J. Beningo, *Embedded Software Design*
https://doi.org/10.1007/978-1-4842-8279-3_14

14. Comms, Command Processing, and Telemetry Techniques

Jacob Beningo[1]
(1) Linden, MI, USA

While there is no such thing as an absolute in embedded systems or life, most embedded systems require the ability

- To communicate with the external world
- To receive and process commands
- To report the system's status through telemetry

To some degree, these are necessities of every system. However, the complexity of each can vary dramatically. For example, some systems only have a simple USART communication interface that is not encrypted and is wrapped in a packet structure. Other systems may communicate over Bluetooth Mesh, require complex encryption schemes, or be encapsulated in several security layers requiring key exchanges and authentication.

This chapter will examine how to build reusable firmware for simple communications and command processing. We will start with developing a packetized communication structure that can be used to send commands to a system. We will then explore how to decode those commands and act on them. Finally, we'll explore how to build reusable telemetry schemes that can be used in real-time applications. The techniques discussed in this chapter will use more straightforward examples that you can then extrapolate into more complex real-world scenarios.

Designing a Lightweight Communication Protocol

There are several problems that developers often face when it comes to sending and receiving data from their embedded systems. First, they need

some mechanism for retrieving the data from their system, whether it's health and wellness data or sensor data. Second, they need some mechanism for commanding their system and any data that tells the system how to carry out those commands. Finally, data transfer to and from an embedded system can take many forms and potential physical interfaces.

A common theme between many physical layer interfaces is that it is up to the developer to design and build their own higher-level application protocol that sits on top of the physical layer. For example, in many instances, a system might have a USART, USB, or Wi-Fi interface that can send/receive data. Still, the developer must decide on the protocol to encode and decode the payload sent over the interface. Even if a standard protocol is selected like CAN or MQTT, these protocols include a payload area where it is up to the developer to define their application data transfer protocol. That protocol defines what all the bits and bytes mean to the application. The need for a higher-level application-specific protocol makes sense because there are a wide variety of systems and requirements, and there is no one solution to fit them all.

The use of communication packets can be an extremely effective method for developers to define their application-specific transfer mechanism because it provides a set of predefined fields that the sender or receiver can decode, validate, and process. In addition, the packet does not have to be a fixed length and can adapt to the system's needs on the fly. A packet can also be assembled and decoded using a state machine that can easily handle real-time streaming and be reset if something does not go as expected. Figure 14-1 shows the general process of packet data entering a state machine and then being converted into a format usable by the application code.

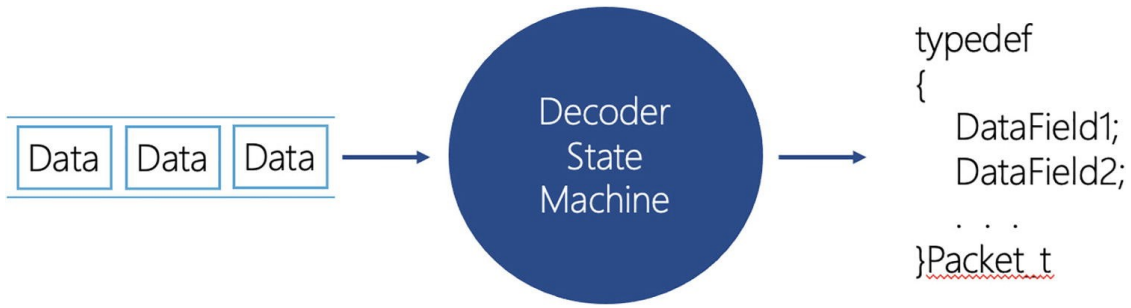

Figure 14-1 A protocol decoder is often implemented as a state machine that consumes input data, transitions to various decoding states, and then, if successful, results in a populated data structure that the application can use

The exciting thing about packet protocols is that if you design and use enough of them, you'll discover that there are several requirements that nearly every packet protocol needs despite the industry of the system that is being built. For example, you'll find that

- A packet's integrity needs verification to ensure the packet was received and has not become corrupted.
- The packet must support the system's different operations through commands and requests.
- The packets need to support the ability to transfer data of various sizes to support the system's commands.

There is also the need to define requirements around how the protocol should operate. For example, if the protocol will be used on a slow serial bus such as a UART, it may be necessary to specify that the overhead should be low. Low overhead means the protocol needs to support sending and receiving information using the fewest number of bytes possible to ensure we don't max out the communication bus's bandwidth.

Defining a Packet Protocol's Fields

When defining a packet protocol to use in your systems, the exact fields that you include will vary dramatically based on your goals and what you want to achieve; however, what you'll find is that you can create a lightweight protocol whose structure remains relatively consistent across a wide range of applications. For example, as a graduate student working on small satellite flight software, we used a simple packet architecture internally to the satellite for point-to-point communication between subsystems. As a result, the communication protocol had the structure seen in Figure 14-2.

Figure 14-2 An example packet protocol definition that supports point-to-point communication[1]

As you can see, the packet protocol only contains five fields, which could be applied to nearly any application. Before we discuss that in more detail, let's discuss what each field is doing. First, the sync field is used to

synchronize the receive state machine. This field is basically a defense against spurious noise from accidentally triggering the processing of a message. You may get a noise spike in noisy environments that tricks the hardware layer into clocking in a byte. Of course, that byte could be anything, so you add a synchronization byte (or two) that must first be received before triggering the receive state machine.

The operational code (opcode) field is used to specify what operation will be performed or to tell the receiver what information can be found in the packet. For example, one might command their system to issue a soft reset by sending an opcode 0x00. Opcode 0x10 might be used to write configuration information for the device and opcode 0x20 to read sensor data. I think you get the idea. Whatever it is that your system does, you create an opcode that is associated with that operation. In smaller systems, I would define the opcode as a single byte, allowing 256 potential commands. However, this field could be expanded to two or more if needed.

Next, the data length field is used to specify how many bytes will be included in the data portion of the packet. In most applications that I have worked on, this is represented by a single byte which would allow up to 255 bytes in the packet. Multiple bytes could be used for larger payloads, though. The data size field generally does not include the sync byte or the checksum. This makes the data length a payload size field that tells how much data to include from the OP code and the data fields. There are many designs for packet protocols, and some include a packet size that covers the entire packet. I've generally designed my packets for data length only because they are more efficient from a performance standpoint.

The data field is then used to transmit data. For example, the opcode 0x10 that writes configuration information would have a data section with the desired configuration data to be written to the device. On the other hand, an opcode 0x20 for reading sensor data may not even include a data section and have a data length field value of zero! Then again, it may contain a data field that tells the device which sensors it wants to get readings back on. How much data for each opcode is entirely up to the engineer

and application.

Finally, we have the checksum field. This field could be a checksum, or it could be a Cyclical Redundancy Check (CRC) that is placed on the data packet. I've used both depending on the application and the hardware I was using. For example, I've used Fletcher16 checksums when working with resource-constrained devices that did not include a CRC hardware accelerator. The Fletcher16 algorithm is nearly as good as a CRC but can be calculated quickly compared to the CRCs calculated in hardware. Optimally, we would use an internal hardware accelerator if it is available.

A Plethora of Applications

The general format of the packet protocol we've been exploring can be extrapolated to work in various environments. Figure 14-2 shows that it is designed to work on a point-to-point serial interface. The protocol could easily be modified to work in a multipoint serial interface like RS-422 or RS-485 with just a few minor modifications. For example, in multipoint communication, there is usually a need to specify the address of the source node and the destination node. Developers could add an additional section to the packet protocol that defines their applications' address scheme, as shown in Figure 14-3.

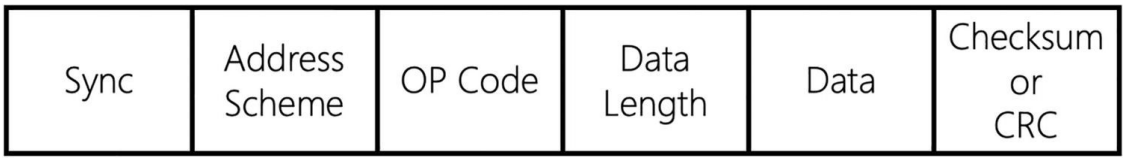

Figure 14-3 An example packet protocol definition that supports multipoint communication by including an addressing scheme

Developers may even want to modify the packet protocol to be a framed packet. A framed packet will often start and end with the same sync character. If the sync character is included in the packet data, it is replaced with some type of escape character to not interfere with the encoding or decoding algorithm. I've often seen these protocols eliminate the data or packet length fields, but I personally prefer to include them so that I can use them as a sanity check. An example of a framed packet protocol can be seen in Figure 14-4.

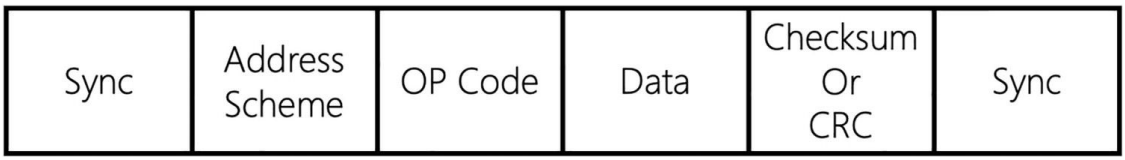

Figure 14-4 An example packet protocol definition that supports framing for multipoint communication

Packet protocols don't necessarily have to only be applied to interfaces like USART and USB. Developers can just as quickly adapt them to run on other interfaces like I2C.

In such a system, a developer wouldn't need the sync field because the I2C bus already includes a slave address which is 7 bits, along with a single bit for reading and writing. The general packet protocol could be modified to work on I2C with the modifications shown in Figure 14-5. It's essentially the same protocol again, just with some interface-specific improvements.

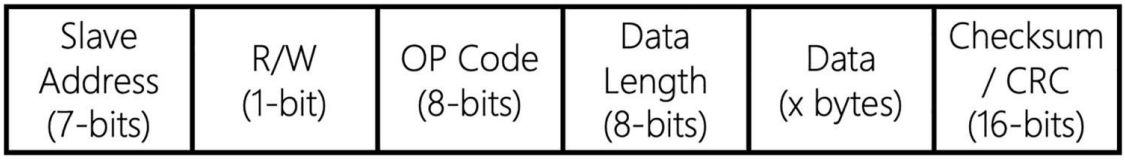

Figure 14-5 An example packet protocol adapted to work on the I2C bus

As you can imagine, this general format can be used on nearly any physical interface. If needed, you could add additional functionality such as encryption, signing, and authentication, which is beyond the scope of what we want to look at in this chapter. Just note those as additional ideas to explore on your own.

Implementing an Efficient Packet Parser

We've looked at several different adaptations for lightweight packet protocols. But, first, let's look at what it can take to architect a protocol that is reusable using the protocol that we defined in Figure 14-2.

The Packet Parsing Architecture

Before we start banging out code, it's helpful to stop and think through how a parsing module would work. There are several characteristics that we would want the module to have:

1. The module should be peripheral independent. For example, the parser should not care if the data stream came in over UART, USART, USB, CAN, I2C, SPI, etc.
2. The module should not depend on any specific circular buffer, message queue, or operating system.
3. The implementation should be scalable and configurable.

Considering these three requirements, we might arrive at a simple data flow diagram for parsing a packet that looks like Figure 14-6.

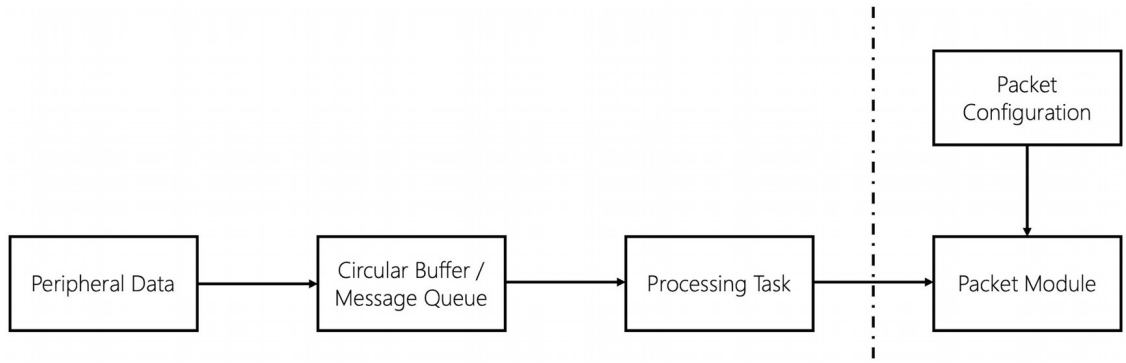

Figure 14-6 The data flow diagram for packet data starts at the peripheral and makes its way to the packet parser. The dotted line represents the interface to the packet parsing module

The preceding diagram conveys a lot of information to us. First, the data is received from a peripheral and could be retrieved through either hardware buffers, interrupts, DMA, or any other mechanism in the system. To the architect, we don't care at this stage. Second, once the data is received, it is passed into a buffer where it waits for a task or some other mechanism to consume it. Finally, the task takes the data and uses the packet parser interface to pass the data into the parsing state machine.

In this design, the packet module is abstracted from anything happening on the left side of the dashed line. It doesn't care about how it gets its data, what interface is used, or anything like that. Its only concern is that it is provided with the correct configuration to know how to parse the data and that it is provided a byte stream. We aren't going to go into details on the generic configuration during this post.

The packet parser must have two public functions to parse the packet. These

public functions include

```
bool Packet_DecodeSm(uint8_t const Data);
PacketMsg_t const * const Packet_Get(void);
```

As you can see, the primary function for our packet parser is the Packet_DecodeSm function which just takes a single byte of data and returns true if that byte completes a packet whose checksum is verified. (This example will not return checksum errors or other status information, but it can be easily added by extending the packet interface.) I abbreviate StateMachine as Sm in my code. The second function, Packet_Get, can be called to get a pointer to the packet so it can be read and processed by the application. As you can see, the packet is kept as constant data along with the pointer for where the data is.

Notice that the user of the packet module has no clue how the packet parser looks. The state machine that parses the data is a black box. This is an example of encapsulation, an excellent object-oriented design technique that can be applied even in C. In fact, with this interface, we could use any number of parsing algorithms, but as long as the user can call this interface, the application code doesn't care.

Receiving Data to Process

Typically, we might start to now code up our packet module, but I want to instead write the code that will use the packet module to demonstrate how the module will be used first. For example, I will use an STM32 development board and leverage their STM32Cube HALs. I will also assume we will use a UART to receive the packet data.

The first step is to set up the callback function for a UART receive interrupt. An example of how the interrupt callback function can be defined can be seen in Listing **14-1**.

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart2)
{
    HAL_UART_Transmit(&huart2, (uint8_t *)aRxBuffer,
ONE_BYTE, DELAY_MAX);
    CBUF_Push(RxDataBuffer, aRxBuffer[0]);
    HAL_UART_Receive_IT(&huart2, (uint8_t *)aRxBuffer, 1);
}
```

Listing 14-1 Defining a UART callback function to receive packet data

You'll notice that this callback uses HAL_UART_Transmit to repeat the sender's character. It also uses cbuf.h, an open source circular buffer implementation, to push the character into a circular buffer. The peripheral is then placed into a receive state to wait for the next character.

Setting up the circular buffer occurs during the system start-up in the initialization routine. How cbuf works is beyond the scope of the book, but in case you want to try this yourself, the code to initialize the circular buffer can be found in Listing **14-2**.

```
// initialize the circular buffer
CBUF_Init(RxDataBuffer);
// Initialize the buffer to all 0xFFFF
for(i = 0; i < myQ_SIZE; i++)
{
    CBUF_Push(RxDataBuffer, 0xFF);
}
// Clear the buffer
for(i = 0; i < myQ_SIZE; i++)
{
    CBUF_Pop(RxDataBuffer);
}
```

Listing 14-2 Example code for initializing cbuf

With these two code snippets in place, Listings **14-1** and **14-2**, we can fill out some application code to receive the data from the circular buffer and then feed it to the packet parsing module. The rate at which a system decodes data packets can be designed in many ways. First, it could be completely event driven. As part of our ISR, we could use a semaphore to tell a task that there is data to process. Second, we could define a periodic task that processes the circular buffer regularly. Which one is correct? It depends on the application's latency needs. For this example, the code could be placed in a 100-millisecond periodic task, as shown in Listing **14-3**.

```
char ch;
bool PacketReady = false;
PacketMsg_t const * const Packet = Packet_Get();
// While the buffer is not empty
while(!CBUF_IsEmpty(RxDataBuffer) && PacketReady == false)
{
    ch = CBUF_Pop(RxDataBuffer);
    PacketReady = Packet_DecodeSm(ch);
 }
 if(PacketReady == true)
 {
```

```
        // Process Packet
        Command_Process(Packet);
        PacketReady = false;
    }
```

Listing 14-3 Example task code for reading the circular buffer data, feeding the data into the packet parser, and detecting if a valid packet is ready

Now that we have the application pieces in place, let's look at the packet parsing module itself.

Packet Decoding As a State Machine

So far, we have designed the interface and application code that feeds the packet parsing module. The application code that feeds the module is entirely independent, and the parser only cares about receiving a data byte and nothing about the physical interface. We are now at the point where we are feeding data into an empty stub. Then, we can define a simple state machine to implement the packet parser where each state represents an element from our packet protocol. For example, compare the packet design in Figure 14-2 with the state machine design defined in Figure 14-7. You'll notice that the only difference is that we now have defined transitions between the various states.

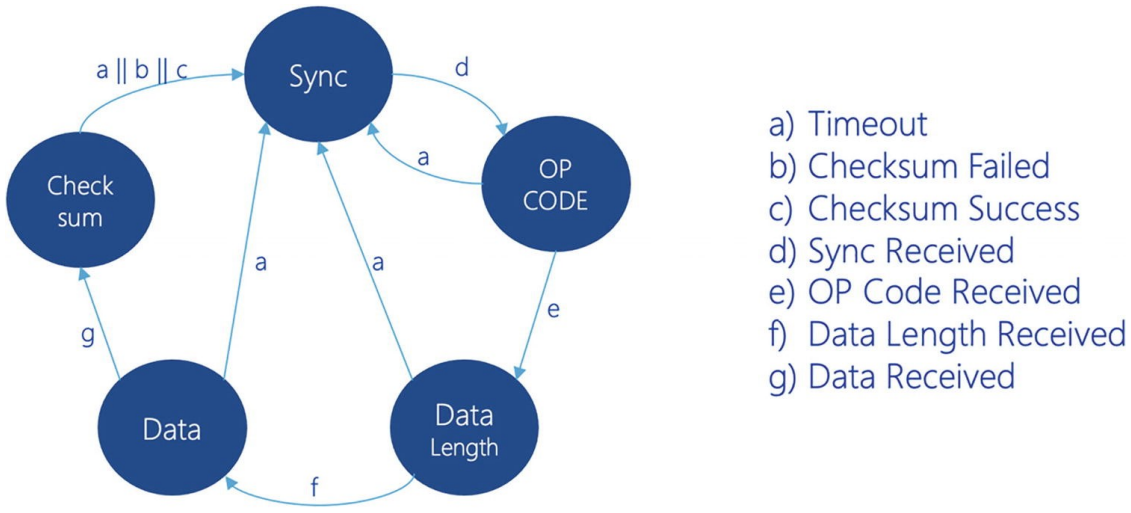

Figure 14-7 An example packet decoding state machine based on the packet protocol fields. Not receiving a byte in a given period resets the state machine as a timeout error

Each of the states defined in the state machine can be represented by an enumeration member, as shown in Listing 14-4.

```
typedef enum
{
    SYNC,                /**< Looks for a sync character  */
    OP_CODE,             /**< Receives the op code        */
```

```
    DATA_LENGTH,        /**< Receives the data size      */
    DATA,               /**< Receives the packet data    */
    CHECKSUM,           /**< Receives the checksum       */
    PACKET_STATE_END    /**< Defines max state allowed   */
} PacketRxState_t;
```

Listing 14-4 The state machine from Figure 14-7 converted into an enum

There are many different ways we could write our state machine in C. However, a method that I am particularly fond of is using tables. Tables allow us to easily specify the function that should be executed for each state and provide a place to easily add, remove, or maintain the states. To start, we define a structure that contains the information required for each state in the state machine, as shown in Listing 14-5.

```
typedef struct
{
    PacketRxState_t State;
    (bool)(*State)(uint8_t const Data);
}PacketSm_t;
```

Listing 14-5 A typedef defines the information needed to execute a state in the packet parsing state machine

We can now define the state machine in table form using the enumeration members and the functions that should be called to execute each state, as shown in Listing 14-6.

```
static PacketSm_t const PacketSm[]=
{
    {SYNC,          Packet_Sync       },
    {OP_CODE,       Packet_OpCode     },
    {DATA_LENGTH,   Packet_DataLength },
    {DATA,          Packet_Data       },
    {CHECKSUM,      Packet_Checksum   }
};
```

Listing 14-6 Defining the table that maps states to their respective function

Notice in Listing 14-6 that we define our array of PacketSm_t to be constant and static. We define the table as const for several reasons. First, we don't want the state machine to be modified during runtime. Second, declaring the table const should force the compiler to store the table in flash rather than RAM. This is good because our table contains function pointers, and we don't want to open an attack surface by allowing a buffer overrun, etc., to modify what the state machine is pointing to. (Most embedded compilers, by default, put const into flash, not RAM. Usually, you must force it into RAM using a compiler switch, but this is compiler

dependent.) Finally, we also define the table as static to limit the scope of the table to the packet module so that no external module can use it or link to it.

Now, with the state machine table set up, you are ready to implement the interface that will use the state machine. Listing 14-7 shows how simple the state machine code can be. First, Packet_DecodeSm performs a simple runtime sanity check that the PacketSmResult variable that tracks the state machine state is within the correct boundaries. We perform this runtime check rather than an assertion because if memory corruption ever occurs on PacketSmResult, we don't want to dereference an unknown memory location and treat it like a function pointer. To prevent this, we wrap a simple boundary condition check, and if that fails, we reset the state machine and set an error.

```
bool Packet_DecodeSm(uint8_t const Data)
{
    bool PacketSmResult = false;
    if(PacketState < PACKET_STATE_END)
    {
        PacketSmResult = (*PacketSm[PacketState].State)
(Data);
    }
    else
    {
        Packet_SmReset();
        PacketError = BOUNDARY_EROR;
    }
    return PacketSmResult;
}
```

Listing 14-7 An example Packet_DecodeSm implementation that uses the PacketSm table and some basic runtime error handling

The key to the Packet_DecodeSm function and running the state machine is dereferencing the function pointer. For example, Listing 14-7 contains a line of code that looks similar to the following:

```
(*PacketSm[PacketState].State)(Data);
```

This line is going to the PacketState element in the PacketSm array and dereferencing the State function pointer. So, if PacketState is SYNC, then the Packet_Sync is executed from the PacketSm table. If PacketState is DATA, then the Packet_Data function is executed, and so on and so forth. There are several advantages to this state machine setup, such as

- A new state can be added by adding a new row to the StateSm ta-

ble.

- Likewise, a state can be deleted by adding a new row to the StateSm table.
- All states are executed through a single point in the packet module rather than through complex and convoluted switch statements.
- Each state function can easily be remapped based on the application needs.

Looking at the code that we have defined so far, it's apparent that we are missing the definitions for each function in our state machine. Defining each function goes above and beyond what we want to explore in this chapter. At this point, you have a framework for decoding packets. Implementing the remaining details in the state machine functions depends on your packet protocol definition. Before we move on to command decoding, we should examine the last state in the state machine and explore how we validate the packet.

Validating the Packet

PacketDecodeSm accepts a character and proceeds through the various decoding states based on the character data it receives. The state machine progresses until it has received the checksum. Once the checksum has been received, the checksum state will run a checksum validation function to ensure that the received packet matches the received checksum. Checksum validation can be performed through a Packet_Validate function. Packet_Validate could use a software-based checksum like a Fletcher16 algorithm or a microcontroller hardware-based CRC calculator. It's completely application dependent. An example of using Fletcher16 can be seen in Listing **14-8**.

```
bool Packet_Validate(void)
{
      uint16_t ChecksumRx = 0;
      bool ChecksumValid = false;
      // Verify the checksum before executing the command
      ChecksumRx = (Packet.Checksum[0] << 8);
      ChecksumRx |= Packet.Checksum[1];
      // Calculate checksum of packet
      // The magic number 3 is to add in the sync, opcode and
      // data length to the calculation
      ChecksumValid =
```

```
Fletcher16_CalcAndCompare((char*)&Packet,
                                    Packet.Length + 3,
ChecksumRx);
      return ChecksumValid;
}
```

Listing 14-8 An example Packet_Validate function that uses a Fletcher16 checksum

The reader can take the example function in Listing 14-6 and modify it so that the Fletcher16_CalcAndCompare function is replaced with the correct function at compile time. Optimally, this function would be assigned at compile time based on how we want to validate the packet. We can do this through a function pointer, which gives us a degree of polymorphism to the implementation, which is an object-oriented design technique.

You'll recall that the Packet_DecodeSm returns a bool. Once the Packet_Validate function validates that a complete packet exists, it returns true to the higher-level application. The application will see that Packet_DecodeSm is valid and can retrieve the completed packet from memory for processing. Let's now look at how we take the validated packet and process the associated command and payload.

Command Processing and Execution

Command processing is a core feature in many embedded systems. The ability to command the system and change its state is fundamental. However, parsing and executing commands can turn application code into a giant ball of bud! Let's explore several methods that can be used to parse a packet to execute a command that is contained within it.

Traditional Command Parsers

When it comes to parsing command packets or any type of messaging protocol in C, I often see developers use two parsing methods:

- If/else if/else statements
- Switch statements

These C constructs can be effective for protocols with only a few commands,

but as the number of commands in the system approaches a dozen or more, they become inefficient. For example, consider a bootloader where we have four commands that can be executed:

- Bootloader exit
- Erase device
- Program device
- Query device

We could quickly write an if/else if/else statement to determine which OP Code was sent in the packet using code similar to that shown in Listing 14-9. In the example, five commands are supported, creating five potential branches in our packet decoder. Notice that the variable Packet is a pointer to the validated packet memory location. Therefore, we need to use -> rather than dot notation to access the OP Code. Overall, it is not too bad for just a few commands using if/else if/else statements. Our command parser, in this case, has five independent branches, therefore a McCabe Cyclomatic Complexity of only 5; therefore, the complexity of the command parser is also relatively low, testable, and manageable. (Recall, we discuss McCabe's Cyclomatic Complexity in Chapter 6.)

```
if(Packet->OP_CODE == BOOT_EXIT)
{
  Bootloader_Exit(Packet);
}
else if(Packet->OP_CODE == ERASE_DEVICE)
{
  Bootloader_Erase(Packet);
}
else if(Packet->OP_CODE == PROGRAM_DEVICE)
{
  Bootloader_Program(Packet);
}
else if(Packet->OP_CODE == QUERY_DEVICE)
{
  Bootloader_DeviceQuery(Packet);
}
else
{
  Bootloader_UnknownCommand(Packet);
}
```

Listing 14-9 An example packet decoding method that uses if/else/else if statements to identify the command and execute the command function

One potential problem with the technique used in Listing 14-9 is that the code

looks through every possible command that could be received until it finds one that matches or decides that the command is invalid. This is a bit inefficient because if we receive the last message in our statement, we're wasting clock cycles working through a bunch of if/else if cases. A more efficient way to implement the parser would be to use a switch statement like the one shown in Listing 14-10.

```
switch(Packet->OP_CODE):
  case BOOT_EXIT:
       Bootloader_Exit(Packet);
       break;
  case ERASE_DEVICE:
       Bootloader_Erase(Packet);
       break;
  case PROGRAM_DEVICE:
       Bootloader_Program(Packet);
       break;
  case QUERY_DEVICE:
       Bootloader_DeviceQuery(Packet);
       break;
  default:
       Bootloader_UnknownCommand(Packet);
       break;
```

Listing 14-10 An example packet decoding method that uses switch statements to identify the command and execute the command function

We could argue that a good compiler will reduce both code sets to identical machine instructions. However, we might only improve human readability and maintainability at this point. The great thing about the switch statement is that the compiler will generate the most efficient code to execute the statement when it is compiled, which usually results in a jump table. Instead of checking every case, the compiler creates a table that allows the value to be indexed to determine the correct case in just a few clock cycles.

The problem with switch statements is that they can become complex and challenging to maintain as they grow to over a dozen or so commands. For example, I've worked on flight software for several different small satellite missions, and these systems often have several hundred possible commands. Can you imagine the pain of managing a switch statement with several hundred entries? Spoiler alert, it's not fun! In addition, the complexity of our function will shoot to the moon because of all the possible independent branches that the code could go through.

When it comes down to it, we need a more efficient technique and one that is less complex and easier to maintain than just using switch statements. This is where a command table can come in handy.

An Introduction to Command Tables

A command table is an array of a structure that contains all information necessary to find a command that needs to be executed and the function that should be executed for that command. Creating one is quite simple. First, define a command structure that contains a human-readable command name and a function pointer to the function that should be executed. This can be done using code like that shown in Listing 14-11.

```
typedef struct
{
    Command_t Command;
    void (*function)( CommandPacket_t * Data);
}CommandRxList_t;
```

Listing 14-11 Defining a command structure that contains all the information necessary to execute a system command

You may notice in Listing 14-11 that we are once again following a similar pattern. We once again have a human-readable value as one structure member and then a function pointer as the other member. In addition, there is an undefined Command_t in the declaration. Command_t is an enumeration that contains all the commands we would expect to receive in our system. For example, for the bootloader, I may have an enumeration that looks like Listing 14-12.

```
/**
* Defines the commands being received by the bootloader.
*/
typedef enum Command_t
{
    BOOT_ENABLE,          /**< Enter bootloader */
    BOOT_EXIT,            /**< Exit bootloader */
    ERASE_DEVICE,         /**< Erase application area of
memory */
    PROGRAM_DEVICE,       /**< Program device with an s-record
*/
    QUERY_DEVICE,         /**< Query the Device */
    END_OF_COMMANDS       /**< End of command list */
};
```

Listing 14-12 The bootloader command list as a typedef enum

With the enumeration and the structure defined, we can create a command

table containing all the commands supported by the system and map those commands to the function that should be executed when the command is received. The bootloader command table then becomes something like Listing 14-13.

```
/**
 * Maps supported commands to a command function.
 */
CommandRxList_t const CommandList[] =
{
    {BOOT_EXIT,             Command_Exit            },
    {ERASE_DEVICE,          Command_Erase           },
    {PROGRAM_DEVICE,        Command_Program         },
    {QUERY_DEVICE,          Command_Query           },
    {END_OF_COMMANDS,       NULL                    },
};
```

Listing 14-13 An example command table that lists the system command and maps it to the command to execute

There are several advantages to using a table like Listing 14-12 to manage commands which benefit the developer, including

- Humans can easily read through the table quickly, which gives an "at a glance" look at the commands in the system.
- If a command needs to be added, a developer just needs to insert a new row into the table.
- If a command needs to be removed, a developer can just remove that row from the table.
- If a command operational code needs to change, it can be updated in the enumeration, and the command table does not need to change.
- The command table can be generated by a script using a system configuration file similar to what we saw in Chapter 13 with the task table generation.

There are additional advantages one could also think up, such as minimizing complexity, but at this point, our time is best spent looking at how we can execute a command from the command list.

Executing a Command from a Command Table

Once a command table has been implemented, executing a command from the table can be in several different ways. First, and most preferably, the Command_t

enum should be sequential without any gaps and start at 0x00. If this is done, executing the command can be done using the code in Listing 14-14. You'll notice that this code looks very similar to how we executed the state machine!

```
void Command_Process(CommandPacket_t const * const Packet)
{
    if(Packet->OP_CODE < END_OF_COMMANDS)
    {
        (*CommandList[Packet->OP_CODE]->function)(Packet);
    }
}
```

Listing 14-14 A function for executing the OP Code, a command, from a received data packet

All our Command_Process function needs to do is index into the array and dereference the function pointer stored there! That's it! Of course, we would probably want to add some additional error checking and maybe add some error codes if we go outside the defined bounds, but I think the reader at this point understands it, and I've removed the code for brevity.

If the enum is not sequential or has gaps in the ordering, the code in Listing 14-14 will not work. Instead, the table needs to be searched for a matching operational code, and only when a match is found is the pointer dereferenced. This makes the command parsing much less efficient. However, several methods could be used. While developers may be tempted to jump into algorithms for binary searches and so forth, using a simple loop such as that shown in Listing 14-15 can often be the simplest solution.

```
void Command_Process(CommandPacket_t const * const Packet)
{
    const CommandRxList_t * CmdListPtr;
    uint8_t CmdIndex = 0;
    // Loop through the command list for a match.
    CmdListPtr = CommandList;
    while((CmdListPtr->function != NULL) ||
          (CmdListPtr->Command != Packet->OpCode))
   {
        CmdListPtr++;
        CmdIndex++;
   }
    // Verify that we found a match and that the function is
    // not NULL
    if(CmdListPtr->function != NULL)
    {
```

```
            // Execute the command and parse out the command byte
            (*CmdListPtr->function)(Packet->Data);
        }
    }
```

Listing 14-15 An example command processing function is when the command OP Codes are not sequential and contain "holes" in the command map

As you can see, we use a while loop to search through the table for a match. This is not as efficient as being able to directly index into the array, but it can still work quite well and is easier to maintain than switch statements. There are also several other ways to write the search code, but it can come down to your personal preference, so use the code provided as an example, not gospel.

Managing System Telemetry

Let's shift focus from parsing packets and executing commands to discussing telemetry. Once again, nearly every embedded system has some type of telemetry data that it sends back to the user or even the manufacturer. The telemetry could consist of sensor values, health, and wellness data, among other things. However, there is a significant problem that is often injected into systems due to telemetry:

1. Telemetry has the potential to break dependency models and turn architecture into a giant ball of mud.

Telemetry code has a bad habit of reaching out into the system and touching every module. As a result, the smallest modules in the system suddenly start to depend on the telemetry module so that they can update it with their data. This is something that we don't want to happen. So in this section, let's explore some ideas on how you can successfully manage your telemetry code without breaking the elegance of your design.

Telemetry As a "Global" Variable

Telemetry data is stored in a structure variable in many systems and declared at global scope. Several problems present themselves when a developer takes this approach with their telemetry. First, every task and module in the system can directly access the telemetry structure. The opportunity for race conditions, data corruption, and so forth dramatically

rises. Now you might think in an RTOS-based application that you can just throw a mutex at the structure, but every module also needs to have an RTOS dependency! Before you know it, the dependency map is out of control!

The trick to minimizing the telemetry dependencies and keeping things in control is to limit how the telemetry structure is updated. In an RTOS-based application, the only application code that should be able to update the telemetry is a task. Therefore, supporting code modules like drivers, board support packages, and middleware should be queried and controlled by their respective tasks, and then that task updates the telemetry structure. This limits how data flows into and out of the structure by default.

Limiting telemetry access to tasks can still present developers with a problem. The temptation to create a global variable still exists. If the global variable exists, the chances are higher that someone who isn't supposed to access it will. However, developers can leverage an exciting technique to create the telemetry variable at a high scope with the application and then pass pointers to the telemetry structure into the task creation function! Listing 14-16 shows a task created with the telemetry task receiving a pointer to the telemetry structure through the parameter's pointer.

```
xTaskCreate(TaskCodePtr,
            TaskName,
            StackDepth,
            &Telemetry,
            TaskPriority,
            TaskHandle);
```

Listing 14-16 Passing the telemetry structure into a task at creation can limit the scope and access to the telemetry data

There are a couple of caveats to passing the telemetry pointer into the task this way. First, the task code will depend on the telemetry module to decode what members are in the structure. That should be okay since there is a limit on how many modules will access it. Next, the task is just receiving a pointer to something. It doesn't know that it is a pointer to telemetry data. In the task function itself, we will need to cast the received pointer into a pointer to the telemetry data. For example, Listing 14-17 shows how the casting would be performed. Finally, there is still a potential for race conditions and the telemetry data to become corrupted due to no mutual exclusion.

```
void Task_Module(void * pvParameters)
```

```
{
    ...
    Telemetry_t * const Telemetry = (Telemetry_t * const)
pvParameters;
    ...
}
```

Listing 14-17 Casting the pointer to telemetry data in a task to a telemetry pointer

The shared variable problem with multiple tasks being able to access the telemetry structure is not something we can ignore. The chances are that different tasks will access different members, but we can't guarantee that. On top of that, it's possible a task could be updating a telemetry member and be interrupted to transmit the current telemetry, sending out a partially updated and incorrect value. However, there is a simple solution. We can build a mutex into the telemetry structure.

Adding a mutex into the telemetry structure bundles the protection mechanism with the data structure. I like to make the structure's first member a pointer to the telemetry data. In most modern IDEs with autocomplete, when you try to access the telemetry structure, the first autocomplete option that will show up is the mutex! The developer should immediately remember that this is a protected data structure, and the mutex should first be checked. (Note: You could also build a telemetry interface that abstracts this behavior.)

```
typedef struct __attribute__((packed))
{
   SemaphoreHandle_t * MutexLock;
   ...
}Telemetry_t;
```

Listing 14-18 Example mutex pointer being added to the telemetry data structure

Telemetry As a Service

Another technique developers can use to limit telemetry from turning an elegant architecture into a giant ball of mud is to treat telemetry as a service. Developers can create a telemetry task that treats the telemetry data structure as a private data member. The only way to update the telemetry data is to receive telemetry data updates from other tasks in the system through a telemetry message queue. For example, examine the architectural pattern shown in Figure 14-8. The figure demonstrates how the message queue is available to all the tasks, but the

telemetry data and access to Task Telemetry are restricted.

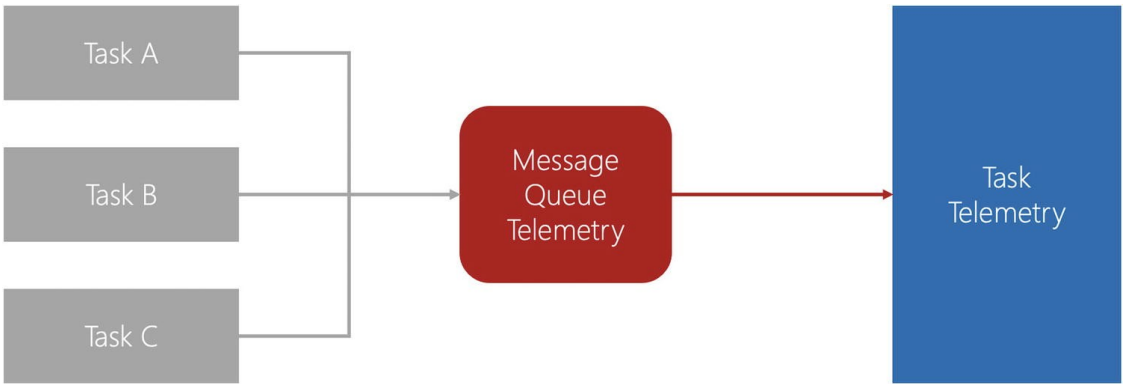

Figure 14-8 A message queue can be used as the "entry point" to perform updates on a private telemetry data member residing in Task Telemetry

Using an architectural pattern like Figure 14-8 has some trade-offs and key points to keep in mind, just like anything else in software and engineering. First, there does need to be some type of coordination or identifier between the tasks (Tasks A–C) and Task Telemetry so that Task Telemetry can identify whose telemetry data it is in the message. The coordination will require each message placed into the queue to have some type of telemetry identifier and fit a specific format. In fact, a simplified version of the packet protocol that we looked at earlier in this chapter could even work.

Next, we need to consider how to properly size the message queue. Each task will send its latest telemetry to the queue many times per cycle. The designer must decide if the message queue is read out once or twice per cycle. If it's just read once, the message queue may need to be slightly larger to maintain all the telemetry message updates. This is a bit inefficient from a memory perspective because there may be multiple duplicate messages that, when processed, will overwrite each other. However, if we process the message queue more often, we're potentially wasting clock cycles that another task could use. It's a trade-off and up to the developer to decide what is most appealing to them.

Finally, if the software changes, the maintenance for this pattern should be minimal. For example, if a new task with telemetry points is added, a new message specifier can be added. Task Telemetry can remain as is

with just the added case for the new telemetry message. The same goes for if a task is removed. Generally, the pattern is relatively scalable and should prevent the software architecture from decaying into a giant ball of mud.

Final Thoughts

Communication, commands, and telemetry are standard features in many embedded systems. Unfortunately, many implementations consider these features as one-off design decisions that result in unnecessary development costs and time expenditures. However, communication interfaces, command processing, and telemetry are common problems that most systems need to solve. In this chapter, we've seen several techniques that developers can leverage to write reusable and easily configurable code for various applications. These techniques should help you to decrease cost and time to market in future projects.

Action Items
To put this chapter's concepts into action, here are a few activities the reader can perform:

- Identify the characteristics that you need in a packet protocol. Does your current protocol meet those needs? Then, create an updated packet protocol that meets your needs.
- What is the difference between a checksum and a CRC? What can be done to improve the efficiency with which a CRC is calculated on a microcontroller?
- Implement an example software checksum and CRC. Compare the execution and memory differences. Implement a hardware-based CRC. How did the memory and execution times change?
- Design and implement a packet parsing state machine for your packet protocol. What techniques can you use to make it more reusable and scalable?
- Experiment with using state tables to transition through the various states of a state machine. How does this compare to other techniques you've used in the past?
- Implement a command parser. What differences do you notice about

using a command table vs. using if/else if/else and switch statements?
- Review how you currently implement your telemetry transmission functions. What changes can you make to minimize module dependencies?

Footnotes

1 I first published this image in *CubeSat Handbook: From Mission Design to Operations*, Chapter 10, Figure 5.