CubedOS Operating System

Last Generated: July 26, 2024

Contents

Legal

1 Introduction

This document describes the design, implementation, and usage of the CubedOS application framework. CubedOS was developed in Vermont State University's CubeSat Laboratory with the purpose of providing a robust software platform for CubeSat missions and of easing the development of CubeSat flight software. In many respects the goals and architecture of CubedOS are similar to those of the Core Flight System (CFS) written by NASA Goddard Space Flight Center [2]. However, unlike CFS, CubedOS is written in SPARK with critical sections verified to be free of the possibility of runtime error. We compare CubedOS and CFS in more detail in Section 1.1.

If you are an application developer who is new to CubedOS, you might want to start with the CubedOS Tutorial in Section 3.1. This tutorial will guide you through the process of creating a simple CubedOS application and orient you to the overall design and usage of the system.

Although CubedOS was developed to support the CubeSat missions at Vermont State University, the intent is for CubedOS to be general enough and modular enough for other groups to profitably employ the system. Since every mission uses different hardware and has different software requirements, CubedOS is designed as a framework into which *modules* can be plugged to implement whatever mission functionality is required. CubedOS provides inter-module communication and other common services required by many missions. CubedOS thus serves both as a kind of operating environment and as a library of useful tools and functions.

It is our intention that all CubedOS modules also be written in SPARK and at a minimum proved free of runtime error. However, CubedOS also allows modules, or parts of modules, to be written in full Ada or C if desired. This allows CubedOS to take advantage of third party C libraries or to integrate with an existing C code base. It is an area for future work to create a bridge between CubedOS and CFS so that current CFS users can experiment with CubedOS and eventually migrate mission critical components to CubedOS.

CubedOS can run either directly on top of the hardware, or on top of an operating system such as Linux or VxWorks, all with the assistance of the Ada runtime system. In particular, CubedOS makes use of Ada tasking without directly invoking the underlying system's support for threads. This simplifies the implementation of CubedOS while improving its portability. However, it does require that an Ada runtime system be available for all envisioned targets. In effect, the Ada runtime system is CubedOS's operating system abstraction layer (borrowing a term from the CFS community).

For resources that are not accessible through the Ada runtime system, CubedOS driver modules can be written that interact with the underlying operating system or hardware more directly. Although these modules would not be widely portable, they could, in some cases, be written to provide a kind of hardware abstraction layer (HAL) with a portable interface. Although we have not yet attempted to standardize a HAL interface, this is also an area for future work.

The architecture of CubedOS is a collection of modules, each containing one or more Ada tasks. Although CubedOS is written in SPARK there need not be a one-to-one correspondence between CubedOS modules and SPARK packages. In fact, modules are routinely written as a collection of SPARK packages in a package hierarchy. ⟨⟨⟨ Provide a forward reference. ⟩⟩⟩ PET

Critical to the plug-and-play nature of CubedOS, each module is self-contained and does not make direct use of any code that is private to another module. However, CubedOS does allow shared library packages to be written for holding utility subprograms of various kinds. The CubedOS distribution includes a collection of such library packages that are used by CubedOS core modules, but that can also be used by application modules. ⟨⟨⟨ It is necessary for shared library packages to have certain properties to ensure "good behavior." Does pragma Pure say what is needed? ⟩⟩⟩ PET

All inter-module communication is done through the CubedOS infrastructure with no direct sharing of data. In this respect CubedOS modules are similar to processes in a conventional operating system. One consequence of this policy is that a library used by several modules must be task-safe. In this respect, library packages resemble shared libraries in a conventional operating system and have similar concerns regarding library-wide global data management.

In the language of operating systems, CubedOS can be said to have a microkernel architecture where task management is provided by the Ada runtime system. Both low level facilities, such as device drivers, and high level facilities, such as communication protocol handlers or navigation algorithms, are all implemented as CubedOS modules. All modules are treated equally by CubedOS; any layered structuring of the modules is imposed by programmer convention.

We are working on extending the architecture to allow communication to occur between modules in different operating system processes or even on different hosts. This extension will allow CubedOS applications to be distributed across multiple spacecraft while still operating as a single, integrated application. ⟨⟨⟨ Provide a forward reference. ⟩⟩⟩ PET

In addition to inter-module communication, CubedOS provides several general-purpose, core services. In some cases different implementations of a service can be exchanged for that provided by CubedOS (e.g., the file server interface might be implemented in various ways depending on the needs of each mission). Having a standard interface allows other components of CubedOS to be programmed against that interface without concern about the underlying implementation used. The following features are either available or are planned to be implemented:

- An asynchronous message passing infrastructure with, eventually, the ability to pass messages across operating system processes or even across different hosts. This, together with the underlying Ada runtime system, constitutes the kernel of CubedOS.
- A runtime library of useful packages, all verified with SPARK.
- Timing services.
- File services. This provides access to non-volatile storage via a general purpose file system abstraction that might be built on different underlying implementations.
- Event services.
- Command scripting services.
- Publish/Subscribe services. This provides the well known publish/subscribe communication model on top of CubedOS's native point-to-point message passing system.
- Log and telemetry gathering services.
- Name resolution services. This provides a mapping between abstract module names and their module ID values. It also provides a registration service for dynamic module ID assignments.

A CubedOS system also requires drivers for the various hardware components in the system. Although the specific drivers required will vary from mission to mission, CubedOS does provide a general *driver model* that allows components to communicate with drivers fairly generically. In a typical system there will be low level drivers for accessing hardware buses as well as higher level drivers for sending/receiving commands from subsystems such as the radio, the power system, the camera, etc. The low level drivers constitute the bulk of the CubedOS HAL.

1.1 Related Work

The most closely related work to CubedOS is NASA's Core Flight System [2]. Like CubedOS, CFS endeavors to be a general purpose framework for building flight software. Also like CubedOS, CFS is associated with a collection of modules that support common functionality needed by many missions. In addition, CFS makes use of a message passing discipline using a publish/subscribe model of message handling. CubedOS can provide support for publish/subscribe message handling by way of a library module.

The main difference between the systems, aside from maturity level (CFS is a long-established project with a history of actual use), is that CubedOS is written in SPARK and verified free of runtime error. In contrast, CFS is written in C with no particular static verification goals.

The CFS architecture is layered, whereas CubedOS modules operate as peers of each other. The CFS architecture makes use of a separate Operating System Abstraction Layer (OSAL) that presents a common interface across all the platforms supported by CFS. In contrast, CubedOS relies on the Ada runtime system for this purpose, and is thus Ada specific, but remains operating system independent. CubedOS also uses a module library, the CubedOS HAL, to provide hardware and OS independence in areas not covered by the Ada runtime system and standard library, but the interface to these modules is not yet standardized.

Kubos [4] is a project with roughly similar goals as CFS and CubedOS. It is not as mature as CFS. Like CFS, Kubos is written in C without static verification goals in the sense that we mean here.

Some CubeSat flight software is written directly on top of conventional embedded operating systems such as Linux, VxWorks [6], or RTEMS [5]. These systems allow application software to potentially be written with a variety of tools and methods, although C is most often used in practice. They provide flexibility by imposing few restrictions, but they also don't, by themselves, provide support for common flight software needs. Also, they are themselves not statically verified as CubedOS is, although the Wind River VxWorks Cert platform [7] does provide a means by which VxWorks can be used in safety critical avionics applications conforming to the DO-178B standard.

2 Design

This chapter describes the design of CubedOS. The design is broken into several sections. The first section describes the requirements that CubedOS is intended to meet. The second section describes the overall architecture of CubedOS. The third section provides a detailed design of the CubedOS core modules.

2.1 Requirements

In this section we detail the requirements of the various CubedOS core modules. These requirements are used to drive the architecture and design of the system. ⟨⟨⟨ There are quite a few design aspects to the description here. It might be valuable to create a better separation between requirements and design. ⟩⟩⟩ PET

2.1.1 Core Requirements

At its heart CubedOS is a message passing microkernel. Conceptually, each component of the system is stored in a *module* consisting of one or more Ada packages arranged in a package hierarchy. Each module has a *module root package* that is the parent of the module's package hierarchy. Modules in CubedOS are conceptually similar to processes in a conventional operating system or "applications" in CFS.

CubedOS uses modules for application code, high level operating system-like components, and device drivers. Applications can be constructed from multiple application level modules. The system does not distinguish between different types of modules (for example, between application specific modules and device drivers). There is no "kernel mode" of execution. The architecture we describe here is thus enforced primarily by convention. In theory, any structuring expressible using normal Ada visibility rules would be acceptable, but there are advantages to following the conventions of CubedOS, as we describe throughout this document. ⟨⟨⟨ We should consider building a tool that enforces the aspects of our convention that are most important. We should also consider writing a document that specifies all CubedOS conventions in a single, easy to reference place. ⟩⟩⟩ PET

For example, it is possible, even if not normally desirable, for an application module to directly invoke functionality in a device driver. In contrast, conventional operating systems typically do not allow this, or only allow it under very controlled circumstances.

However, such direct invocations may be more efficient; an important consideration for a constrained embedded system. This example shows that while we recommend certain design approaches when using CubedOS, we also do not want to be overly constraining. CubedOS users with special needs, should be able to work with the system rather than outside it.

CubedOS does distinguish between active modules and packages of shared library code. A module is distinguished from library code in that a module contains at least one task, whereas library packages are entirely passive. CubedOS conventions do permit library packages to be shared among multiple modules, but it is not permitted for any module to directly call subprograms or entries in another module. It is also not permitted for one module to communicate with another module via the internal state of a library package. ⟨⟨⟨ Ada has some pragmas in the distributed systems annex that we can perhaps use here to provide compiler enforcement of this restriction. ⟩⟩⟩ PET The only way functionality in another module can be invoked is by sending that module a message.

In each module one task selected by the programmer, called the module's *server task*, is distinguished. The server task is the *only* task in a module that is allowed to receive messages for that module. However, messages can be sent by any task. In many modules, the server task is the only task. The following requirements apply:

- **Core.Ada2022**. The base language for CubedOS and CubedOS applications shall be Ada 2022. Note that the Core.SPARK requirement still restrains what features from Ada 2022 can be used. However, any feature of the Ada 2022 standard that is supported by SPARK is explicitly allowed.
- **Core.SPARK**. The core system, including the core modules described in this document, shall be written in SPARK and proved free of information flow errors and execution runtime errors.
- **Core.Jorvik**. The core system, including the core modules described in this document, shall be written to conform with the restrictions of the Jorvik profile. Although the Jorvik profile puts more requirements on the runtime system than, for example, the Ravenscar profile, Jorvik also provides: relative delays, "pure" barrier expressions, and access to Ada.Calendar, to name a few things. These are features CubedOS applications can use to good effect.
- **Core.Static**. Modules shall be statically allocated. All modules the system will ever have shall exist at deployment time. There is no dynamic loading of modules. ⟨⟨⟨ What about on-the-fly software updates? Are we ruling that out, or is that outside the scope of these requirements? ⟩⟩⟩ PET
- **Core.UniqueID**. Each active module shall have a unique ID number that is either statically assigned by the application developer or dynamically assigned at system initialization time by the Name Service. ⟨⟨⟨ The Name Service is controversial. We might not want it and instead require that module ID numbers all be statically assigned. ⟩⟩⟩ PET

- **Core.IDNumbers**. Module ID numbers shall be small integers beginning at 1 and ranging up to the number of modules in the system.

2.1.2 Runtime Library

The CubedOS runtime library is a collection of packages containing general purpose components of interest to many CubedOS applications. Unlike the Ada standard library and many third party libraries, the CubedOS runtime library is written in SPARK. The following requirements apply:

- **Lib.Ada2022**. The base language for the CubedOS library shall be Ada 2022. Note that the Core.SPARK requirement still restrains what features from Ada 2022 can be used. However, any feature of the Ada 2022 standard that is supported by SPARK is explicitly allowed.
- **Lib.SPARK**. The runtime library shall be written in SPARK and proved free of information flow and execution runtime errors.
- **Lib.Tests**. Unit tests shall be provided for all library components with an emphasis on sections that may not yet be fully proved.
- **Lib.TaskSafety**. All library components shall be task safe in the sense that they can be called simultaneously from multiple tasks without error. This is needed because library packages are shared among the modules.
- **Lib.PackageDeployment**. Only the packages actually used by an application shall become part of the application's executable. This means the library can contain a rich collection of packages without burdening applications that don't require them.

The precise library components provided is not specified here but is intended to be an open-ended set of components found useful in CubedOS applications. Examples of such components include, but are not limited to:

- String processing, including regular expression processing
- Statistical processing
- Linear algebra
- Image processing
- Error checking and error correction
- Communication protocol processing
- Lightweight database support

- Security processing such as encryption/decryption
- Data compression

What distinguishes runtime library material from active CubedOS modules is that none of the runtime library components contain tasks or perform blocking operations (such as delays). Notice, however, that some library components may wish to send messages to other active components, such as the file server. In that case the library cannot directly receive the reply. Only a module can receive from a mailbox. Instead, the library code must rely on the module using the library to forward replies into the library as appropriate. This unusual organization also distinguishes the CubedOS runtime library from most other libraries. ⟨⟨⟨ Probably it would be better for library components to never send messages. Maybe we should make that another aspect of our CubedOS coding conventions. ⟩⟩⟩ PET

2.1.3 Core Modules

Any CubedOS based system shall contain certain *core modules* as a minimum. Note that it is possible that some core modules will not be required by some applications. In this section we describe these core modules.

Message Manager

The message manager is not a proper CubedOS module but rather forms the CubedOS kernel. It thus does not have a module ID number. Instead, the message manager holds mailboxes used to facilitate communication between the other active modules in the system. The following requirements apply:

- **MessageManager.Mailbox**. The message manager shall provide exactly one mailbox for each module in the system.
- **MessageManager.Access**. Mailboxes shall be accessed using the module ID number of the recipient. These ID numbers shall be globally available to all modules either as statically initialized constants or via the Name Service using abstract names as keys.
- **MessageManager.Async**. Messages shall be delivered asynchronously to a module's mailbox. It is not necessary for a module to receive a sent message immediately.
- **MessageManager.Sender**. Messages shall carry the full CubedOS address of the sender to so that replies can be returned.

- **MessageManager.Unstructured**. Messages shall be unstructured octet arrays. This allows all modules to use a common message type. However, to simplify the task of creating and decoding messages, CubedOS shall provide a standard message encoding/decoding facility. [1]
- **MessageManager.Sizes**. It shall be possible for the application developer to tune the size of the mailboxes (number of pending messages) and the size of the messages themselves.
- **MessageManager.Priority**. Messages shall have a priority assigned to them which shall be, by default, the priority of the sending module.
- **MessageManager.PrioritySet**. It shall be possible for the sender of a message to set the priority of that message to reduce its priority on behalf of a low priority client.
- **MessageManager.FIFO**. Messages shall be received by a module in FIFO order by priority. In other words, high priority messages shall be received first (in FIFO order), followed by lower priority messages (also in FIFO order).
- **MessageManager.PriorityInheritance**. When a high priority message is received, the server task in the receiving module shall have its priority automatically elevated to match that of the message it is processing. The server task's priority shall return to its previous setting after handling the message. ⟨⟨⟨ This requires dynamic priorities and thus is in conflict with **Core.SPARK** ⟩⟩⟩ PET
- **MessageManager.Lost**. Messages may be lost during sending. When a module attempts to send to a full mailbox either a) the message shall be lost with a status return to inform the sender, b) the message shall be lost with no status return, or c) the oldest message with the same or lower priority shall be purged from the mailbox. If no such message exists in the last case (because they all have higher priority), the message shall be lost.

The message manager only provides support for asynchronous messages. No support for synchronous messages is needed. If a synchronous request/reply is required, an explicit reply message must be sent. This means a module may have to field other, unrelated messages while it waits for a reply from an earlier request. This impacts the implementation of modules since they must remember the status of all pending requests. ⟨⟨⟨ The CubedOS library should provide some assistance with this. ⟩⟩⟩ PET However, it also promotes a high degree of concurrency since, in effect, no requests are ever blocking.

Time Server

The Time Server module provides real-time clock services. However, due to the latency and overhead of message passing, modules that require high speed, real-time timing

[1] This facility is implemented in the Merc tool.

services may need to use internal tasks instead. The timeserver is suitable for slow speed or non-critical timing services. ⟨⟨⟨ This needs to be investigated further. What can be specifically said about the timing of the message passing service? ⟩⟩⟩ The following requirements apply: PET

- **Tick.RealTime**. The timeserver shall use a real time clock as the basis of its timing. Such a clock is monotonic and tracks time independently of any software activity.
- **Tick.Periodic**. The timeserver shall provide a service whereby a module can request the delivery of periodic *tick messages* with a period ranging from 1 ms to 1 hour (3,600,000 ms). The total set of tick messages generated in response to such a request is called a *periodic series*.
- **Tick.Latency**. The time between when a request for a periodic series is received by the timeserver and when the first tick message in that series is sent shall be not more than one millisecond plus the latency due to message passing and processing.
- **Tick.OneShot**. The timeserver shall provide a service whereby a module can request the delivery of a single tick message either after a specified delay (in the range from 1 ms to 1 hour), or at a specified absolute time. In this case the response message constitutes a *one shot series* of size one.
- **Tick.SeriesID**. Every request shall contain a *series ID*, specified by the requester, that identifies a particular series.
- **Tick.MultiSeries**. The timeserver shall support multiple series being delivered to the same module. For example a periodic series and a one shot series, or multiple periodic series with different periods, all going to the same module is supported.
- **Tick.Message**. Tick messages shall contain a counter, starting at one, that indicates which message in the series it is.[2] Also tick messages shall contain the series ID number.
- **Tick.Cancel**. The timeserver shall accept messages that cancel a pending[3] or active series. The cancellation message shall contain the series ID. If the ID is invalid, there shall be no effect. ⟨⟨⟨ Implementing cancellation could be tricky, and it may not be important. This requirement should be considered low priority. ⟩⟩⟩ PET

File Server

Not all missions may require a file system, but many will. The core File Server module provides a simple file system interface, but the precise implementation is not specified here. It could be implemented as a layer around a host operating system API, through a third party library, or even as a RAM disk. The following requirements apply:

[2]The counter shall always be one in the case of one shot ticks.
[3]For example a one shot tick that has not yet occurred can still be canceled

- **FileServer.Names**. File names shall be limited to the FAT16 style "8.3" naming convention. This is a least common denominator naming convention that should be supportable by almost any underlying implementation. ⟨⟨⟨ What should we say about the case sensitivity of file names? ⟩⟩⟩ PET
- **FileServer.Size**. File sizes shall be represented using an unsigned integer with at least 31 bits. This supports files potentially as large as $2^{31} - 1$ bytes in size, depending on the underlying implementation.
- **FileServer.FileOps**. The following operations shall be allowed on files: Open, Close, Read, and Write.
- **FileServer.MultiOpen**. It shall be possible to have multiple files open at once. The limit on the number of simultaneously open files is not specified here but should be "reasonable."
- **FileServer.GlobalHandles**. File handles (representing open files) shall be global for the entire system. In particular a handle sent from one module to another shall be usable by the receiving module to access a file opened by the sending module.
- **FileServer.Modes**. It shall be possible to open files for reading, writing, and append mode.
- **FileServer.Access**. It shall be possible to open files for either sequential or random access.
- **FileServer.Sharing**. A file shall be openable many times simultaneously for reading, but any writing or appending shall require exclusive access.
- **FileServer.Binary**. There shall be no distinction between text files and binary files. All data transfers shall be explicit (if a carriage return or line feed character is desired, it shall be written explicitly and processed explicitly on reads).
- **FileServer.Directory**. A single directory of files shall be provided. Subdirectories shall not be provided.
- **FileServer.Attributes**. In addition to its name, a file shall also have the following attributes: Size, Date/Time stamp when last modified. The Date/time stamp shall be updated whenever the file is created or written.
- **FileServer.DirOps**. The following operations shall be allowed on directories: EnumerateFiles, DeleteFile, GetFileAttributes.

Roughly speaking the file server interface should provide all the usual file operations, but with a minimum of extra functionality. The maximum size of data transferred when reading or writing files is left unspecified. It is recommended that implementations allow transfers up to maximally sized messages. PET

⟨⟨⟨ Do we want some kind of "garbage collection" of open file handles? Maybe have handles automatically closed if they are inactive too long? ⟩⟩⟩

Interpreter

The Interpreter module shall be able to run mission specific commands at a set time, expressed either as an absolute time or as a time interval relative to the previously issued command. It shall make use of the Time Server, so all limitations of that service also apply here. The Interpreter is meant to issue raw CubedOS messages to other modules. At this time it is not required to have any inherent command language of its own. However, the addition of "meta commands" that support control flow or subroutines might be an appropriate avenue for future work. The following requirements apply:

- **Interpreter.Commands**. The commands executed by the Interpreter shall be CubedOS messages consisting of a target address triple (Domain ID, Module ID, Message ID) and message arguments already encoded into an octet stream.
- **Interpreter.Ignore**. The Interpreter shall ignore all reply messages sent to it in response to the command messages it sends. The only incoming messages the Interpreter will process shall be those specifically directed to control its functionality (i.e., to provide it with the command sequence to later interpret).
- **Interpreter.Scheduler**. Each command given to the Interpreter shall have an associated time stamp, either an absolute time, a time relative to the previous command, or a time relative to *deployment time*[4]. The Interpreter shall issue commands at their designated time. ⟨⟨⟨ What should happen if the Interpreter encounters a command that must be issued in the past? Should it ignore the command or issue it "immediately?" ⟩⟩⟩ PET
- **Interpreter.ScriptBuilder**. A separate tool shall be provided that can be used to prepare the input for the Interpreter. It as assumed that hand preparation of that input will be excessively difficult and error-prone.

The ScriptBuilder tool has its own requirements, design, and implementation, and is described elsewhere.

Log Server

The Log Server module is used to log messages from other modules. [5] Here we use "log" expansively to include telemetry messages as well as traditional log messages and debug messages.

The log can be stored on non-volatile storage, printed on a console (if one is available), or gathered in a data structure for later transmission to a ground station. Unlike telemetry "data" which might be in an unreadable binary form, telemetry messages are expected to be human-readable. The following requirements apply:

[4] The absolute time when the system is deploy after launch.

[5] Library components shouldn't attempt to log messages directly. They should return appropriate status indications to the calling module and that module can then log an appropriate message if necessary.

- **Log.MessageSize**. Log messages are intended to be short and shall be limited to only 128 characters at most. This effectively creates a lower bound on the size of CubedOS messages at 128 octets since log messages are transmitted to the log server via the usual CubedOS message system. Restricting log messages to 128 characters, however, means that all useful instantiations of the generic Message Manager will be able to pass log messages without issue. The log server should not impose a large message size on applications that don't need it.
- **Log.CharacterSet**. Log messages shall use only printable ASCII characters, free from all control characters (including tabs, carriage return, or line feed). This will minimize problems with processing the messages later.
- **Log.Language**. Log messages shall be written in (American) English using, as much as possible, standard English spelling and punctuation. Log messages are intended to be human-readable.
- **Log.Level**. The log server shall allow messages to be marked with different "levels" including (but not limited to): debug, informational, warning, error, and critical levels. This can be used to classify each message with respect to its criticality.
- **Log.Filtering**. The log server shall provide a runtime way to filter out messages based on their source mode and log level. For example, it should be possible to only log messages at a certain log level and above (to ignore debugging messages), and to do so on a module-by-module bases.
- **Log.Source**. The log server shall automatically prefix log messages with an indication of which module sent the message (domain ID and module ID).
- **Log.Timestamp**. The log server shall automatically record a timestamp for each log message that it receives and prefix the message with the timestamp. The timestamp shall be the number of seconds since the system started and be displayed in at least millisecond precision. The timestamp shall reflect the time the message was generated by the sending module and not, for example, the time it was processed by the log server.
- **Log.Storage**. The log server shall store a certain number of log messages and make them available for later transmission as telemetry messages. ⟨⟨⟨ How many messages should it store? Should that be configurable? ⟩⟩⟩ PET

2.1.4 Hardware Abstraction Layer

PET

⟨⟨⟨ Need to talk about the modules that interface to standard busses and other low level hardware resources. The idea is that "high level" device drivers could be written in terms of low level operations making it possible to port a CubedOS application to a new platform by only rewriting the HAL modules. ⟩⟩⟩

2.2 Architecture

In this chapter we describe the high level architecture of CubedOS. This includes both the architecture of the system and the architecture of the development environment.

2.2.1 Overview

To understand the context of the CubedOS architecture, it is useful to compare the architecture of a CubedOS-based application with that of a more traditional application. Since CubedOS abides by the restrictions of the **Core.Ravenscar** requirement, we must compare CubedOS with other Ravenscar-based approaches.

Figure 2.1 shows an example application using Ravenscar tasking. Tasks, which must all be library level infinite loops, are shown in open circles and labeled as T_1 through T_4. Tasks communicate with each other via protected objects, shown as solid circles and labeled as PO_1 through PO_4.

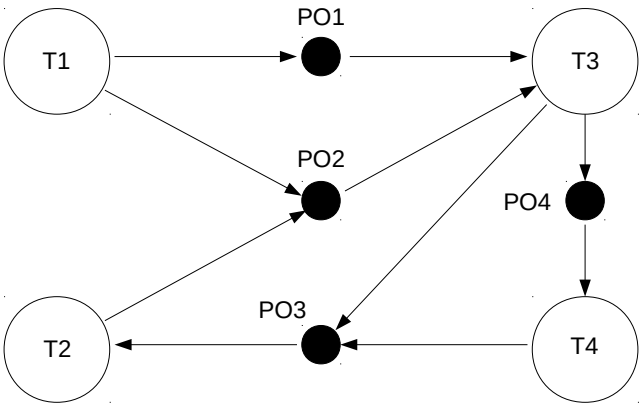

Figure 2.1: Traditional Ravenscar-based Architecture

Arrows from a sending task to a protected object indicate calls to a protected procedure to install information into the protected object. Arrows from a protected object to a receiving task indicate calls to an entry in the protected object used to pick up information previously stored in the object. Entry calls will block if no information is yet available, but protected procedure calls do not block.

Ravenscar requires that protected objects have at most one entry and that at most one task can be queued on that entry. In CubedOS applications each protected object is serviced by exactly one task. This ensures that two tasks will never accidentally be queued on the protected object's entry *provided the mapping from task (actually module, see below) to protected object is truly bijective.* In the figure this means only one arrow

can emanate from a protected object, and each task can have only one arrow incident upon it. Note that this architecture is actually more restrictive than what Ravenscar allows. However, in any case, multiple arrows can lead to a protected object, since it is permitted in CubedOS for many tasks to call the same protected procedure or for there to be multiple protected procedures in a given protected object.

In the example application of Figure 2.1, tasks T_1 and T_3 call protected procedures in two different protected objects. This presents no problems since protected procedures never block, allowing a task to call both procedures in a timely manner. However, task T_3 calls two entries, as allowed by Ravenscar but not CubedOS, one in PO_1 and another in PO_2. Since entry calls can block, this means the task might get suspended on one of the calls leaving the other protected object without service for an extended time. The application needs to either be written so that will never happen or be such that it doesn't matter if it does.

There are several advantages to the traditional organization:

- The protected objects can be tuned to transmit only the information needed so the overhead can be kept minimal.
- The parameters of the protected procedures and entries specify the precise types of the data transferred so compile-time type safety is provided.
- The communication patterns of the application are known statically, facilitating analysis.

However, the traditional architecture also includes some disadvantages:

- The protected objects must all be custom designed and individually implemented, creating a burden for the application developer.
- The communication patterns are relatively inflexible. Changing them requires overhauling the application.
- Third-party library components are awkward to write since they would have to know about the protected objects that are available in the application in order to communicate with other application-specific components. This is a particular problem for client/server oriented systems where a general purpose, reusable server component needs some way to send replies to clients for which it has no prior knowledge. [6]

The CubedOS version of the sample application has an architecture as shown in Figure 2.2. In this case CubedOS provides the communication infrastructure as an array of

[6] With access types, the client could potentially send a "return address" consisting of an access value that points at a client-owned protected object implementing an agreed upon synchronized interface. This design is ruled out by SPARK.

general purpose, protected mailbox objects. CubedOS *modules* communicate by sending messages to the receiver module's mailbox. The messages are unstructured octet streams, and thus completely generic. Each module has exactly one mailbox associated with it and contains a single task (the *server task*) dedicated to servicing that mailbox alone, creating a one-to-one relationship between modules and mailboxes. A module's mailbox servicing task extracts messages from the mailbox, decodes the messages, and then acts on the messages, perhaps sending messages to other modules in the process.

Note that CubedOS modules are allowed to have additional internal tasks, if required, as long as the usual Ravenscar rules are obeyed. These internal tasks can send messages to other modules, but they do not attempt to receive messages; that work is reserved for the server task alone.

The communication connections shown in Figure 2.2 are the same as those shown in Figure 2.1 except that the two communication paths from T_1 to T_3 are combined into a single path going through one mailbox.

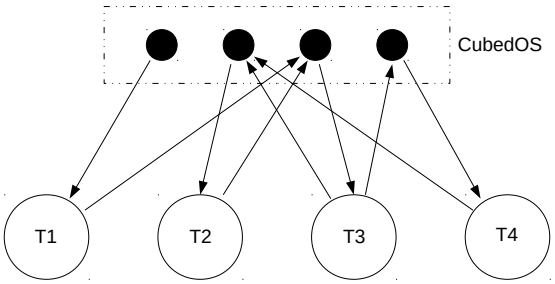

Figure 2.2: CubedOS-based Architecture

CubedOS relieves the application developer of the problem of creating the communications infrastructure manually. Adding new message types is simplified with the help of a tool, `Merc`, stored in a different repository of the CubeSat Laboratory GitHub. In addition to providing basic, bounded mailboxes, CubedOS also provides other services such as message priorities and multiple sending modalities (for example, best effort versus guaranteed delivery). ⟨⟨⟨ These quality of service features are currently completely unimplemented. ⟩⟩⟩ Many of these additional services would be tedious to provide on a case-by-case basis following the traditional architecture. PET

CubedOS also allows any module to potentially send a message to any other module. Thus the communication paths in the running application are very flexible. In particular, implementing reusable server modules becomes very straightforward. Each request message contains the module ID of the sending module, which is simply an index into the mailbox array. The server that receives the message uses that ID to send the reply to the correct mailbox. There is need for the server to have prior knowledge of its clients.

Although the CubedOS architecture supports only point-to-point message passing, CubedOS comes with a library module that supports a publish/subscribe discipline. The module allows multiple channels to be created to which other modules can subscribe. Publisher modules can then send messages to one or more channels. Since the messages themselves are unstructured octet streams, the publish/subscribe module can handle them generically without being modified to account for new message types.

Every CubedOS module has an ID number. These numbers are currently statically assigned by the application developer except for the CubedOS core modules which have "well known" module ID numbers. It is our intention to implement a name service that can map module names to ID numbers dynamically. Roughly, at system initialization time a module registers itself with the name service and received a dynamically assigned ID number. Other modules can resolve the name to that ID number by making a query to the name service. This allows third party library modules the ability to adapt to their environment in a way that would be impossible if they needed statically assigned addresses. Applications using multiple third party library modules would have no way to assure conflict free ID assignments using a static method.

We are also defining standard message interfaces to certain services, such as file handling, that third party modules could implement. This allows modules to use a service without knowing which specific implementation backs that service.

However, CubedOS's architecture also carries some significant disadvantages as well:

- All mailboxes must have the same size since they are stored in an array. This arises because the **Core.SPARK** requirement forbids the use of pointers. Consequently some mailboxes will be larger than necessary, wasting space. ⟨⟨⟨ SPARK no longer forbids pointers outright. Can this requirement be relaxed? ⟩⟩⟩ PET

- All messages have the same type.[7] This means the size of a message must be large enough to satisfy the needs of every module. As a result, in many cases messages will be larger and slower to copy than necessary.

  The common message type also requires that typed information sent from one module to another be encoded into a raw octet format when sent, and decoded back into specifically typed data when received. The encoding and decoding increases the runtime overhead of message passing and reduces type safety. Modules must defend themselves, at runtime, from malformed or inappropriate messages, causing certain errors that were compile-time errors in the traditional architecture to now be runtime errors. This is exactly counter to the general goals of high integrity system development.

- In order to return reply messages, the mailboxes must be addressable at runtime using module ID numbers. Accessing a statically named mailbox isn't general

[7] **Core.SPARK**'s prohibition on pointers prevents more flexible designs. However, a more flexible design might be possible with modern SPARK

enough. As a result, the precise communication paths used by the system cannot easily be determined statically.

In particular, since SPARK does not attempt to track information flow through individual array elements, it is necessary to manually justify certain SPARK flow messages. ⟨⟨⟨ This doesn't seem to be true any longer. ⟩⟩⟩ Although the architecture of CubedOS ensures that there is a one-to-one correspondence between a module and its mailbox. The tools don't know this and the spurious flow messages they produce must be suppressed. PET

The details of CubedOS mitigate, to some degree, the problems above. For example, the mailbox array is actually instantiated from a generic unit by the application developer. This allows the developer to tune the sizes of the mailboxes, and the messages they contain, to the application's needs. CubedOS does not attempt to provide a one-size-fits-all mailbox array that will be satisfactory to all applications.

Also, every well-behaved CubedOS module should contain an API package with subprograms for encoding and decoding messages. This package can be generated by the `Merc` tool. The parameters to these subprograms correspond to the parameters of the protected procedures and entries in the traditional architecture, and provide much of the same type safety. However, using the API subprograms is not enforced by the compiler. It is also possible to accidentally send a message to the wrong mailbox. Thus, modules still need to include runtime error checking to detect and handle these problems.

So far we have described two extremes: a traditional approach that does not use CubedOS at all, and an approach that entirely relies on CubedOS. However, hybrid approaches are also possible. Figure 2.3 shows a combination of several CubedOS mailboxes and a hand-made, optimized protected object to mediate communication from T_3 to T_4.

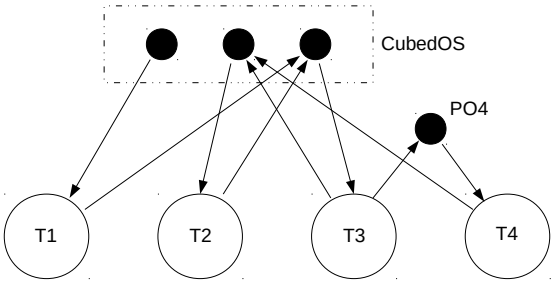

Figure 2.3: Hybrid Architecture

This provides the best of both worlds. The simplicity and flexibility of CubedOS can be used where it makes sense to do so, and yet critical communications can still be optimized if the results of profiling indicate a need. In Figure 2.3 task T_4 can't be

reached by CubedOS messages. The hand-made protected object creates a degree of isolation that can also simplify analysis as compared to a pure CubedOS system. In effect, from CubedOS's point of view, T_4 is an internal task of module #3 and thus part of module #3.

It is also possible to instantiate the CubedOS message manager multiple times in the same application, effectively creating multiple communication domains using separate mailbox arrays. Figure 2.4 shows an example of where T_4 is in a separate domain from the other modules (because it receives from a mailbox that is separate from the others). In a more realistic example, the second communication domain would contain more than one module.

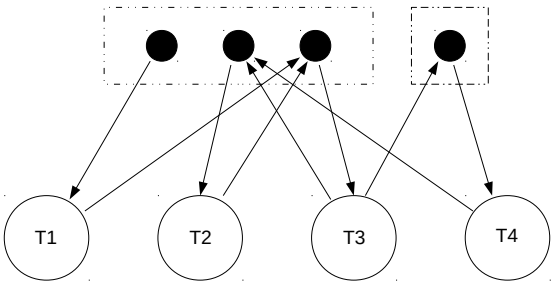

Figure 2.4: Multiple Communication Domains

This approach allows the CubedOS infrastructure to be used for easy development while still partitioning the system into semi-independent sections. For example, the sizes of the mailboxes and of the messages used in each communication domain need not be the same. The parts of the application that require large messages could be grouped into a domain separate from the parts that only require small messages.

Notice in Figure 2.4 tasks T_3 and T_4 send messages into multiple domains. This is, of course, sometimes necessary if the domains are going to interact. In such a scenario each communication domain would regard the other domain as a collection of internal tasks that is part of the module(s) that send messages into the other domain. We currently have very little experience with systems designed in this way, but this is an area for future work.

2.2.2 Message Priorities

In this section we describe the architecture of message priorities in CubedOS. Priorities are built on top of the task priority system provided by the Ada runtime environment. CubedOS does not directly interact with the underlying operating system's notion of

priorities. This provides portability to any system with a suitably capable Ada runtime environment. Specifically, message priorities are supported on selected bare board systems without an operating system.

Justification

PET

⟨⟨⟨ Why do we need this? Fill out this section when we know! ⟩⟩⟩

Architecture

All modules are assigned a "base priority" statically using suitable Ada language pragmas applied to the module's server task. We define the server task of a module as the task that reads the module's mailbox and processes the module's messages. We speak of a module's priority as a shorthand for the "priority of a module's server task." Note that modules are allowed to have internal tasks in addition to the main task. The priority of those tasks is an internal matter that does not concern us here.

CubedOS also allows messages to be given priorities that can be distinct from the priorities of the modules that send and receive them. Message priorities are not a concept known to the Ada runtime environment; they are purely a construct of CubedOS.

When a module sends a message that message is, by default, given a priority equal to that of the sending module. However, a module can optionally lower the priority of a message below its own priority. If a module attempts to raise the priority of a message above its own priority, it is not an error, either statically or dynamically, but the priority is simply set to the module's priority instead. ⟨⟨⟨ It might be better if this was a statically detected error. ⟩⟩⟩ PET

Messages are removed from a module's mailbox in priority order in $O(\log\ n)$ time (where n is the number of pending messages in the mailbox). ⟨⟨⟨ I'm assuming the mailboxes are priority heaps ⟩⟩⟩ When a module with priority m is processing a message at priority $p > m$, the module's priority is dynamically increased to p and any messages it sends, including replies to the original message, are sent by default at priority p (although the module can downgrade the priority of sent messages, as usual). When a module completes the processing of the high priority message, just before it fetches the next message from the mailbox, its priority is returned to its original base priority level. PET

When a module with priority m is processing a message at priority $p < m$, the module's priority remains unchanged at its base priority. Thus, a high priority module services low priority messages at its usual (high) priority. In contrast, a low priority module services high priority messages at a temporarily elevated priority. This *priority inheritance* is needed, so a high priority client isn't slowed down when it requests service from a low priority module. When executing on behalf of a high priority client, a low priority server inherits the priority of the client temporarily.

This structure also allows a high priority module to downgrade the priority of outgoing messages when servicing a low priority client. The idea is that the high priority module might want to make requests to other modules using the client's priority (which is available to it in the original request message sent by the client). It is up the application developer to do this if it is desired. The default behavior is for all sent messages to be given the (dynamic) priority of the sender.

Static Analysis

Unfortunately SPARK does not currently support the dynamic modification of task priorities. Thus, use of that feature in a CubedOS application must be specifically justified to avoid spurious SPARK diagnostics. However, doing this means that SPARK can no longer be relied upon to find all potential deadlocks or livelocks. It is therefor necessary to employ an additional form of analysis that can supplement SPARK and statically show freedom from those issues in any case. This additional analysis has yet to be defined. It can, however, take advantage of certain restrictions that still remain in CubedOS tasking despite the addition of dynamic priorities.

- Modules only interact via message passing.
- Messages carry priorities and module priorities are dynamically adjusted as described above.
- No other dynamic priority adjustments are allowed. ⟨⟨⟨ It might be necessary to forbid a module from lowering the priority of its outgoing messages... perhaps in a first implementation anyway ⟩⟩⟩ This is statically checked with a supplementary tool. ⟨⟨⟨ Using libadalang, perhaps? ⟩⟩⟩ PET PET PET

⟨⟨⟨ I wonder if spin can be used to do what we need. Perhaps a spin module can be automatically extracted from the Ada source using a supplementary tool. That same tool could perhaps do the additional static checking mentioned above ⟩⟩⟩

2.2.3 Message Encoding

CubedOS mailboxes store messages as unstructured octet arrays. This allows a general purpose mailbox package to store and manipulate messages of any type. Unfortunately this also requires that well-structured, well typed message information be encoded to raw octets before being placed in a mailbox and then decoded after being retrieved from a mailbox.

The CubedOS convention is to use External Data Representation (XDR) encoded messages. XDR is a well known standard [8] that is also simple and has low overhead. We have defined an extension to XDR that allows SPARK's constrained scalar subtypes to be represented and that allows limited contracts to be expressed. The tool Mercury

compiles a high level description of a message into message encoding and decoding subprograms. Our tool is written in Scala and is not verified, but its output is subject to the same SPARK analysis as the rest of the system. In our case it is easier to prove the output of Mercury than it is to prove the correctness of Mercury itself.

The use of Mercury mitigates some of CubedOS's disadvantages. The developer need not manually write the tedious and repetitive encoding and decoding subprograms. Furthermore, those subprograms have well-typed parameters thus shielding the programmer from the inherent lack of type safety in the mailboxes themselves.

To illustrate CubedOS message handling, consider the following short example of a message definition file that is acceptable to Mercury.

```
enum Series_Type { One_Shot, Periodic };

typedef unsigned int Module_ID range 1 .. 16;
typedef unsigned int Series_ID_Type range 1 .. 10000;

message struct {
    Module_ID      Sender;
    Time_Span      Tick_Interval;
    Series_Type    Request_Type;
    Series_ID_Type Series_ID;
} Relative_Request_Message;
```

This file introduces several types following the usual syntax of XDR interface definitions. The syntax is extended, however, to allow the programmer to include constrained ranges on the scalar type definitions in a style that is normal for Ada. The message itself is described as a structure containing various components in the usual way. The reserved word message prefixed to the structure definition, an extension to XDR, alerts `Merc` that it needs to generate encoding and decoding subprograms for that structure. Other structures serving only as message components can also be defined.

Mercury has built-in knowledge of certain Ada private types such as Time_Span (from the Ada.Real_Time package). Private types need special handling since their internal structure can't be accessed directly from the encoding and decoding subprograms. There is currently no mechanism in Mercury to solve this problem in the general case.

Each message type has an ID number that is unique across the application. This is required to distinguish valid messages from invalid ones. Mercury assigns these message type ID numbers automatically. The encoding subprogram installs the ID number into the message as a hidden component. The decoding subprogram checks the message type ID number and returns a failure status if it is invalid. A CubedOS module extracts a message from its mailbox and then applies the relevant decoder subprograms to that message until one succeeds. If all decoder subprograms fail the message either has an

unexpected type (for example it was accidentally sent to the wrong mailbox) or the message is malformed in some way. The module must then make a runtime decision about how to handle that situation.

Mercury is a work in progress. We intend to ultimately support as much of the XDR standard as we can including, for example, variable length arrays and discriminated unions.

2.2.4 Code Organization

In this section we describe the layout and organization of the source code of CubedOS. If you intend to work on the development of the software you should read this section carefully.

- `src`. The main source folder. The code in this folder is hardware independent; it calls into various packages to interact with low level devices. All the code in this folder will fly in space and must be SPARK.

  Note that this folder is taken as the root folder for all relative paths used in this document from now on. This folder will now be referenced as the "root folder" or as "." depending on the context.

- `Check`. This folder contains the main file and all supporting code for the AUnit based unit test program. Hardware specific packages, if they are implemented at all, contain stubs that receive data from the higher level packages and present that data to the test program for evaluation. The file `check.adb` is the test program's main file. None of the code in this folder needs to be SPARK since it will not fly in space.

- `Mock`. This folder contains software simulations of the low level hardware devices. Using these simulations it is possible to build CubedOS as an executable file on a conventional operating system such as Windows. The simulated hardware logs activity for subsequent analysis. Unlike the unit test program, the Mock program is a complete version of the system as it will fly in space. None of the code in this folder needs to be SPARK.

- `CubeSat`. This folder contains the actual code needed on the CubeSat. All the code in this folder will fly in space and must be SPARK.

In addition to the source folders, there is a `build` folder as well. This folder contains subfolders for the various build configurations that are defined. The object files and executable files are placed in these subfolders.

The `build` folder also contains some test programs used to exercise the physical hardware in various ways. See the `README.txt` files in those subfolders for more information.

At the time of this writing all SPARK artifacts are put into the `CubeSat` build folders. The project files thus use the CubeSat build as their default build.

2.3 Design

In this chapter we describe the design of CubedOS, including various design trade-offs made. This information is primarily of interest to developers looking to enhance the CubedOS infrastructure itself. Developers interested in only using CubedOS do not need to read this section unless they are interested in the rationale behind some of CubedOS's design decisions.

2.3.1 Message Manager

One of the central problems facing the message manager is the handling of message mailboxes. Since SPARK does not allow dynamically allocated memory, any method that involves dynamically allocating space for messages is not an option. Instead, mailbox space must be allocated statically. This forces users of CubedOS to anticipate the maximum sizes of the mailboxes they require.

We note that an upcoming version of SPARK *will* allow the use of dynamically allocated memory, although in a restricted way. This opens the possibility of revisiting the design described here and perhaps working around some of the issues we identify below. We have not yet investigated this in detail; SPARK's support for dynamically allocated memory is still immature.

We considered three strategies for managing mailboxes.

- *Define a Mailbox type that can hold a fixed number of pending messages. Create a separate mailbox for each module.* This approach has the advantage of being simple. It also separates the total mailbox space by module. The value of the separation is that should one mailbox fill due to a dead or slow module, that won't impact the communication between other modules using different mailboxes.

  The main problem with this approach is that it tends to waste space. We assume most modules won't require many pending messages (a large mailbox size). Yet if even one module does require a large size, the Mailbox type must be adjusted to accommodate it, causing unnecessarily large mailboxes in most cases.

- *Define a single Mailbox object that holds pending messages for all modules together.* This approach addresses the disadvantage of the previous approach by consolidating the free space into a single mailbox object. Modules requiring many pending messages can in effect "borrow" free mailbox space from modules with few pending messages.

  Implementing this design is more complicated, but a bigger disadvantage is that modules can interfere with each other. If one module is slow and gets backlogged, the (one and only) mailbox could fill up with messages for that module, thus preventing communication between other, unrelated modules. While some

workaround to this problem could probably be implemented, we feel this is a serious enough problem to rule out this design choice.

- *Define a Mailbox type that is discriminated on its size. Create a separate mailbox for each module, tuned for that module's needs.* This is a variation of the first approach except here the mailbox instances can each have a different size. The author of a module defines the size of the mailbox needed for that module; large mailboxes are only used where necessary and space is conserved. This approach also maintains the freedom from interference provided by the first approach.

The third approach is appealing but unfortunately does not allow replies to arbitrary modules in an environment where pointers are prohibited. Although a target mailbox can be mentioned by name when sending a message, the destination of any reply must be computed at runtime. Without pointers there is no way to create a map from module ID number to the mailbox for that module. The first approach can work around this problem by putting all mailboxes in a common array and using the module ID number as an array index. However, that also requires all mailboxes to be the same size.

A more subtle problem with the third approach is that, in general, the author of a module can't know the maximum number of pending messages that are appropriate to specify for that module. The correct number is a system-wide artifact that depends in large measure on the number and rate of message send operations done by other modules. Setting the mailbox sizes is something best done during system integration, and that is more easily accomplished if all mailboxes are in their own package.

For these reasons CubedOS follows the first approach, using a generic package to supply the mailboxes. The package can be instantiated differently for each CubedOS application with mailbox sizes set appropriately for that application.

Selecting the most appropriate size for the mailboxes is the one major issue remaining. If a mailbox fills during operation, sending further messages to that mailbox will fail. Senders do not block because doing so would introduce the risk of deadlock. Thus, failure to send a message due to a full mailbox creates awkwardness for senders who must manually retry the send operation, if desired. Furthermore, senders will not likely want to retry sending indefinitely and risk being tied up doing a send operation that will never succeed. This leaves open the question of how the failed send should ultimately be handled.

It would be better if mailbox sizes were tuned so that sending could never fail. However, doing this requires high level reasoning about the flow of messages in the system. We propose constructing a formal model of module communication that includes finite sized mailboxes, and then use a model checker (such as `spin`?), or some other tool, to find a configuration of mailbox sizes where no send operation can ever fail. Doing this is an avenue for future work.

In lieu of such an analysis, human reasoning could be used to estimate appropriate mailbox sizes followed by rigorous testing. Of course, such an approach is likely to be

more error-prone than the formal approach.

If it can be formally shown that no send operation can ever fail, all calls to the Send procedure could be replaced with calls to Unchecked_Send which does not return an error indication. Then all error handling in the modules related to failed sending could be removed. This would be a great simplification. However, we do *not* recommend this approach. Despite any formal modeling we might do in CubedOS's initial design, we foresee several possible ways mailboxes might overflow anyway:

1. Our formal model is incorrect.
2. We never get around to creating a formal model that works.
3. Users of CubedOS might not be in a position to augment the formal model to account for their own, application-specific modules.
4. A module thread dies for reasons outside the software's control (such as hardware failure). Although it might be possible to incorporate this scenario into the formal model, forcing freedom from mailbox overflow in this situation might impose unreasonable requirements on mailbox size or application architecture (or both).

For these reasons we advocate continued checking for failed send operations even if none are expected in normal operation. PET

⟨⟨⟨ We should also say a few words about selecting the maximum size of messages, and about the handling of module ID numbers. For example, we originally considered allowing ID numbers to be dynamically assigned and then using a name resolver module to map module names to their (dynamic) ID numbers. However, that design was overly complicated. ⟩⟩⟩

2.3.2 Message Encoding

To remain generic the messages sent between modules are essentially nothing more than arrays of raw octets. Any structure on the information in a message is imposed by agreement between the sender and receiver of that message. This creates a degree of type unsafety and requires modules to defend against malformed messages at runtime. To mitigate this problem, each module contains an API package that provides convenience subprograms that encode and decode messages. The parameters of these subprograms are strongly typed, of course, so as long as they are used consistently compile-time type checking is retained.

Since the problem of encoding and decoding typed data has been solved before we leverage an existing standard, name External Data Representation (XDR) [8]. This standard defines a "wire format" (in our case a message data format) for various types and simple data structures. The advantage of using XDR is that it is well specified and understood. Using it simplifies our specification needs and invites the use of third party tools to manage message data.

We also considered using Abstract Syntax Notation One (ASN.1) [9] as a message data format. Unlike XDR, ASN.1 includes type information in the data stream allowing for more flexible message structures. However, ASN.1 also has more overhead, both in terms of space used in the message and time used to encode and decode the messages. For this reason we elected to use the simpler XDR approach.

The encoding/decoding subprograms in the API packages must contend with the richness of Ada's type system. In particular, they should be able to handle multiple parameters of various types, called *messageable types*, where a messageable type is defined inductively as follows:

- An integer type, an enumeration type, or a floating point type.
- An array of messageable types.
- A record containing only messageable type components.

Roughly a messageable type is any type allowed by the XDR standard (although we do not aim to provide support for XDR discriminated union types at this time).

The encoding and decoding of messageable types is tedious and repetitive. Supporting subprograms to handle basic scalars and simple arrays are provided by way of a library package `CubedOS.Lib.XDR`. The handling of XDR arrays and records can be done using the lower level subprograms provided by the library.

Since the creation of the API packages is largely mechanical, CubedOS comes with a tool, `Merc`, that converts a textual representation of the message types into a corresponding API package written in provable SPARK. In effect, `Merc` plays a role similar to that played by `rpcgen` in many ONC RPC [10] implementations. In particular, it converts a high level interface description into code that encodes/decodes messages to/from the implementer of that interface. However, `Merc` differs from other similar tools in several important respects:

- It targets CubedOS; the generated code makes use of CubedOS's XDR library for handling primitive types.
- It generates provable SPARK 2014 code.
- It contains extensions to the language in [8] that support Ada's ability to define range constrained scalar subtypes.

A CubedOS application developer defines the messages she needs for a module in an XDR file and then uses `Merc` to generate the API package and related materials. The developer can then the subprograms in that package to encode messages for that module or decode messages from that module. In addition, `Merc` provides subprograms the module author can use to decode incoming messages and encode replies.

Currently, it is still possible for a developer to directly access message data but doing this is strongly not recommended. ⟨⟨⟨ This could possibly be enforced by making the message PET

a private type. However, that would require all the generated API packages to be children of the generic message manager package (and hence generic themselves). This might be an overly complicated design. ⟩⟩⟩

3 Manual

This chapter contains the user's manual for CubedOS. The intended audience of this chapter is CubedOS application developers. Contributors to the CubedOS core and runtime library may also find the information in this chapter useful.

The information in this chapter is intended to support both new and experienced CubedOS application developers. If you are new to CubedOS and not sure how to get started, begin by reading the Section 3.1 tutorial that describes how to create a CubedOS application from scratch.

Documentation on the CubedOS core modules is in Section 3.2. Documentation about the sample programs provided with the CubedOS distribution is in Section 3.3.

3.1 Tutorial

Welcome to CubedOS!

3.1.1 Introduction

This document is for new CubedOS developers who want to learn how to build CubedOS applications. Here we will walk you through the process of creating a simple application and then refer you to the `samples` folder in the CubedOS GitHub repository for additional examples.

CubedOS applications consist of one or more *domains* where each domain consists of one or more *modules*. Simple (and even many not-so-simple) applications are built as a single domain. Multi-domain applications are outside the scope of this tutorial, but the Networking sample shows an example of such an application.

Each module in a domain communicates with other modules by passing messages. Sometimes modules act as servers; they receive request messages, process those requests, and return reply messages to the requester. Sometimes modules act as clients; they send request messages to servers, receive the replies, and do other processing. It is normal for modules to play both roles at different times; they receive request messages and send requests of their own as they process their incoming requests.

CubedOS encourages a client/server application architecture, or, more generally, a distributed application architecture with messages being sent between modules freely. However, the Request/Reply terminology is conventionally used throughout. In some cases "requests" are sent even when no reply is expected, and in other cases "replies" are sent unsolicited.

Normally modules are written in SPARK/Ada as a package hierarchy. The top-level, parent package has the same name as the module. Child packages are used to implement the various subsystems within a module. Some child packages have conventional names, as we will describe later. This precise organization is not strictly required, but it is recommended and certain aspects of the system assume this organization. It is possible, in principle, to write CubedOS modules in other languages such as full Ada or even C. We do not discuss how to do that in this tutorial.

Communication in CubedOS is point-to-point, asynchronous, and unreliable. There is no built-in mechanism for broadcasting or multicasting messages; doing so requires writing code that explicitly sends multiple point-to-point messages as needed. There is, however, a publish/subscribe core module that offers a "channel" abstraction. Modules can subscribe to particular channels and receive messages that are published to those channels by other modules. This decouples senders and receivers so they are not aware of each other and provides a form of multicasting.

Messages are asynchronous in the sense that the sender is allowed to continue before the message has been received. The send operation only queues the message for later delivery. After a successful send, the sending module can reuse any data structures it populated for the outgoing message. Every module has at least one task, called the *module main task* that receives and processes messages. Modules may have other internal tasks for other purposes. Multiple modules execute in parallel. It is possible, even common, for senders to send multiple requests, possibly to different modules, before getting any replies, and replies may arrive in a different order from which the requests were sent. It is up to you to coordinate this activity in your modules, although CubedOS provides some assistance, as we will discuss.

Messages are unreliable in the sense that final delivery is not guaranteed. Senders can elect to receive a status code to inform them if their messages are dropped by the message passing infrastructure, but even this status code does not indicate if the message was actually received by the destination module (for example, if the destination module is stuck or crashed or unreachable over a network). If a server module wishes to indicate an error in a request, or any other kind of failure, it must send an explicit reply message saying so.

Every CubedOS domain contains an array of "mailboxes" that hold pending messages. There is a one-to-one correspondence between the modules of a domain and the mailboxes. In other words, each module has exactly one mailbox and each mailbox services exactly one module. The main task of a module is normally blocked trying to read a message from that module's mailbox. When a message is available, the main task fetches

it, processes it, and then (likely) blocks again trying to read the next message. Since the mailboxes have, in general, space for holding multiple messages, it is possible that a message can arrive while the module was processing an earlier message. In that case, the new message is retrieved at once. Modules never process more than one message at a time. Outstanding messages are queued in the module's mailbox. The mailbox array thus defines the domain and forms the communication infrastructure for the domain.

In normal operation, all module tasks are blocked, and the CPU is idle. When a hardware interrupt is generated, an interrupt service routine inside one a driver module for the associated hardware device activates and sends a message, as appropriate, to another, higher-level module to handle the interrupt. That module may send other messages, etc., creating a cascade of activity in the system. Once the interrupt has been fully handled and that activity dies down, all modules return to their quiescent state of being blocked waiting for a message. If multiple interrupts happen at about the same time, this presents no problem as the modules are able to queue messages and execute in parallel as needed. However, the modules do need to be coded with this scenario in mind. CubedOS is thus a *reactive* programming system since it reacts to external input, but does not typically initiate any activity.

CubedOS comes with a collection of *core modules* that provide services commonly needed by many applications. The publish/subscribe server mentioned earlier is an example of a core module. There is also a timeserver module that can be requested to send messages to other modules at specified times or periodically. "Reacting" to such messages gives modules a way to perform activities without other specific hardware stimulation (i.e., routine monitoring, telemetry reporting, and so forth).

CubedOS also comes with a collection of *library packages* that contain general-purpose reusable code of various kinds. Unlike a module, a library package does *not* contain a task[1]. Subprograms inside library packages are called in the usual way and return results in the usual way, and not via messages[2].

Library packages can also be shared by modules. One consequence of this is that library packages can have no modifiable global data[3] since they might be called by multiple tasks at once. All task interactions are via the message passing mechanism. Modules should not attempt to communicate via global data inside library packages.

The rest of this document walks you through the steps for creating a simple, single-domain CubedOS application. You can use these steps as a framework for creating your own applications. Although we build the application here incrementally for educational purposes, you can review the final form of the application in the `moonshot` folder of the `samples` folder in the repository.

[1] In a previous version of CubedOS, library packages were called "passive modules" to distinguish them from active modules with tasks. That terminology has been dropped.
[2] It is permitted for a library package to send a message as a side effect.
[3] Unless wrapped in a protected object.

3.1.2 Moonshot Application

The application we will build demonstrates many, but not all, aspects of CubedOS development. It is intended to be easy to understand, implement, and modify. It is not intended to be an exhaustive demonstration of CubedOS and its libraries.

We will create an application, called *Moonshot*, that roughly simulates a CubeSat mission orbiting the Moon. It is not our intention for this application to be an accurate simulation of a real mission. Many details will be left out or simplified. However, imagining that the application might be controlling a real spacecraft will hopefully make it more fun to think about!

Although in a real mission, the application would obviously need to control real hardware, in this tutorial we will simulate the hardware. The modules that might ordinarily serve as device drivers will instead implement "mock" devices with simulated behavior. This allows the application to run on any development system you might have available.

Moonshot will make use of several CubedOS core modules. Specifically:

1. The Log Server will be used to log messages of various kinds for later transmission to the Earth. The Log Server is widely used. Almost all CubedOS applications will require it.

2. The Time Server will be used to trigger periodic actions in other modules. In Moonshot, without any real hardware to generate interrupts, the Time Server will be the initiator of all activity once system initialization is complete.

3. The File Server will be used to store image files taken by the mock camera. The Moonshot mission will entail taking picture of the lunar surface, so we need a place to store those images. In a real mission it is common for there to be some sort of non-volatile storage precisely for this purpose.

4. The Publish/Subscribe Server will be used to allow modules to subscribe to various channels. The point of the Publish/Subscribe Server is to decouple senders and receivers of messages so that reusable modules can be more readily created. We describe this in more detail later.

Moonshot will also have several application-specific modules for managing the (mock) hardware devices. Specifically:

1. The Camera module will simulate a camera that can take pictures of the lunar surface.

2. The Radio module will simulate a radio that can send messages to and receive messages from the Earth.

3. A Thruster module will simulate maneuvering thrusters that can adjust the orbit and orientation of the CubeSat.

4. A Controller module that will server as the main control logic for the mission. It will coordinate the action of the other modules to fulfill the mission objectives.

To get started, you will need space for your application's files. You will also need to clone the CubedOS repository from GitHub. At the time of this writing CubedOS is distributed in source form via its development repository. There is no release distribution yet.

The folder for your application need not be included in the CubedOS repository folder. In fact, we recommend that you do *not* include it there. Instead, create a folder somewhere else on your system for your application. You can organize this folder as you please, but for this example, we will assume all the source files are stored directly in the folder.

It is often useful to use an existing application as a baseline for new work. The Echo sample in the CubedOS repository is a good place to start. First, create a project file in your application folder named `moonshot.gpr` that contains the following:

```
project Moonshot is
   for Main use ("main.adb");
   for Object_Dir use "build";
   for Source_Dirs use (".", "../../src/modules", "../../src/library");
   for Languages use ("Ada");

   package Compiler is
      for Default_Switches ("Ada")
        use ("-gnat2022",     -- Use Ada 2022
             "-gnatW8",       -- Enable UTF-8 source files
             "-fstack-check", -- Enable stack checking
             "-gnatwa",       -- Enable all warnings
             "-gnata",        -- Enable assertions
             "-g");           -- Enable debugging
   end Compiler;
end Moonshot;
```

The source paths should include the current folder (“.”), or wherever you plan to store the source files for your application, together with the two source folders from the CubedOS repository. Edit the paths above as appropriate for your system. Create a folder named `build` in your application folder (so the relative path in the project file works) to hold the build artifacts of the application. The default switches are fine for this getting started application. They enable all warnings, runtime assertion checking, and debugging support. In a real application you may want different switches or multiple build scenarios.

Notice that CubedOS applications are intended to use Ada 2022 as their base language. Although this is not strictly necessary, Ada 2022 does provide some niceties that are helpful. Finally, the `-gnatW8` switch enables support for Unicode UTF-8 encoded source

files, allowing certain Unicode characters, such as Greek letters, in identifier names. That is also not strictly necessary, but it can be helpful for clarity in some cases.

Every Ada program requires a library-level, parameter-less procedure to serve as the program's entry point. CubedOS applications don't actually use this procedure, but it is required for the Ada language. Use the name `main.adb` as indicated above. We will describe what must be in this file later.

Filling in the rest of the application requires the following steps:

1. Instantiating the CubedOS generic message manager to create a domain (mailbox array) for your application.
2. Writing a name resolver package to assign addresses to the modules you will use, including the core modules.
3. Creating skeletons for the modules you will write by copying the templates from the CubedOS repository and editing them.
4. Writing the Ada-required main procedure to ensure the modules you need will be compiled into your executable.
5. Verifying that your completed skeleton application builds and runs.
6. Writing a publish/subscribe channel definition package to define the channels you will use in your application.
7. Writing MXDR interface definition descriptions for the messages that will be sent between modules, and using the Mercury tool to generate the Ada code for those message encoding and decoding subprograms.
8. Implementing the full logic of your application by filling in the skeletons you created earlier.

3.1.3 Instantiating the Message Manager

Every domain in CubedOS requires a message manager to handle the message passing between modules. The message manager is a generic package that is instantiated with several arguments that define the number of modules in the domain and the dimensions of the mailboxes used by those modules.

At the time of this writing, the generic instance *must* be called Message_Manager. This name is hard-coded in the core modules. In a multi-domain application, you would have multiple instances of the generic message manager, one for each domain, but since every instance must have the same name, they must reside in different executable units. This is a limitation of the current CubedOS design that might be lifted in the future, if it proves to be problematic.

Here is a suitable instantiation of the generic message manager for the Moonshot application:

```
with CubedOS.Generic_Message_Manager;
pragma Elaborate_All(CubedOS.Generic_Message_Manager);

package Message_Manager is
  new CubedOS.Generic_Message_Manager
    (Domain_Number =>  1,
     Module_Count  =>  8,
     Mailbox_Size  =>  8,
     Maximum_Message_Size => 1024);
```

The meaning of the generic arguments is as follows:

Domain_Number The domain number is a unique identifier for the domain. It is a positive integer. For a single domain application, such as Moonshot, the domain number should always be 1.

Module_Count The module count is the number of modules in the domain. For the Moonshot application, we have four application-specific modules (Camera, Radio, Thruster, and Controller) and four core modules (Log Server, Time Server, File Server, and Publish/Subscribe Server). That makes a total of eight modules.

Mailbox_Size The mailbox size is the number of messages that can be queued in a module's mailbox (also called the "depth" of the mailbox). The mailbox size should be large enough to hold all the messages that might be sent to a module before it can process them.

It can be difficult to decide the most appropriate value for this parameter. If a module can always process messages quickly, or if messages arrive infrequently, a small mailbox size is appropriate. However, if a module can be slow to process messages or if messages can sometimes arrive quickly, a larger mailbox size is needed. A mailbox size that is too small can lead to messages being dropped. However, a large mailbox size wastes memory.

Be aware that all modules in the same domain will use the same mailbox size. Choose a size large enough to satisfy the needs of the module requiring the most depth of any in the domain. A mailbox size of 8 is a reasonable default.

Maximum_Message_Size The maximum message size is the maximum number of 8-bit bytes (also called "Octets") that can be sent in a single message. The maximum message size should be large enough to hold the largest message that will ever be sent in the domain.

The total amount of memory (not counting certain overheads) required for the mailboxes is the product of Module_Count, Mailbox_Size, and Maximum_Message_Size. In the Moonshot application, the total memory required for the mailboxes is $8 \times 8 \times 1024 = 64$ KiB. Depending on the platform on which the application will run, this may be a significant amount of memory. Be sure to consider the memory requirements of your application when choosing these parameters.

3.1.4 Writing the Name Resolver

In CubedOS, every module has an address represented by a pair of integers (Domain ID, Module ID). In the Moonshot application, the domain ID is 1, and the module IDs are in the range from 1 to 8 inclusive.

You must assign these module IDs to the modules yourself, in any way you see fit (it could be arbitrary), but you must ensure that each module has a unique ID. The core modules do *not* have predefined ID numbers. You must assign them as well.

The ID assignments are made in a package called Name_Resolver. As with the message Manager package, the name resolve package must also have a specific name because references to it are hard coded in the core modules. Here is a suitable Name_Resolver package for the Moonshot application:

```
with Message_Manager; use Message_Manager;

package Name_Resolver is

   -- Core Modules
   Log_Server              : constant Message_Address := (0, 1);
   Time_Server             : constant Message_Address := (0, 2);
   File_Server             : constant Message_Address := (0, 3);
   Publish_Subscribe_Server: constant Message_Address := (0, 4);

   -- Application-Specific Modules
   Camera                  : constant Message_Address := (0, 5);
   Radio                   : constant Message_Address := (0, 6);
   Thruster                : constant Message_Address := (0, 7);
   Controller              : constant Message_Address := (0, 8);

end Name_Resolver;
```

The Message_Address type is defined in the Message_Manager package. It is a record type with two components for the domain ID and module ID.

Note that, although all the modules in our application are in domain 1, the value 0 is used for the domain ID in the Name_Resolver. This value is treated as "this" domain by the message passing system. If a module sends a message to domain 0, it is sending that message to a module in the same domain as the sender. The explicit value of 1 would also work in this case, but using 0 is a bit more general. The use of domain ID 0 is more important in a multi-domain application when you want to send a message to a module in the same domain without knowing your own domain ID.

It is important that you assign unique module IDs to each module in your application. If two modules are given the same ID, they will both try to fetch messages from the same mailbox. Although, it is permitted under the Jorvik profile used by CubedOS for multiple tasks to queue on the same mailbox entry, it is unspecified which messages will be picked up by which tasks. This will result in the two conflicting modules receiving many messages intended for the other, producing many "invalid message" errors at runtime.

It is also important that you don't accidentally assign a module ID outside the range of allowed ID values implied by the Module_Count parameter of the Message_Manager instantiation. However, in this case you will get compile-time errors, so it is less likely to be a problem.

Finally, there is no requirement that the module IDs be contiguous or start at 1. For example, you could assign the core modules IDs 1 through 4 and the application-specific modules IDs 10 through 13, leaving the range 5 through 9 reserved for future expansion without renumbering the existing modules. This does cause the mailboxes to use more memory than is necessary, since you are allocating mailboxes for modules you aren't using. Also, renumbering modules later in development is not necessarily a problem since the compiler will manage the module ID values for you when it rebuilds your application.

At the time of this writing, module ID 2 is reserved for the inter-domain router used in multi-domain applications. This router is internal to CubedOS and not something application developers interact with directly. In a multi-domain application, module ID 2 cannot be used for either a core module or an application module. However, in a single domain application such as Moonshot, module ID 2 is available for use.

A Word About Addresses

As you have seen modules have two-dimensional addresses in CubedOS, consisting of domain ID, module ID pairs. Each module also has a collection of messages that it can send and receive, and those messages have IDs as well. The message IDs are local to the module. The module IDs are local to the domain. The domain IDs are globally unique.

The full address of a message is a triple consisting of the domain ID, module ID, and message ID. For example: $(1, 3, 2)$ is the address of message 2 in module 3 of domain 1.

This is a different message from $(1, 3, 3)$ in the same module, and different from $(1, 4, 2)$ in a different module in the same domain, and of course different from $(2, 3, 2)$ in a different domain altogether. In most cases, you do not need to work with message IDs directly. The CubedOS message passing system takes care of that for you. However, you should be aware of this addressing scheme in case you need to debug a problem or understand the CubedOS internals.

3.1.5 Creating Module Skeletons

Now we are ready to start building the modules for the Moonshot application. To start, we will focus on the controller module. The controller module is the main control logic for the mission, and thus serves as the "main" program for the application.

CubedOS modules are organized as a package hierarchy. The top-level package has the same name as the module, and is mostly used as a place to hang various child packages that implement the module's logic. Here is a complete top-level package for the controller module:

```
pragma SPARK_Mode(On);

package Controller is
  -- Empty
end Controller;
```

This package only serves to form the root of the module's package hierarchy, and thus does not need to contain any functionality (or have a body). It might be reasonable to include some module-wide declarations here, such as types needed throughout the module, but in this case we have none.

Recall that in Ada, child packages have visibility into their parents automatically (meaning without using **with** statements). The bulk of the module can be in child packages, or even in grandchild packages, and yet can see the module-global declarations in the top-level package without making those declarations global program-wide. This is a nice example of Ada's scope and visibility-management features in action.

CubedOS modules have at least two child packages: API and Messages.

The API package contains subprograms for creating messages from well-typed values, breaking messages back into well-typed values, and testing messages for validity. A more typical API is a collection of subprograms (functions, methods, etc.) that take well-typed parameters, perform some action, and return a result. In CubedOS, modules communicate by sending messages. The API is thus about which messages a module receives (called "Requests") and sends (called "Replies"). The API package provides subprograms that manipulate these messages.

Characterizing messages as requests and replies follows from the view of message passing as a kind of client/server interaction where the client makes a request, and the server replies to it. However, in CubedOS, modules can act as both clients and servers, depending on what they are doing. Each module specifies in its API what requests it will accept and what replies it may produce. Also requests can be sent without expecting a reply, and replies can be sent without ever receiving a request (i.e., unsolicited replies).

The CubedOS repository includes a folder `templates` that contains template files for the API and Messages packages. You can copy these templates into your module's folder and edit them to suit your needs. They are heavily commented to help guide you. The expectation is that you will ultimately remove most, if not all, of the provided comments and replacement them with application-specific comments as appropriate.

The Messages package contains a task that fetches a message from the module's mailbox, processes that message, and then fetches the next message. This task is called the module's main task.

The template files in the `templates` folder show the recommended structure. The implementation of the main task (at the bottom) is very simple. It immediately passes each message it receives to procedure Process. That procedure, in turn, examines the message and calls the appropriate subprogram to handle that message.

Handling a message entails using an API decoding procedure to first break the message into its well-typed components, and then acting on the message. Normally, this involves sending other messages to other modules, and ultimately, in most cases, sending a reply message back to the requester.

Copy the sample module templates to your working folder, renaming the files as appropriate for the Controller module. Edit the templates to change names, as necessary. For now, don't try to implement any real logic. Let's get the skeleton to compile and run first.

3.1.6 Writing the Main Procedure

All Ada programs require a library-level main procedure. In a CubedOS application, this procedure does nothing, but it is where we will inform the compiler of the modules we want to include in the executable, so it knows what code to locate and compile. Here is a suitable main procedure for the Moonshot application so far:

```
-- Bring in the necessary modules.
with CubedOS.Log_Server.Messages;
with CubedOS.Time_Server.Messages;
with CubedOS.File_Server.Messages;
with CubedOS.Publish_Subscribe_Server.Messages;
with Controller.Messages;
```

```
pragma Unreferenced(CubedOS.Log_Server.Messages);
pragma Unreferenced(CubedOS.Time_Server.Messages);
pragma Unreferenced(CubedOS.File_Server.Messages);
pragma Unreferenced(CubedOS.Publish_Subscribe_Server.Messages);
pragma Unreferenced(Controller.Messages);

procedure Main is
begin
   null;
end Main;
```

Although Main returns immediately, the process won't terminate until all the module tasks have terminated. However, the module tasks are all infinite loops (as required by the Jorvik profile) so the program never ends. This is common for embedded systems. The control program runs for as long as the machine it is controlling runs.

The **with** statements bring the module tasks into the executable. The Ada compiler will form the transitive closure of package dependencies, locating all the necessary program components, based on these **with** statements and on **with** statements in the included packages, etc. Since no module directly interacts with the message loop of any other module, it is essential that you **with** the Messages packages here. Otherwise, a module's message loop will not be linked into your executable and that module's mailbox will never be serviced.

The **pragma** Unreferenced directives suppress warnings about unused **with** statements, but have no other effect.

Notice that we are including all the core modules we will use, but only the Controller skeleton module we made previously. We will add the other application-specific modules as we create them, editing Main to include them when we are ready.

At this point, the program should compile and run. When executed, all modules will block immediately trying to fetch a message from their mailboxes. Since no messages are ever sent, nothing more happens. We're off to a good start![4]

3.1.7 Using Mercury

If you spent much time studying the template API module, you probably noticed that it is intricate. Each encoding and decoding subprogram involves repeated calls to the XDR library to encode (or decode) individual message parameters into (or from) an array of raw octets. This is tedious and error-prone.

[4]It didn't crash, did it?

To help with this, CubedOS includes a tool called Mercury that generates the encoding and decoding subprograms for you. Mercury reads a simple, human-readable description of the messages sent and received by a module, and it outputs a SPARK-provable API package for that module.

Mercury is a work in progress, and it may not be able to handle all your API needs automatically. However, even when the API package it produces is incomplete or insufficient, you can still use Mercury as a starting point. It is much easier to make adjustments to the generated code than to write the entire API package from scratch.

Like all message-passing systems, CubedOS is inherently distributed. In principle, messages can be passed from one module to another in the same process, in different processes on the same computer, or even on different computers connected by a network. From the point of view of one module, all other modules are "remote" code. Multi-domain CubedOS applications capitalize on this inherently distributed nature to allow applications to be spread across a network of spacecraft in a natural way.

Many distributed application environments exist: COBRA, ONC RPC, SOAP, Ice, and gRPC, to name only a few. Essentially all of these environments use some *interface definition language* (IDL) to describe the interface of remote code. That IDL is then compiled into client-side code (often called "stubs") that *marshal* the parameters of a remote procedure call into a message that can be sent over a network, and server-side code (often called "skeletons") that *unmarshal* the message back into parameters that can be passed to the actual remote code. These stubs and skeletons, together with the message-passing infrastructure connecting client and server form the *middleware* layer of the application.

In CubedOS terms, the middleware layer is the message manager together with the API packages generated by Mercury. Marshaling is done by the encoding functions generated by Mercury, and unmarshaling is done by the decoding functions generated by Mercury. Because the API packages are part of the middleware layer provided by the system, it is unreasonable and inappropriate for application developers to be required to write them manually. Thus, Mercury is not just a convenience; it is an essential part of CubedOS's architecture.

CubedOS currently uses XDR (eXternal Data Representation) [8] as its encoding format, but it could just as easily use some other approach such as ASN.1, Protocol Buffers, or even JSON. As a CubedOS application developer, switching to a different encoding format should be as simple as setting an option on Mercury's command line. You should not be required to manually rewrite the marshaling and unmarshaling code in your application (i.e., the API packages).

MXDR

The interface definition language used by Mercury is an extension of XDR [8] called MXDR (Modified XDR). MXDR includes features for specifying constraints on basic

types, and for specifying messages and their direction of information flow. We will illustrate MXDR using the Moonshot application.

Consider, for example, the Camera module. For simplicity, we will assume that it only knows about two messages: a request to take an image, and a reply containing the file name of where it stored the image in the file system. It would be reasonable for the Camera module to send the image data directly, but in this case it replies with what is, in effect, a pointer to where that data can be found. Here is the MXDR description of these two messages:

```
message struct -> Take_Image_Request {
    void;
};

message struct <- Take_Image_Reply {
    string file_name<128>;
};
```

The core of an MXDR description are the message definitions. Messages are conceptualized as structures with named fields for the message parameters. The direction of the arrow indicates the direction of information flow. An arrow pointing to the right is for messages that go from a client to the module being described. An arrow pointing to the left is for messages that go from the module being described to the client. Thus, an MXDR description is always from the point of view of the module being described as playing the role of a server. When a module acts as a client, it will use the API package of the other modules that services its requests.

In this case the Camera module provides an operation "Take Image" and so accepts a message Take_Image_Request and replies with a message Take_Image_Reply. The names of the messages are arbitrary, but the use of "Request" and "Reply" is conventional. The names do not necessarily need to be related, i.e., the reply message could be named in some other way if desired.

In this simple example, the request messages have no parameters. In a more realistic example, it is likely that additional parameters would be needed to specify exactly how the camera should capture the image. Those additional parameters would be fields in the message structure. The special field value void is used here to explicitly indicate that there are no other parameters. It is required to be used in such cases.

The reply message contains a single parameter, a string of up to 128 characters. The string type maps to the usual Ada string type, but in this case, there will be an additional decoded value to indicate how much of the string is actually used.

When Mercury compiles this MXDR description, it will generate the following six subprograms:

```
function  Take_Image_Request_Encode(...) return Message_Record;
function  Is_Take_Image_Request(...) return Boolean;
procedure Take_Image_Request_Decode(...);

function  Take_Image_Reply_Encode(...) return Message_Record;
function  Is_Take_Image_Reply(...) return Boolean;
procedure Take_Image_Reply_Decode(...);
```

For clarity, the details of the subprogram parameters are left out. However, all CubedOS messages include a Request_ID parameter that the sender can use to mark messages with An identifier meaningful to that sender. This is useful for correlating requests and replies.

For example, the sender might command the camera to take several images, perhaps using different parameters (in general, but not shown here). The sender can mark each request with a unique Request_ID. When the Camera module replies, it will copy the sender's Request_ID into the reply. The sender can then use that information to match the reply with a particular request. If there is no need for a sender to correlate requests and replies, the Request_ID can be set to zero and ignored.

Keep in mind that the sender might send multiple requests before receiving any replies, allowing it to tend to other matters while the camera acquires the images and writes them to storage. This highly asynchronous behavior is typical of CubedOS applications.

Mercury uses the name of the MXDR file to construct the name of the generated Ada package. In this case, the MXDR file should be named `Camera.mxdr`, with the indicated casing. Mercury will create the package Camera.API in the files `camera-api.ads` and `camera-api.adb`, as required by the GNAT Ada compiler.

To understand how to use a module, you only need to read the MXDR for that module, resorting to the generated API package only when you need to know the exact subprogram signatures. For example, if the Controller module wants to access the image file created by the Camera module, it will need to send messages to the file server module. Here is a part of the file server's MXDR:

```
enum Mode_Type { Read, Write };

// The special value 0 is used for invalid files.
typedef unsigned int File_Handle_Type range 0 .. 64;
typedef File_Handle_Type Valid_File_Handle_Type range 1 .. File_Handle_Type'Last;

// Types to count the number of octets read/written.
const Maximum_Read_Size  = 256;
typedef unsigned int Read_Result_Size_Type range 0 .. Maximum_Read_Size;
typedef unsigned int Read_Size_Type range 1 .. Read_Result_Size_Type'Last;
```

```
// Open the named file in the given mode.
message struct -> Open_Request {
    Mode_Type Mode;
    string    Name<>;
};

// Returns the invalid handle if the open operation failed.
message struct <- Open_Reply {
    File_Handle_Type Handle;
};

// Request that Amount octets be read from the indicated file.
message struct -> Read_Request {
    Valid_File_Handle_Type Handle;
    Read_Size_Type         Amount;
};

// Returns data from the file.
message struct <- Read_Reply {
  Valid_File_Handle_Type Handle;
  Read_Result_Size_TYpe  Amount;
  opaque                 File_Data<>;
};
```

The protocol speaks for itself. First a module must send an Open_Request message to the file server, specifying the desired file name. The server responds with an Open_Reply message containing a "handle" to the file (or zero if the open failed). The module then sends repeated Read_Request messages to the file server specifying the previously opened handle. The server responds with a Read_Reply messages containing the requested data.

The full interface to the file server also includes messages for writing files, closing files, and some other operations. The file server implementation provided with CubedOS uses the file system on your development system. In a real mission, the implementation (i.e., the Messages package) could be replaced with a version that works with whatever storage is available on the spacecraft without changing the interface or the modules that rely on that interface. In this way, CubedOS supports a limited object-oriented-like programming style.

Notice how various types are defined before the message structures can be declared. Mercury will generate corresponding Ada type definitions into the API package accordingly. As an extension to XDR, Mercury allows you to specify constrained ranges on the types, following the usual Ada style. This is essential. Mercury attempts to generate SPARK-provable code, and SPARK's proofs are greatly facilitated by the use of constrained types. This feature sets Mercury apart from most other IDL compilers.

Invoking Mercury

Mercury is included in the CubedOS repository in the `mercury` folder. It must be built before it can be used. Fortunately, this is easy to do. However, since Mercury is written in Scala, it may require that you install some additional tools. Specifically:

1. You must install a Java runtime. Mercury requires Java 21 or later. Download Java from `https://www.oracle.com/java/technologies/downloads/`.
2. You must also install SBT. Download SBT from `https://www.scala-sbt.org/`. Any recent version of SBT will be fine. It fetches and caches the precise version required by a project when you first build that project. SBT is a build tool for Scala, similar to Maven or Gradle for Java projects.

Significantly, you do not need to explicitly install Scala. SBT will fetch the correct version of Scala for you, along with any other required libraries and plug-ins, when you build Mercury for the first time.

Next, open a command prompt or terminal session, and change to the `mercury` folder of the repository. Execute the command `sbt assembly`. This will build Mercury and create a standalone executable JAR file containing the tool. Although not required, you might also want to use the command `sbt test` to run the test suite for Mercury.

There is a shell script named `mercury.sh` in the `bin` folder off the root of the repository. This script is a convenience wrapper around the Java command to run Mercury. A symbolic link to this wrapper script can be created wherever you find convenient.

Mercury requires template files that it uses during code generation. These templates are located in the `mercury/templates` folder of the repository. The `mercury.sh` script specifies this location to Mercury as a command line option. Unless you want to use different templates (which you almost never will), you do not need to worry about this.

Once Mercury is ready, apply the shell script wrapper to your MXDR file to generate the API packages as follows:

```
$ mercury.sh Camera.mxdr
```

Note that this only produces `camera-api.ads` and `camera-api.adb`. It does not produce the top level package itself (`camera.ads` in this case). When configuring the build of your application, you must arrange for Mercury to be run on the MXDR files, if necessary, before the Ada compiler is invoked.

Mercury is intended to produce files that can be provided free of runtime errors with SPARK. Thus, you can (and should) use the SPARK tools to verify the generated code along with your handwritten code. If a bug in Mercury causes it to generate buggy code, you will want to give SPARK an opportunity to find that error!

Also, the files produced by Mercury are intended to be human-readable and human-editable. Ideally you would not need to edit them, but it might be necessary in some cases. Those cases could reasonably be described as exhibiting a deficiency in Mercury, but the tool is still useful even when it cannot completely automate the generation of the API packages in all situations.

There is much more that can be said about MXDR and using Mercury. We will describe additional features as we encounter them in the Moonshot application. A complete tutorial for Mercury is in Section 3.1.7.

3.1.8 Moonshot MXDR

Now we are ready to write the MXDR for the application-specific modules in the Moonshot application. In a real application we would likely have detailed requirements to help guide us in this task. However, in this tutorial we will use highly simplified interfaces since we are only trying to illustrate the application development process.

Camera Module

The MXDR for the Camera module presented in Section 3.1.7 is good enough for our purposes. We repeat it here for convenience:

```
// FILE: Camera.mxdr

message struct -> Take_Image_Request {
    void;
};

message struct <- Take_Image_Reply {
    string file_name<128>;
};
```

Radio Module

Finish Me!

Thruster Module

Finish Me!

Controller Module

The MXDR for the Controller module is unusual in that there is no MXDR file required! The Controller module plays the role of the "main" program of the application. It coordinates the activity of the rest of the system. As a result, it only acts like a client to the other modules and provides no services that the other modules will use. It accepts no request messages, and has no corresponding replies. It therefor has no MXDR interface definition and no API package.

3.1.9 Writing the Application

Now that we have the skeletons of the modules and the MXDR descriptions of the messages, we can begin to write the application logic. Let's build this application from the bottom up, starting with the Camera module.

Camera Module

We have only one message to implement, Take_Image_Request. In this tutorial we will always return one of two hard-coded images. The file `Image-Copernicus.jpg` is a picture of the Copernicus crater [1], and the file `Image-Korolev.jpg` is a picture of the Korolev crater [3] (on the far side of the Moon).

In a real application this module would need to control the camera hardware, capture an image, and store it in a file. For demonstration, however, we will just alternate between the two images when a request is received, sending first the Copernicus image and then the Korolev image.

Once we decide what image to return, we simply send a Take_Image_Reply message to the sender using the Route_Message procedure provided by the CubedOS message manager. Normally, we would need to decode the Take_Image_Request message to extract any desired parameters from it. However, in this application the message is empty so no decoding is required. However, we will still need to encode the reply using a suitable encoder from the API package generated by Mercury.

Radio Module

Finish Me!

Thruster Module

Finish Me!

Controller Module

The Controller plays the role of the application's main program. It performs some initialization duties and then coordinates the behavior of the rest of the system.

Since CubedOS applications are reactive[5], we will describe the behavior of the Controller module, not in procedural terms, but in terms of how it responds to events (i.e., messages).

In a typical application, interrupts drive the action of the system. When a hardware device creates an interrupt, the module that serves as that device's driver will service the interrupt and, in general, send messages to other modules to take additional action. Those modules will, in turn, send messages to still other modules, resulting in a flurry of activity across the system. Once the interrupt, and all its consequences have been fully processed, system activity dies down again, and all modules are left blocked on their mailboxes waiting for the next message.

In this tutorial application, there is no physical hardware to create interrupts. Instead, we will use the timeserver to generate periodic "tick" messages to kick-start Controller activity. This is a normal use-case for the timeserver. In a real application, the tick messages can be used to trigger periodic housekeeping activities, backups, or (slow) polling of hardware components.

During initialization, before entering the main message fetching loop, we do:

- Configure a *periodic series* with the time server, requesting tick messages every minute. We will need to specify a *series ID* for this series. Any value will suffice (it is up to us), but we should remember it. In general, an application might have multiple time series active at once, and we will want to distinguish one from the next using their unique series ID values.

In this simple application, there are no other initialization activities.

Table 3.1 shows the Controller's responses to various messages.

There is an implicit assumption in Table 3.1 that the Controller will be able to completely handle one image file before the time server says it's time to ask the Camera for another. In this simple tutorial application, where the Camera can acquire an image "instantly," that assumption is reasonable. In a more realistic application, it might be necessary to plan for the possibility of there being multiple active images at once. In that case, the Controller could store pending image file names in a data structure that is global to the module. When one file is closed, the Controller could request that the next be opened, thus serializing what would otherwise be interleaved operations.

CubedOS's message passing paradigm would also allow interleaved file reads by using different request IDs for the different files. When the module receives some file data, it

[5]They react to external events, but otherwise don't initiate any activity.

Table 3.1: Controller Message Actions

Message	**Action**
Tick message from the time server.	The time server is trying to wake us up, so we can do something. As the Camera to take a picture.
Camera replies with image file name.	Ordinarily the Camera might alternatively reply with an error, but in this simple application that will never happen. Send a message to the file server to open the image file.
File server replies to open request.	If the open request failed, log the error. Otherwise, prepare a space packet with a data size of 512 octets and APID of 42 (an arbitrary value selected by us). Ask the file server to read the first 256 octets of file data.
File server replies with data.	Pack the data received into the currently active space packet. If the packet is full, send it to the Radio module for transmission to Earth and prepare a new space packet. If 256 octets received, ask the file server for the next 256 octets, otherwise ask the file server to close the file.
File server replies to close.	If the currently active space packet is partially filled, send it to the Radio module for transmission.

could consult the request ID on the received message to determine to which data stream that data belongs. Of course, the Radio module would have to also be prepared to accept interleaved file transfers. These are some of the design questions that arise when building CubedOS applications.

In this tutorial application, the Controller repackages file data into *space packets* as defined by the Consultative Committee on Space Data Systems (CCSDS) ⟨⟨⟨ Citation needed ⟩⟩⟩ . It is the space packets that are sent to the radio, encapsulated in CubedOS messages. The CCSDS is clear that space packets can be used for intermodule communication, as well as for spacecraft-to-spacecraft and spacecraft-to-Earth communication. The CubedOS library contains a package, CubedOS.Lib.Space_Packets, that facilitates the construction and manipulation of space packets. PET

The header of a space packet contains an *application ID* (APID), that applications can use to identify the semantic significance of the packet's contents. In our case, we arbitrarily use an APID of 42 for file data. This choice would need to be known to ground software so that it can distinguish file data from other kinds of telemetry.

Note also, that space packets contain a flag that can be used to distinguish the first,

last, and continuation packets of a data stream that spans multiple packets. We will use that flag to ensure that ground software understands that multiple packets are used to hold the data from a single file.

It would be reasonable for the Controller to not dirty itself with low-level activities like reading files. Instead, it could pass the name of the file to the Radio and let that other module deal with the details of getting the file data. However, a reusable Radio module would mostly likely only want to work with space packets, leading to the approach here.

Actually, in a real mission, it is most likely that an entirely separate module implementing the CCDS File Deliver Protocol (CFDP) would take care of reading the file and building CFDP protocol units to be encapsulated in space packets. The Controller would just pass the name of the image file to the CFDP module for transfer.

3.2 Core Modules

This section describes the core modules that are part of the CubedOS distribution. These modules are intended to be used by most CubedOS applications. They provide basic services. Additional information about a core module can be found in the comments in the module's top level package and API package specifications, or in the MXDR file used by Mercury to generate the modules API package.

3.2.1 Log Server

The log server is a core module that provides a simple logging facility. The default log server outputs log messages to the console via Ada's standard library package Ada.Text_IO. For systems where that package is not available, developers can write their own body for package CubedOS.Log_Server.Messages that handles incoming messages in whatever way makes sense for their application.

Log messages are intended to be short strings of human-readable text. Each message has an associated log level, described below, and is automatically timestamped by the log server with a boot-relative time in seconds.

The log server's API package provides a convenience procedure named Log_Message that takes care of the details of constructing a message and sending it to the log server. It is expected that most uses of the Log Server will be by way of calling Log_Message. Currently, the Log Server does not send any reply messages. Logging is assumed to always be successful. This is an area for future improvement.

Of particular interest is the Log_Level parameter to Log_Message. This parameter is used to indicate the severity of the log message. It is an enumeration type. The Log levels are intended to follow those used by the Unix syslog facility, both in name and meaning. The following log levels are defined:

- Debug: Messages that are only of interest to developers. These messages are typically very detailed and are used to diagnose problems in the system. Debug messages are not typically enabled in a production system.
- Informational : Messages that are of interest to users. These messages are typically high-level and are used to inform the user of the system's normal operation.
- Notice: Like informational messages, notice messages occur during normal system operation. However, they log particularly significant events that are meant to stand out compared to ordinary informational messages.
- Warning: Messages that indicate a potential problem. The system is still functioning correctly, but the message indicates that something might be wrong.
- Error: Messages that indicate that a problem has occurred. The system is still functioning, but the message indicates that something unintended or unexpected has happened.
- Critical : Critical messages are similar to error messages, except with more severity and criticality. A critical message indicates that the system is in a bad state and that the problem should be addressed soon if the system is to continue functioning normally.
- Alert: Messages that indicate a problem that has occurred that is so severe that the system cannot function normally without corrective action. The system will likely soon fail completely.
- Emergency: Emergency messages are the most severe. They indicate that the system has failed, is unusable, or soon will be, and that immediate action is required.

The timestamps added to every log message show the time the message was processed by the log server relative to when the system started. Timestamps are in seconds. This time includes the overhead of passing the message from the sending module to the log server. Because multiple messages might be queued in the log server's mailbox, and because of task priority issues, the timestamp might indicate a time that is, potentially, significantly later than when the message was generated by the sending module.

One area for future work in the log server would be for the sending module to generate a timestamp immediately before when the message is sent and pass that timestamp to the log server as part of the message. In that case, the timestamps would be a more accurate reflection of when the message was generated, being free from queuing and task priority effects. The convenience procedure Log_Message could take care of the timestamp management details, so the developer wouldn't need to do any extra work.

Of course, many applications won't care about such precise timestamps. The current implementation will likely be sufficient for those applications. However, applications with very tight timing requirements might benefit from better debugging and monitoring that would be made possible by more accurate timestamps.

An implementation that displayed timestamps using an absolute time was considered. However, such an implementation would need access to a real-time clock via, for example, package Ada.Calendar. While Ada.Calendar is supported under the Jorvik profile, some small embedded environments nevertheless do not support it. To keep CubedOS as widely applicable as possible, it was decided to avoid the use of Ada.Calendar and instead focus on only boot-relative timestamps.

Other future enhancements to the log server include (but are not limited to) the following:

- There should be a way to specify the log level that is displayed. Under normal operation, users will not want to see Debug messages, for example. Furthermore, because such messages might be plentiful and could potentially consume significant resources to transmit in volume, the filtering of those messages should be done in Log_Message and not by the log server itself. That would avoid transmitting messages that are of no interest.
- As an extension of the idea above, it should be possible to control the log level based on the module ID of the sending module. The use-case for this ability is when the developer would like to log Debug messages from a particular module being debugged without having to see all Debug messages generated from all other modules too.

3.3 Samples

This chapter contains documentation about the sample programs provided with CubedOS. These programs illustrate various aspects of using CubedOS in the real world that might not be covered by the more formal documentation presented so far. New users of CubedOS are encouraged to study, build, execute, and modify the sample programs as a way of learning about the system and how it can be used.

3.3.1 Echo Sample

This section describes the Echo Sample that can be found in the `samples/echo` folder. This sample exemplifies a simple message/reply demo. These messages take place as simple pings back and forth between a Client module and a Server module.

The main.adb of this program has several with clauses to include Echo_Client.Messages and Echo_Server.Messages. It also uses Log_Server.Messages in order for the modules to be capable of logging various errors or informational output. This main file contains a single procedure of first priority with a single loop that has a small delay. This loop spends most of its time sleeping, and the real work is done inside the Client and Server modules which are unreferenced in the main itself.

Both Echo_Client.ads and Echo_Server.ads have with clauses for Message_Manager. These are of course unreferenced, but are very important. The package Message_Manager creates a new instance with a domain number, module count, mailbox size, and maximum message size. This sets the environment for all future message sending. This file also elaborates CubedOS.Generic_Message_Manager, which contains all necessary functions and procedures for managing messages.

Echo_Client.Messages.adb implements the main part of its module. At its core, this component sends an initial request message to the server, and continues to send messages each time it gets a reply back. There is a single task called Message_Loop in the body. This task first runs the initialize procedure, and then loops while waiting for messages. Inside the initialize procedure is where the first message gets sent. This procedure uses the Echo_Server.API to encode a ping request message. The sender address is set to Name_Resolver.Echo_Client. The Name_Resolver.ads file contains the message addresses for the Client, Server, and Log Server. After initializing, the task starts looping. In each loop, it waits for a message. Once it receives a message, it processes that message and sends another.

Echo_Server.Messages.adb Has a similar Message_Loop. This task does not send an initial message, but instead, has just a loop. The loop waits for a message right away. When one is received, it processes it and sends a reply message using Echo_Server.API. Then, it starts the Message loop over again.

This goes on infinitely. The log server logs information every time a reply is received by the Client. It also logs errors when a message goes wrong. This is to show the user the basic functionality of the log server, and how it can be used with the message manager.

3.3.2 LineRider Sample

Finish Me!

3.3.3 Pathfinder Sample

This sample is intended to demonstrate the problem of priority inversion in CubedOS. It is loosely based on Mars Pathfinder, which had an actual issue with priority inversion during the mission.

Section 2.2.2 describes the priority architecture of CubedOS. Briefly, every module has a statically assigned priority based on the priority given to the message loop that fetches and processes messages. There is currently no facility to change module (task) priorities once they are assigned. Priority inversion can happen in the following circumstances:

- A low priority module begins processing a message.

- A medium priority module becomes ready to run, and the runtime system suspends the low priority module in favor of the medium priority one.
- A high priority module becomes ready to run, and the runtime system suspends the medium priority module in favor of the high priority one.
- The high priority module sends a message to the low priority module. This message requires a reply, so the high priority module waits for that reply to arrive by blocking on fetching from its mailbox.
- The runtime system resumes the medium priority module which, unfortunately in this hypothetical case, runs for an indefinitely long time.

In this scenario, the low priority module is never given a chance to finish what it started, never sees the message from the high priority module, and never replies to that message. The high priority module is now waiting, in effect, on the medium priority module to finish, since that is what is holding up the low priority module from making progress. Yet high priority tasks should not be held up by the execution of any task with lower priority. This is priority inversion. PET

⟨⟨⟨ Finish this! More needs to be said about the sample and how to interpret its output. It would also be good to include some citation(s) regarding the actual Pathfinder issue. ⟩⟩⟩

3.3.4 STM32F4 Sample

This section describes the architecture of the STM32F4 sample that can be found in a folder of the same name in the CubedOS repository. This sample makes use of a simple evaluation board from STMicroelectronics. The README file in the STM32F4 folder gives specifics about the board and about how to set up the development environment for it.

The sample program is simple in effect, but written in a way as to illustrate some of CubedOS's structure and features. The program flashes the LEDs on the board in a rotating sequence, from green to orange to red to blue. The program also watches the user button on the board (the blue button). When pressed, the speed of the flashing increases in steps, from low to medium to high. The program has no other features.

This program illustrates several things.

- Using CubedOS with a non-native CPU. The sample was developed on a Linux desktop system running a 64 bit Intel processor. The project, however, is configured to generate code for an ARM Cortex M4 CPU. This affects not only the compiler being used, but also the SPARK configuration used to analyze the code.
- Using a small, bare board embedded platform. The sample makes use of AdaCore's light tasking runtime for the STM32F4 along with a few files provided by AdaCore for accessing the hardware on the board. It does not make use of any conventional operating system.

- Using CubedOS modules as drivers for onboard hardware. Although this simple application could be more easily programmed without creating driver modules, for purposes of illustration it includes two driver modules for the hardware used: an LED driver and a button driver.
- Using a CubedOS "controller" module as the main module of the program. The controller module responds to events generated on the board by sending messages to other modules, as appropriate. The controller serves as the main part of the application and encodes the high level application logic. The Ada main procedure contains no application functionality at all. This is a typical for CubedOS applications.
- Using the publish-subscribe server, a core CubedOS module, for decoupling event sources (as generated by the button driver) to event sinks (the controller module). This allows the driver modules to be reusable in other applications for this platform. Specifically the button driver does not directly interact with the application-specific controller. It could be dropped into another application, along with the core publish-subscribe server module, without modification.

In the subsections that follow more details are given about each of these points.

Non-Native CPU

The STM32F4 sample uses an ARM Cortex M4 processor. This entails installing the necessary Ada cross-compiler from AdaCore. The project file specifies the tool chain to be used. The other important aspect of using a non-native CPU is configuring SPARK to know about the target platform attributes. Since the sizes and ranges of the primitive types are often different on the embedded platform than on the native platform, the sizes and ranges of the Ada base types are also often different. This affects when computations might overflow. Consider, for example, this simple case:

```
declare
   type Some_Type is range 0 .. 20_000;
   A, B, C : Some_Type;
begin
   -- etc...
   A := (B + C) / 2;
end;
```

On a system where the base type of Some_Type has 32 bits, there is no possibility of overflow when computing (B + C)/2 since, at most, B + C would be 40,000. However, if a 16 bit base type is used, as might be the case on a small microcontroller platform, B + C might go outside the range of 16-bit signed integers, and an overflow would be possible. The SPARK tools know what base types are used by the Ada compiler, but that depends on the platform being targeted by that compiler. It is thus essential to inform the SPARK tools about the target platform so that the proofs are relevant and correct

for the target context. By default, the SPARK tools will assume the native platform's properties, which might not be appropriate when cross compiling.

Finish Me!

4 Mercury

Mercury is an interface definition language (IDL) compiler for CubedOS. It takes as input a file describing the messages a CubedOS module receives (requests) or sends (replies), and outputs an Ada package that defines the API to that module. The package provides subprograms for encoding messages, decoding messages, and classifying messages.

One unique aspect of Mercury is that it generates SPARK code that can be proved free of runtime error without any further intervention by the user.

The IDL used by Mercury is an extension of XDR as defined in RFC-4506 [8], that we call modified XDR or MXDR. The modifications incorporate features that facilitate SPARK's ability to prove the absence of runtime errors.

Mercury is written in Scala and is invoked from the command line.

In this chapter we describe the requirements, design, and usage of Mercury.

4.1 Requirements

In this section we describe the requirements of Mercury.

The developer specifies the interface to a module using MXDR (described later). The following requirements apply:

- **Mercury.IDL**. Mercury shall take as input a file describing the messages of a CubedOS module in *Modified XDR* (MXDR) format.
- **Mercury.Output**. Mercury shall produce an Ada package that defines the API to the module. The package shall provide subprograms for encoding messages, decoding messages, and classifying messages.
- **Mercury.Provable**. Mercury shall generate SPARK code that can be proved free of runtime error without any further intervention by the user.

4.1.1 Competitive Analysis

The Core Flight Executive (cFE [2]) passes space packets between modules with no special internal encoding/decoding procedures for separate 'proprietary messages.'

Most primary competitors to Mercury, such as ONC RPC, Ice, Java RMI, and WSDL, take a formal language (generally a data representation language) and generate *stubs* that package message parameters into message bodies on the client side, and generate *skeletons* that decompose messages on the server side back into the original parameters.

- **ONC RPC** is similar to Mercury in that it uses an IDL that is compiled into XDR encoding stubs and XDR decoding skeletons. However, ONC RPC operates in a more complex environment and contains many more features that are not relevant in the case of CubedOS.
- **Ice** uses a proprietary language called Slice which can then be compiled to a variety of languages including C++, Java, C#, Python, Ruby, and PHP. It uses traditional network protocols like TCP/IP and UDP/IP as the underlying transport.
- **Java RMI** is like Ice, but only uses Java and the JVM for two-way data transfers.
- **WSDL** is like Java RMI, but uses WSDL (Web Services Description Language).

4.1.2 Base Requirements

The following requirements apply:

- **Mercury.Robustness**. Mercury must be robust enough to handle all conceivable forms of error in the MXDL file and output a clear and correct message describing each error.
- **Mercury.Simplicity**. Mercury must be simple enough that a reasonable (and even novice) user could successfully insert encoding and decoding subprograms into their CubedOS project with minor to no difficulty.
- **Mercury.XDR**. Mercury shall implement the whole of the XDR specification [8].

4.1.3 XDR Extensions

We refer you to the XDR specification [8] for the full details of the XDR encoding that Mercury shall support in accordance with the **Mercury.XDR** requirement. In this section, we focus on the extensions to XDR that Mercury shall support.

Describe the following: constrained typedefs, message structs, message aspects (e.g., message invariants).

4.1.4 Future Work

In the future, Mercury could support other encoding schemes besides XDR (for example, ASN.1, to name just one possibility). Also, a future version of Mercury could generate

API packages that use standard network protocols, such as TCP/IP (or some other), encoding messages into, for example, a UDP datagram.

4.2 Design

In this section we describe the design of Mercury. This section will be primarily of interest to developers who are working on the implementation of Mercury. Developers who are using Mercury to generate API packages for CubedOS modules should refer to the tutorial in Section 4.3.

4.2.1 Mercury Pipeline

- **Scala**. Mercury is written in Scala, a functional/OOP hybrid language for the Java Virtual Machine (JVM). Mercury capitalizes on the Scala/Java ecosystem in terms of tooling (such as SBT) and library support.
- **Grammar**. MXDL is described as an ANTLR4 grammar that is then processed by ANTLR to generate the lexical analyzer and parser (in Java).
- **Lexer**. Mercury's lexical analyzer is ANTLR generated. It is fairly standard in terms of its handling of whitespace, identifiers, numeric literals, etc.
- **Parser**. The parser is also ANTLR generated, and Mercury uses ANTLR generated classes for traversing the parse tree produced by the parser. Syntax errors are detected by the ANTLR generated code.
- **Semantic Analyzer**. After parsing, the semantic analyzer walks over the parse tree searching for type errors and other violations of MXDR's rules. After this stage of processing, the input MXDR is considered valid and code can be generated from it.
- **Code Generation**. The code generator takes the validated MXDR and generates the Ada package that defines the API to the module. Two files are produced: the package specification and the package body. Code generation entails walking over the parse tree again and emitting Ada code based on the nodes encountered.

Since approximately half of the files in this Mercury compiler pipeline are auto-generated by the ANTLR tool, the three distinct areas of focus at the code level are the grammar, the semantic analyzer, and the code generation. SPARK skeletons are filled in by the code generation phase after the previous compiler phases check the soundness and validity of the modified XDR.

4.3 Tutorial

Mercury is a compiler that takes the definition of a CubedOS module interface, written in an interface definition language (IDL), and outputs an API package for that module in SPARK. The intention is for the generated code to be provably free from runtime errors without any further intervention on the part of the developer. Thus, Mercury greatly simplifies the work involved in creating API packages for CubedOS modules.

The IDL used as input to Mercury is an extension of the XDR language described in RFC-4506. The extended features allow expression of constraints on scalar types, as well as some additional features pertaining to the SPARK target language. We call this extended version of XDR "modified XDR" or MXDR. The central abstraction in MXDR is that of the message. Rather than declaring remote procedures directly, or specifying classes with methods as is done in most OOP-based IDLs, MXDR allows you to describe messages by the information they contain. Each message implies three subprograms:

1. A function that takes the message contents as typed **in** parameters and encodes the message into a raw octet array for transmission.
2. A procedure that decodes the message as a raw octet array back into typed **out** parameters.
3. A function that verifies that a given octet array contains an instance of a particular message.

The MXDR user only need to specify the structure of the message. Mercury generates the three associated subprograms without any additional assistance. This message-centric focus is how MXDR differs from most interface definition languages, and is one of the main reasons we developed an entirely new tool for CubedOS.

We choose to base Mercury on XDR to simplify interoperability with other systems in the future. At the time of this writing, CubedOS only supports messages between modules in the same operating system process, all of which are written in SPARK. However, we anticipate extending the architecture to support message passing between processes, and even between machines. Using a standard wire protocol, such as XDR, may simplify that work. All the extensions to XDR that we define are at the IDL source level only, and are used to guide the generation of provable SPARK. We specify no changes to the underlying XDR format.

This section of the Mercury documentation is a tutorial for Mercury users. When you are contemplating a new CubedOS module, you should first express the API to your module as a collection of messages it sends and receives. Write an MXDR file that declares the structure of those messages, and then use Mercury to generate the API package. You can then focus the bulk of your programming effort on the part of the module that processes the messages you defined without wasting time writing tedious and highly repetitive message encoding and decoding subprograms.

Bibliography

[1] Copernicus image. https://en.wikipedia.org/wiki/Copernicus_(lunar_crater).

[2] Core flight executive. http://opensource.gsfc.nasa.gov/projects/cfe/. Accessed 2017-01-22.

[3] Korolev image. https://en.wikipedia.org/wiki/Korolev_(lunar_crater).

[4] Kubos. http://www.kubos.co/. Accessed 2017-01-22.

[5] Real time executive for multiprocessor systems. https://www.rtems.org/. Accessed 2017-01-22.

[6] Vxworks. http://www.windriver.com/products/vxworks/. Accessed 2017-01-22.

[7] Wind river vxworks cert platform. http://www.windriver.com/products/product-overviews/vxworks-cert-product-overview/. Accessed 2017-01-22.

[8] M. Eisler. *RFC-4506: XDR: External Data Representation Standard*. Internet Engineering Task Force, May 2006. http://tools.ietf.org/html/rfc4506.html.

[9] ITU. *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*. International Telecommunications Union, November 2008. http://www.itu.int/rec/T-REC-X.680/en.

[10] R. Thurlow. *RFC-5531: RPC: Remote Procedure Call Protocol Specification Version 2*. Internet Engineering Task Force, May 2009. https://tools.ietf.org/html/rfc5531.