgrade or downgrade was successful. If `{continue_after_restart, OtherVsn, Descr}` is returned, the runtime system and the core applications are being upgraded, requiring an emulator restart before the remainder of the script is executed.

If errors we can recover from have occurred, `{error, Reason}` is returned. Recoverable errors include `Vsn` already being the permanent release, or the `relup` file missing alongside others which will result in the installation of the release failing, but not requiring a node restart. If the upgrade fails due to an unrecoverable error, the node is restarted or rebooted.

```
release_handler:check_install_release(Vsn)
release_handler:check_install_release(Vsn,Options) -> ok | {error, Reason}
```

Installing releases and upgrading code can be a risky and time consuming operation. This function mitigates risks of issues happening, checking if `Vsn` can be installed, ensuring that all mandatory files are available and accessible, as well as evaluating all low level instructions in the `relup` file prior to the `point_of_no_return`. `Options` is a list containing `[purge]`, which soft purges the code when doing the checks. This will speed up the installation of the release itself, as all modules are soft purged prior to the upgrade itself.

```
release_handler:make_permanent(Vsn) -> ok | {error, Reason}
```

When we have installed a new release and executed the instructions in the `relup` file, we keep the nodes under observation, possibly running diagnostic tests. If there are issues, restarting the node will use the old boot file and cause a restart of the old version. By calling `make_permanent/1`, the boot scripts which points to the upgraded release becomes the one used when rebooting or restarting the node. This call can fail for a variety of reasons, including `Vsn` not being the current version or not being a release at all.

```
release_handler:remove_release(Vsn) -> ok | {error, Reason}
```

If a release has been made permanent, files specific to old releases can be removed. This call will delete old applications no longer in use, the `Vsn` directory containing the `rel`, `boot` and `sys.config` files in the `releases/Vsn` directory. This call also upgrades the available releases in the `RELEASES` file. It fails if `Vsn` is permanent or non-existing release.

```
release_handler:reboot_old_release(Vsn) -> ok | {error, Reason}
```

Houston, we have had a problem. If your current release is not operating as expected and you need to revert to an old release (which you have not removed), this call reboots the runtime system with the old boot file, making it the new, permanent version.

```
release_handler:which_releases(Status)
release_handler:which_releases() -> [{Name, Vsn, Apps, Status}]
```

This call uses the `RELEASES` file and returns all the releases known to the release handler. `Status` is one of `unpacked`, `current`, `permanent` or `old`.

The `release_handler` module exports functions which make it possible to upgrade and downgrade single applications, creating a release upgrade script on the fly and evaluating it. These functions (which we are not covering in this book) are meant to facilitate and automate testing of application upgrades. They should not be used in production systems, as the changes are not persistent in the case of a system restarts.

It is possible to install upgrades without the release handler whilst keeping its view consistent and up to date. This functionality comes in handy when dealing with OS specific packages, when you do deployments and upgrades with other tools, or even write your own. There are functions which allow us to inform the release handler process

of the addition and removal of releases and release specific files. You can read about these functions as well as the ability to up -and- downgrade single applications in the `release_handler` manual pages which come with the standard Erlang distribution.

Upgrading Environment Variables

When upgrading your release, the mandatory `sys.config` file is picked up when creating the package. The package will also include a new `app` file for every new and upgraded application. These files might contain new or updated application environment variables, and if the files are no longer needed, they will have been omitted all together. During the upgrade, the application controller will compare old environment variables with their current counterparts in the start scripts (set with the `-application key value` flag), `config`, `app` files, deleting or updating them accordingly. When done, the following callback function is called in the new application callback module, prior to resuming the processes:

```
Module:config_change(Updated, New, Deleted)
```

`Updated`, `New`, and `Deleted` are lists of `{Key, Value}` tuples, where each key is an environment variable and the value is what you want the variable set to. This is an optional callback that can be omitted, but is useful when process states depend on environment variables read at startup.

Making a release permanent will change the `sys.config` file pointed to by the start scripts to the new version. It is done only now, because rebooting a node with a release that is not permanent reverts back to the previous release.

Upgrading Special Processes

Upgrading special processes is no different from upgrading behaviours. If you are doing a simple code replacement, load the new module through the `add_module` instruction. If the upgrade has to be a synchronized code replacement, use the same update high level instruction you would use for OTP behaviours. Upon receiving a message of the format `{system, From, Msg}`, the special process invokes `proc_lib:handle_system_msg/6`, which suspends the process. (We covered system messages in [Link to Come].) If the update command had the `{advanced, Extra}` parameter in its `Change` field, the following callback function is called in the special process callback module.

```
Mod:system_code_change(LoopData, Module, Vsn, Extra) -> {ok, NewLoopData}
```

This call returns the tuple `{ok, NewLoopData}`. `Module` is the name of the callback module, `Vsn` is either the version you are upgrading to, or in the case of a downgrade, is `{downgrade, Vsn}`, `Vsn` being a string in both cases.

One final note. Remember the system message `{get_modules, From}` that special processes have to handle when they are not aware of their dependent modules? The ones

where we use the `dynamic` atom in the supervisor specification, covered in [Link to come]? When upgrading, all processes whose child specifications in the supervisor have module dependencies set to `dynamic` reply to such a message with `From!{modules, ModuleList}`, containing the list of modules on which the special process currently depends. This will inform the release handler coordinating synchronized upgrade if this special process is part of a dependency chain and should be suspended during the upgrade of a particular module.

Upgrading In Distributed Environments

Synchronized software upgrades in distributed environments? Is that possible? Are we crazy enough to try it? If you have a small cluster, trust your network, and have dependencies connected to your upgrade across your nodes, then why not? Remember that distributed Erlang was originally intended for clusters that ran behind firewalls in the same data center, and more often than not, also in the same subrack. If you were upgrading a switch, distributed Erlang often ran on the same backplane the switch was controlling, so if you lost your network, there was nothing to control because you also lost your switch.

In a small cluster with a few nodes running in the same subrack, you have little to worry. For larger clusters, clusters across data centers, or where networks are unreliable, devise a strategy to upgrade a node without the need to synchronize.

Enough warnings. Let's drink some Red Bull and get on with it. If you include the `sync_nodes` low level instruction in your `appup` file, the `relup` script that gets generated will synchronize with the other nodes also waiting to be upgraded and upgrade them also when they are also attempting to synchronize.

Synchronization is triggered by the following instruction:

```
{sync_nodes, Id, NodeList} {sync_nodes, Id, {Mod, Func, ArgList}}
```

This instruction invokes `apply(Mod, Func, ArgList)` to get the list of nodes that recognize `Id`, which are the nodes to synchronize. `Id` can be any valid Erlang term. You can also hard-code `NodeList` in the `appup` file. For the synchronization to be successful, remote nodes must be executing the same instruction with the same `Id`.

If you lose connectivity toward a remote node with which you are attempting to synchronize, either because of a network partition or because the remote node crashed, the node is restarted with the old release. There is no timeout, so if a remote node is not being upgraded or out of sync, the local node attempting to upgrade will hang until all remote nodes have executed `sync_nodes` or connectivity towards one of the nodes is lost. This is why the technique in this section has some risks for nodes distributed across a wide-area network.

If you have not synchronized your upgrades properly, your cluster will hang waiting for all other nodes. And if there are issues with your network connectivity or the upgrade in one of the other nodes fails, you will trigger a series of node restarts that will hopefully recover and continue running the old release. But in the worst case, this technique might cause a cascading failure where you knock out a node after another when they fail to cope with the restart. You have been warned! Use synchronized distributed upgrades only when it is safe and the use case motivates it.

Upgrading The Emulator and Core Applications

You upgrade the emulator and the core applications by providing their new versions in the new release *rel* file. The rest is taken care for you when generating the *relup* file. Just remember to include the `erts` option in the `systools:make_tar/2` call when upgrading the Erlang runtime system, as it will include the emulator in the new tar file. If you think it sounds simple, it is, but there are a few catches you need to be aware of.

Upgrading the emulator and core applications (ERTS, Kernel, STDLIB, and SASL) requires a restart of the virtual machine, usually triggered by the `restart_new_emulator` instruction. Unlike other upgrades, this will be the first instruction executed in the file, starting the new emulator and the new core applications, together with the old versions of the remaining applications. This two-phase approach allows the remaining behaviours and special processes being upgraded to call `code_change` as part of their upgrade, using new versions of the core applications while doing so.

If you are not happy with this approach, you can edit the *relup* file by hand. Replacing `restart_new_emulator` with the `restart_emulator` instruction will restart the emulator with the new versions for all applications. A restart of the emulator (which is not the new emulator) is the last instruction you should be executing in your *relup* file, as all it does is restart the system with the new boot file. This means that any instructions that follow `restart_emulator` are ignored, whilst any instructions before it are executed with the old emulator. An helpful instruction you have to add manually is `apply`, which you could use instead of `code_change` if opting to start the new versions of the applications directly.

Non-Backward Compatible Upgrades and Downgrades

There are times when, as a result of the `restart_new_emulator` instruction, you restart old applications that you plan on subsequently upgrading with the new core applications. If the upgrade spans several releases, you might run the risk of your non-core applications calling deprecated functions in the core applications that have since been removed. Deprecated functions are kept for two major releases, with warnings printed out when you compile the code that uses them, after which you can safely remove the functions. The solution is to replace any deprecated functions as soon as possible, and

upgrade in several steps while testing the upgrades to ensure that all applications are forward compatible.

If you are still running an emulator version older than R15 (and we know many of you are), you might run into problems when downgrading, as an attempt to load the new versions of the beam files will be made after restarting the old emulator. If you are affected, compile your new code with the old emulator and its corresponding version of the compiler.

In both of these edge cases, testing upgrades and downgrades is critical and will at a very early stage highlight any potential issues and incompatibilities.

Summing Up

As with most things we've seen in this book, Erlang provides powerful basic language constructs which OTP uses to build libraries and frameworks that hide complexity, simplifying the development, deployment and maintenance of Erlang based systems. Starting with `code:load_file/1`, which handles the loading of a module in your runtime system, we looked at how to manage state changes in processes, database schema changes together with synchronization of processes and their dependencies, and dependencies in distributed environments.

In order to upgrade a target system, you need to start with a baseline installation. It will usually be the first release, the one we created manually. Unless you were using *relx* and *rebar*, it has to be a manual task, because most of the release upgrade tools are written in Erlang and will not run without your baseline system. It's a classic chicken-and-egg problem.

With the baseline release in place, you need to follow these steps to successfully upgrade your system. Don't panic, a lot of these steps are either automated, handled by existing tools, or both:

- Add the new functionality, package it into their respective modules and applications, and bump up the module and application versions respectively.
- Create the new *rel* file containing new and upgraded applications whilst omitting the deleted ones.
- Generate your start scripts and new *sys.config* file, ensuring you can boot the new release on its own.
- If any of your behaviours or special processes require a state change or use a different data format (including database schema changes) as part of the upgrade, migrate your state and data formal from the old version to the new one and back in your `code_change` functions.

- Write an *appup* file for each application you are upgrading. Place these files in the *ebin* directory.
- Create a *relup* file containing all the low-level instructions executed during the upgrade.
- Create a package that you can deploy in the *releases* directory of the installation you are upgrading.
- Unpack the release and install it.
- If stable, make your new release permanent. If unstable, reboot the node restarting the old release.

Once the release is unpacked, a number of transitions can take place on the node being upgraded. When you install a release and the upgrade is successful, the system starts running the new version. If the upgrade fails for any reason, the system is rebooted and reverts to the previous version. When running the new version, it can be made permanent. When this happens, any subsequent node restart will restart the latest version (Figure 12-5).

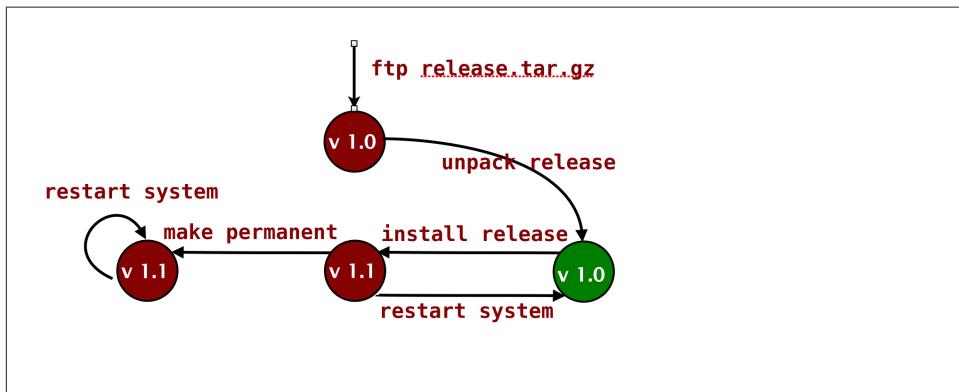


Figure 12-5. Upgrading a Release

We also covered upgrades in distributed environments, which allow you to synchronize the nodes. This happens in the real world, but only for very small clusters where the network is reliable. If you are dealing with distributed data centers, cloud computing, or virtualization, as well as lots of other layers of complexity and instability, you need to take a different approach to upgrades. Make sure that old and new nodes are backward-compatible and interoperable with each other, allowing them to co-exist in the same cluster. Upgrade a few nodes, monitor them to ensure all is well, and keep on upgrading. If you lose a few machines or get a network partition or upgrade failures, keep on trying until all nodes have been successfully upgraded.

Let's take this argument a step further. For clusters where you have no single point of failure with multiple instances of the nodes running, do live upgrades really make sense? If you are able to cleanly shut down nodes without losing any requests and stopping traffic, isn't it easier to shut down one node at a time, upgrade its code, then restart it to bring it back into the cluster? You would be able to upgrade your code without showing the embarrassing *Our system is down for maintenance, bear with us, we are doing this because your business is important and we value you as a customer* screen most online banks show us a little too frequently, and ensuring that you do not lose any requests as a result of the upgrade?

How you do your upgrades depends entirely on the size of your cluster, the infrastructure you have in place to control it, your redundant capacity, and the experience and size of your team. Software upgrades take time and money to implement, test, and deploy. And if things go wrong, most of the time, they will go wrong during an upgrade. If you are a startup that does not have to provide 99.999% availability, no one will care whether you bounce your nodes every now and then. If you are upgrading tens of thousands of switches, however, where each switch handles traffic for millions of subscribers with contractual penalties for downtime and outages, or an ecommerce site generating thousands of dollars in revenue every minute, users will care!

Software upgrades are a unique and powerful feature you can use in rare, but critical, moments. Use them where the extra effort makes sense, ensuring you test your upgrades and downgrades under heavy load, covering as many failure scenarios as possible.

If this chapter is not enough, the user guides and reference manuals, along with the module documentation that comes with the standard Erlang distribution, contain scattered but detailed information on release upgrades. You should start with the section on *Creating and Upgrading a Target System* in the *System Principles User's guide*. Tools are covered in the module documentation for *systools* and *release_handler*. Finally, *relup* and *appup* files both have manual pages that describe the formats of the files, including all the instructions they may contain. Don't miss the *Appup Cookbook* chapter, in the *OTP Design Principles User's Guide*. The same guide also contains descriptions of the *code_change* functions in the respective sections for every behaviour and special process.

What's Next?

With the knowledge provided in this chapter on how to package releases and perform live upgrades without affecting traffic, time has come to look at how to architect a system. If you want a system with five nines availability, what basic functionality should all of your nodes in production have? What distributed architectural patterns should you be applying to get your nodes to scale. In the next chapter, we will look at what it takes. So what are you waiting for? Turn the page and read on!