

8

TECHNIQUES FOR WRITING EMBEDDED CODE



E-next

THE NEXT LEVEL OF EDUCATION

FOR THE MOST part, writing code for an embedded platform is no different to writing code for a desktop or server system. However, there are a few differences, and it is worth bearing them in mind as you write. In this chapter we explore what some of these issues are and outline ways that you can avoid or work around the problems.

As you saw in Chapter 5, “Prototyping Embedded Devices”, one of the big differences between embedded systems and “normal” computing platforms is the lack of resources available. Whereas on laptops or servers you have gigabytes of memory and hundreds of gigabytes of storage, on a microcontroller the resources are typically measured in kilobytes. For example, your web browser will think nothing of slurping 330KB of HTML, CSS, JavaScript, and images into memory, and then copying it around to parse it and rework it into a better data structure for displaying just to show you the (relatively simple) home page of Google. That’s 150 times the total memory available on an Arduino Uno, just for the download—before you start any processing of it.

Aside from resource constraints, connected devices are, by their nature, likely to be something that people turn on and then “forget about”—not literally, of course, because, one hopes, they provide a valuable service or brighten up people’s lives. However, the owner of the device doesn’t expect to have to regularly restart or maintain it. Consequently, your system should expect to run for months or years at a time without any user intervention.

The same goes for any configuration or tuning of the system. Although it’s just about acceptable for server software to require an administrator to keep an eye on things and run through some maintenance procedure from time to time, this is not true for devices such as laptops and PCs. We’ve seen tasks such as disk defragmentation fading in importance as an explicit job for end users, for example. The aim with ubiquitous computing devices should be to take that idea of automated or *self*-maintenance further still.

MEMORY MANAGEMENT

When you don’t have a lot of memory to play with, you need to be careful as to how you use it. This is especially the case when you have no way to indicate that message to the user. The computer user presented with one too many “low memory” warning dialog boxes will try rebooting, and so will the system administrator who spots the server thrashing its disk as it pages memory out to the hard drive to increase the amount of virtual memory. On the other hand, an embedded platform with no screen or other indicators will usually continue blindly until it runs out of memory completely—at which point it usually “indicates” this situation to the user by mysteriously ceasing to function.

Even while you are developing software for a constrained device, trying to debug these issues can be difficult. Something that worked perfectly a minute ago now stops inexplicably, and the only difference might be a hard-to-spot extra character of debug logging or, worse still, something subtler such as another couple of iterations through the execution loop.

TYPES OF MEMORY

Before we get into the specifics of how to make the most of the resources you have available, it’s worth explaining the different types of memory you might encounter.

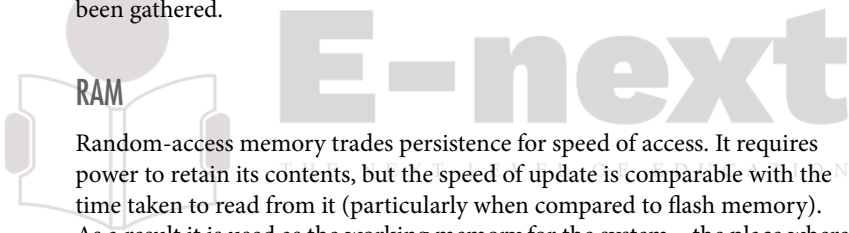
ROM

Read-only memory refers to memory where the information stored in the chips is hard-coded at the chips’ creation and can only be read afterwards.

This memory type is the least flexible and is generally used to store only the executable program code and any data which is fixed and never changes. Originally, ROM was used because it was the cheapest way of creating memory, but these days it has no cost advantage over Flash chips, so their greater flexibility means that pure ROM chips are all but extinct.

Flash

Flash is a semi-permanent type of memory which provides all the advantages of ROM—namely, that it can store information without requiring any power, and so its contents can survive the circuit being unplugged—without the disadvantage of being unchangeable forever more. The contents of flash memory can be rewritten a maximum number of times, but in practice it is rare that you'll hit the limits. Reading from flash memory isn't much different in speed as from ROM or RAM. Writing, however, takes a few processor cycles, which means it's best suited to storing information that you want to hold on to, such as the program executable itself or important data that has been gathered.



Random-access memory trades persistence for speed of access. It requires power to retain its contents, but the speed of update is comparable with the time taken to read from it (particularly when compared to flash memory). As a result it is used as the working memory for the system—the place where things are stored while being processed.

Systems tend to have a lot more persistent storage than they do RAM, so it makes sense to keep as much in flash memory as is possible. Obviously, the program code itself lives in flash. You can also provide hints to the compiler (the program which turns your source code into the machine code that the processor understands) to help it place as much as possible of the running program into flash.

If you know that the contents of a variable won't ever change, it is better to define that variable as a constant instead. In the C and C++ programming languages (which are commonly used in embedded systems), you do this by using the `const` keyword. This keyword lets the compiler know that the variable doesn't need to live in RAM because it will never be written to—only read from. Using constants this way can be a big saving if you have any large lookup tables or other big data structures such as fonts or bitmaps. Even text strings for debugging purposes can take up a noticeable amount of space. It's much better to get this storage out of RAM and into flash.

For certain processor architectures—in the authors’ experience, most notably the Harvard architecture of the Atmel chips used in Arduino—the data (RAM) and program (flash/ROM) memory spaces are separated, which means that they aren’t trivially interchangeable. As a result, you may have to do a little additional work to copy the data from flash into RAM when you want to use it. You usually can find tried-and-tested methods to do this, but they involve a tiny bit more work than just making strings and other large variables constant. This issue is therefore something to be aware of when you are working on these platforms.

The Arduino platform, for example, provides an additional macro to let you specify that certain strings should be stored in flash memory rather than RAM. Wrapping the string in `F(. . .)` tells the system that this is a “flash” string rather than a “normal” one:

```
Serial.println("This string will be stored in RAM");  
Serial.println(F("This one will be in flash"));
```

MAKING THE MOST OF YOUR RAM

Now that you’ve moved everything that you can out of RAM and into flash, all that remains is to work out ways to make better use of the free memory you have.

When you have only a few kilobytes or tens of kilobytes of RAM available, it is easier to fill up that memory, causing the device to misbehave or crash. Yet you may want to use as much of the memory as possible to provide more features. This consideration is important, and it’s easier to make the best trade-off between maximising RAM usage and reliability if your memory usage is *deterministic*—that is, if you know the maximum amount of memory that will be used.

The way to achieve this result is to not allocate any memory dynamically, that is, while the program is running. To people coming from programming on larger systems, this concept is exotic; after all, when you’re downloading some information from the Internet, for example, how could you possibly know beforehand exactly how large it is going to be? What happens if the web page you’re retrieving has gained an extra paragraph or two since you wrote the code? Your algorithm has to take into account this possibility. The standard mechanism on desktop or server systems would be to allocate just enough memory at the time you’re downloading things, *when* you know how much you’ll need.

In a deterministic model, you need to take a different tack. Rather than allocate space for the entire page, you set aside space to store the important information that you're going to extract and also a buffer of memory that you can use as a working area whilst you download and process the page. Rather than download the entire page into memory at once, you download it in chunks—filling the buffer each time and then working through that chunk of data before moving on to the next one. In some cases you might need to remember some part of one chunk before moving to the next—if a key part of the page you're parsing spans the break between chunks, for example—but a well-crafted algorithm usually works around even that issue.

An upside of this approach is that you are able to process pages which are much larger than you could otherwise process; that is, you can handle datasets which are bigger than the entire available memory for the system! The code used for Bublino, which runs on an Arduino board with only 2KB of RAM, can easily process the standard response XML from Twitter's search API, which is typically 10–15KB per download. Because Bublino's code condenses all that text into a single number—the count of new tweets—it can discard a lot of the data as it goes. All it needs to track is whether or not each tweet is newer and the timestamp for the newest tweet it has seen (so it can pick up where it left off next time around).

The downside of this sort of single-pass parsing, however, is that you have no way to go back through the stream of data. After you discard the chunk you were working on, it's gone. If the format of the data you're consuming means that you know if something is needed only by the time you reach a later part of the file, you have to set aside space to save the potentially useful segment when you encounter it. Then when you reach the decision point, you still have it available to use and can discard it then if it's not required.

Using this type of parsing also means that you generally can't build up complex data structures to check that the data is correct or complete before you process it. Rather than a rigorous parsing of an XML file, for instance, you tend to be restricted to more basic sanity tests: that it's well formed (looks like an XML file) rather than whether it's valid (conforms to a given schema). If it's a vital feature of your system that you can do the additional complex checks, you need to consider alternative approaches. In these cases you could choose a system with more memory available in the first place. Alternatively, you might cache the download to an SD card or other area of flash memory, where it could be processed in multiple passes without needing to read it all into RAM at once.

Organising RAM: Stack versus Heap

When the system first boots up, it has all RAM available to store things in, but how does it decide what goes where and how to find it later? Two general concepts for arranging memory are used: the stack and the heap. Each has its advantages and disadvantages, and computers (including most embedded systems) tend to make use of both.

The stack is organised just as the name implies—like a stack of papers. New items which are added to the stack go on the top, and items can be removed only in strict reverse order, so the first thing to be removed is the last item that was placed onto the stack.

This arrangement makes it easy for the processor to keep track of where things are and how much space is being used because it has to track only the top of the stack. The downside to this approach is that if you're finished with a particular variable, you can release the memory used for it only when you can remove it from the stack, and you can do that only when everything added since it was allocated is removed from the stack, too.

Consequently, the stack is really only useful for

- ◆ Items that aren't going to survive for long periods of time
- ◆ Items that remain in constant use, from the beginning to the end of the program

Global variables, which are always available, are allocated first on the stack. After that, whenever the path of execution enters a function, variables declared within it are added. The parameters to the function get pushed onto the stack immediately, while the other variables are pushed as they are encountered. Because all the variables within a function are available only to code inside it, when you reach the end of that function, all those parameters and variables are ready to be discarded. So the stack gets unwound back to the same size it was just before control passed to the function.

The space on the stack is used up as the algorithm dives deeper into the nest of functions and released as execution winds back. For example, consider this pseudocode:

```
// global variables
function A {
    variable A1
    variable A2
    call B()
}
function B {
    variable B1
    variable B2
    variable B3
    call C()
    call D()
}
```

```

function C {
    variable C1
    // do some processing
}

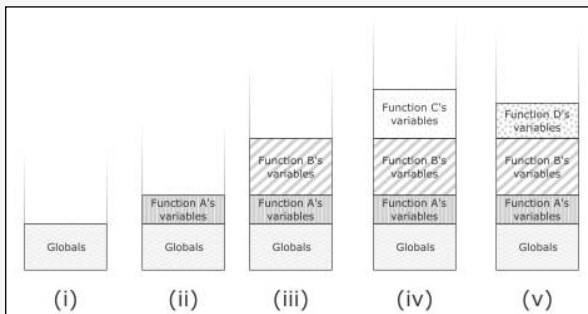
function D {
    variable D1
    variable D2
    // do some other processing
}

// Main execution starts here
// Just call function A to do something...
call A()
...

```

Then, stack usage proceeds as follows:

1. Before function A is called, the stack looks like state (i).
2. As execution moves into function A, its variables are added to the stack (ii).
3. Function A then calls function B, resulting in its variables being added to the stack (iii).
4. Inside function B, first function C is called, resulting in its variables being added to the stack (iv).
5. When execution returns from function C, its variables are removed from the stack, taking you back to stack (iii).
6. Then function D is called, so its variables are pushed onto the stack instead (v).
7. Then execution returns to function B, with D's variables removed (iii).
8. And back to A, removing B's variables (ii).
9. And, finally, you leave function A, dropping back to just the global variables being defined (i).



Stack usage at points during the example program execution.

continued

continued

As you can see, the maximum memory usage on the stack depends very much on the execution path that code can take through your code.

The heap, in comparison, enables you to allocate chunks of memory whenever you like and keep them around for as long as you like.

The heap is a bit like the seating area of a train where you fill up the seats strictly from the front and have to keep everyone who is travelling as a group in consecutive seats. To begin, all the seats are empty. As groups of people arrive, you direct them to the next available block of seats.

If, for example, a group of six people gets off at one stop, you are left with a block of six empty seats in the middle of the train. If the next group to get on is a group of three, those people can take up half of those empty seats, which leaves you with an empty block of three.

When a group of four people gets on at the following stop, they can't fit into the three empty seats because they have to sit together, so they sit towards the rear of the train where all the empty seats are.

This sort of behaviour continues happily, with people coming and going and filling up and vacating seats. But two possible problems exist. First, you might simply have more people to fit on the train than there are seats (this is the same problem as running out of memory). The second problem is more subtle: though you theoretically have enough free seats for the next group of passengers, those free seats are spread across the train and aren't available in a continuous block of free seats. This last situation is known as memory fragmentation.

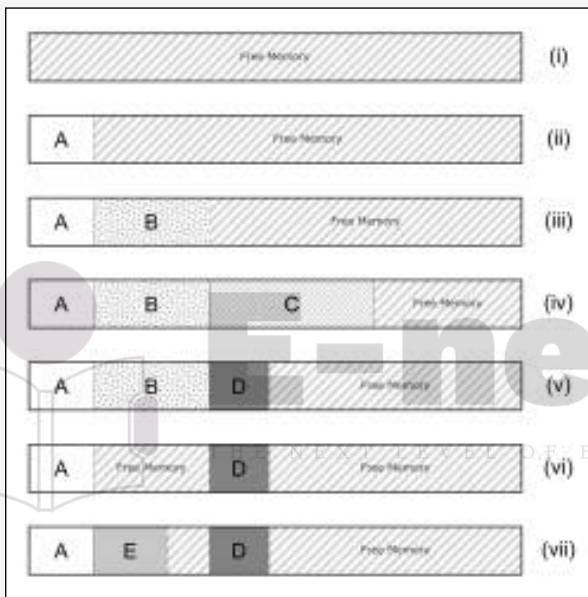
Like we did with the stack, some pseudocode will help demonstrate normal usage of the heap:

```
create object A (size 20 bytes)
create object B (size 35 bytes)
create object C (size 50 bytes)
// do some work that needs object C
delete object C
create object D (size 18 bytes)
// do more work with objects B and D
delete object B
create object E (size 22 bytes)
```

Then, as execution flows through the code, the heap will evolve as follows:

1. At the start of execution, the heap will be empty (i).
2. Object A is added to the heap (ii), taking up 20 bytes of space.
3. Object B is added to the heap (iii), consuming a further 35 bytes straight after the space for object A.
4. Object C is added to the heap (iv), adding 50 bytes to the heap right after object B.
5. Object C is no longer needed and is deleted, releasing the space it consumed on the heap and taking us back to heap (iii).

6. Object D is created and takes up 18 bytes of the space just vacated by object C (v).
7. Now object B is finished with and deleted. As other code might be relying on the position of object D, we can't move it, so there's now a free space between objects A and D (vi).
8. Object E is created. It requires 22 bytes of space, which means it will fit in the hole left by object B (vii).



Heap usage at points during the example program execution.

What we've called *deterministic memory usage* boils down to avoiding placing things on the heap at all costs. That is, in systems with only a few kilobytes of RAM, we recommend exclusively using the stack to store variables. But strictly speaking, it is still possible to run out of memory when just using the stack. This situation is called *stack overflow*. On some processors this situation occurs because the stack is a fixed block partitioned off in memory: if your program has used more of the stack at some point during its execution (for example, in a deeply nested recursive routine), it could outgrow this fixed space. However, even on architectures where the stack can automatically grow to use all available RAM, some path through your algorithm may use more than is available.

One way to reduce the chance of this situation occurring is to keep the number of global variables to a minimum. Global variables are attractive,

particularly to newcomers to coding because they're available everywhere. You don't have to worry about how to pass them from function to function or work out why the compiler is complaining that variable such-and-such hasn't been defined despite it being plainly obvious to you...just there...in that *other* function.... However, because they are always allocated (as you can see from the example in the preceding sidebar), any global variables take up valuable RAM at all times. In comparison, local variables exist only during the function in which they're declared (assuming you don't use modifiers such as `static` to make them persistent) and so take up space only when they're needed. If you can move more of your variables inside the functions where they're actually used, you can free up more space for use during other parts of the execution path.

The other way to keep down your stack usage, or at least within well-defined bounds, is to avoid using recursive algorithms. Although they are elegant and can make your code easier to understand, the stack grows with each recursion. If you know how many times a given function will recurse for the expected inputs, this may not be a problem. But if you cannot be certain that the algorithm won't recurse so many times that it blows your stack, it may be better, in an embedded system, to rework your algorithm as an iterative one. An iterative algorithm has a known stack footprint, whereas a recursive one adds to the stack with each level of recursion.

PERFORMANCE AND BATTERY LIFE


When it comes to writing code, performance and battery life tend to go hand in hand—what is good for one is usually good for the other. Whether either or both of these are things that you need to optimise depends on your application. A device which is tethered to one place and powered by an AC adaptor plugged into the wall isn't as reliant on energy conservation, for example. However, consuming less energy is something to which all devices should aspire.

Similarly, if you're building something which doesn't have to react instantly—maybe an ambient notifier for a weather forecast, which doesn't have any ill effect if it updates a few seconds later—or if it doesn't have an interactive user interface which needs to respond promptly to the user's actions, maximising performance might not be of much concern.

For items which run from a battery or which are powered by a solar cell, and those which need to react instantaneously when the user pushes a button, it makes sense to pay some attention to performance or power consumption. Although there is a lot of truth in the Donald Knuth (or was it Edsger Dijkstra,

opinion seems divided (<http://hans.gerwitz.com/2004/08/12/premature-optimization-is-the-root-of-all-evil.html>) quote that “premature optimization is the root of all evil”, it is useful to know some of the techniques which can be applied. An awareness can help you default to more efficient algorithms when there is an otherwise arbitrary choice to be made. (However, we firmly believe that understandable, maintainable code trumps the efficient if the latter is more obtuse—at least until you profile your code to work out where the optimizations are needed.)

A lot of the biggest power-consumption gains come from the hardware design. In particular, if your device can turn off modules of the system when they're not in use or put the entire processor into a low-power sleep mode when the code is finished or waiting for something to happen, you have already made a quick win. That said, it is still important to optimize the software, too! After all, the quicker the main code finishes running, the sooner the hardware can go to sleep.



One of the easiest ways to make your code more efficient is to move to an event-driven model rather than polling for changes. The reason for this is to allow your device to sit in a low power state for longer and leap into action when required, instead of having to regularly do busywork to check whether things have changed and it has real work to do. Setting up this model is trickier to do with the networking code if you are acting as a client, rather than waiting as a server. The techniques such as long polling that you saw in Chapter 7, “Prototyping Online Components”, or protocols like Message Queue Telemetry Transport (MQTT) or basic socket connections enable you to work around this situation.

On the hardware side, look to use processor features such as comparators or hardware interrupts to wake up the processor and invoke your processing code only when the relevant sensor conditions are met. If your code needs to pause for a given amount of time to allow some effect to occur before continuing, use calls which allow the processor to sleep rather than wait in a busy-loop.

If you can reduce the amount of data that you're processing, that helps too. The service API that you're talking to might have options which reduce how much information it sends you. When you're downloading tweets from a certain account on Twitter, for example, you can ask for only tweets after a specified ID. The first time you call the API, you have to deal with all the tweets it sends, but on subsequent calls, you can ask just for tweets since the ID of the most recent one that you already processed.

This optimization works only if the API that you're connecting to offers that sort of feature. For many operations, restricting things is a concept that simply doesn't make sense. If it would make sense to restrict the data, but the existing API doesn't provide that possibility, you always have the option of writing a service of your own to sit between the device and the proper API (known as a *shim service*). The shim service has all the processing power and storage available to a web server and so can do most of the heavy lifting. After this, it just sends the minimum amount of data across to be processed in your embedded system.

For instance, Bublino talks directly to Twitter and downloads a full set of search results as XML each time it checks for new messages. All this data is processed and finally reduced down to a single number—how many new tweets it finds. In theory, an optimised intermediary service could perform the searches and just transmit a single number to the Bublino device. The downside to that is you would need to write another service and maintain infrastructure to support it. Also, as the product scales, it would funnel more and more requests over the Internet route between the “Bublino Server” and the Twitter servers. In this case, it is better to have that traffic distributed across the globe wherever the Bublino devices themselves live.

When it comes to raw performance of your coding algorithm itself, nothing beats profiling to work out where the speed bottlenecks are. That said, following are a few habits that, if they become ingrained, help make your code generally more efficient:

- When you are writing if/else constructs to choose between two possible paths of execution, try to place the more likely code into the first branch—the *if* rather than the *else* part—as follows:

```
if something is true
    The more likely to happen code goes here
else
    The less likely path of execution should go here
```

This allows the pre-fetch and lookahead pipelining of instructions to work in the more common cases, rather than requiring the lookahead instructions to be discarded and the pipeline refilled. Having to re-prime the pipeline takes only a few extra processor cycles, but every bit helps.

- You saw in the “Memory Management” section that declaring data as constant where it is known never to change can help the compiler to place it into flash memory or ROM, but it can also help the compiler to know how to optimise the code. In some scenarios it is quicker to insert a value into the code as a plain number rather than to load that value

from a variable's location in memory somewhere, and if the compiler knows what the variable's value is always going to be, it can make that substitution.

- Avoid copying memory around. Moving big chunks of data from place to place in memory can be a real performance killer. In an ideal world, your code would look only at the data it needed to and read it only once. In practice, achieving this result is difficult but is a good way to help challenge your assumptions on how to write code. This is particularly an issue in protocol work, where data is initially read into a packet buffer for the Ethernet layer, say, and then passed up to the IP layer, then the TCP layer, followed by the HTTP code, and finally the application layer. A naive approach would copy the data at each step, as you unpack the relevant protocol headers. This approach could result in all the application data being copied five times as it travelled up the stack. A better approach would be to have a pointer or a reference to the initial buffer passed from layer to layer instead. In addition to the length of the data, you may need to store an offset into the buffer to show where the relevant data starts, but that's a small increment in complexity to greatly reduce how much data is copied around.
- Related to the previous point, when you do need to copy data around, the system's memory copying and moving routines (such as `memcpy` and `memmove`) usually do a more efficient job than you could, so use them. Where possible, and this is particularly the case on 32-bit processors such as the ARM family, they use processor instructions that copy more than one byte in a single operation, thus considerably speeding up the process.

LIBRARIES

These days, when developing software for server or desktop machines, you are accustomed to having a huge array of possible libraries and frameworks available to make your life easier. Need to parse a chunk of RSS XML? No problem. Just pull in the RSS parsing library for your language of choice. Want to send an email? Don't worry; there's a module for that, too. And so on and so on.

In the embedded world, tasks are often a little trickier. It's getting better with the rise of the system-on-chip offerings and their use of embedded Linux, where most of the server packages can be incorporated in the same way as you would on "normal" Linux. The trickiest part is likely to be working out how to recompile a library for your target processor if a prebuilt version isn't readily available for your system—for example, for ARM.

On the other hand, microcontrollers are still too resource-constrained to just pull in mainstream-operating system libraries and code. You might be able to use the code as a starting point for writing your own version, but if it does lots of memory allocations or extensive processing, you probably are better off starting from scratch or finding one that's already written with microcontroller limitations in mind.

We don't have space here to cover all the possible libraries that are available, and we're by no means aware of everything that's available. However, here are a few which might be of interest:

- **lwIP:** lwIP, or LightWeight IP (<http://savannah.nongnu.org/projects/lwip/>), is a full TCP/IP stack which runs in low-resource conditions. It requires only tens of kilobytes of RAM and around 40KB of ROM/flash. The official Arduino WiFi shield uses a version of this library.
- **uIP:** uIP, or micro IP (http://en.wikipedia.org/wiki/UIP_%28micro_IP%29), is a TCP/IP stack targeted at the smallest possible systems. It can even run on systems with only a couple of kilobytes of RAM. It does this by not using any buffers to store incoming packets or outgoing packets which haven't been acknowledged. This means that some of the retransmission logic for the TCP layer bleeds into the application code, making your code more tightly coupled and more complex. It's quite common on Arduino systems which don't use the standard Ethernet shield and library, such as the Nanode board, using the Ethercard port for AVR (<https://github.com/jcw/ethercard>).
- **uClibc:** uClibc (<http://www.uclibc.org/>) is a version of the standard GNU C library (glibc) targeted at embedded Linux systems. It requires far fewer resources than glibc and should be an almost drop-in replacement. Changing code to use it normally just involves recompiling the source code.
- **Atomthreads:** Atomthreads (<http://atomthreads.com/>) is a lightweight real-time scheduler for embedded systems. You can use it when your code gets complicated enough that you need to have more than one thing happening at the same time (not quite literally, but the scheduler switches between the tasks quickly enough that it looks that way, just like the multitasking on your PC).
- **BusyBox:** Although not really a library, BusyBox (<http://www.busybox.net/>) is a collection of a host of useful UNIX utilities into a single, small executable and a common and useful package to provide a simple shell environment and commands on your system.

DEBUGGING

One of the most frustrating parts of writing software is knowing your code has a bug, but it's not at all obvious where that bug is. In embedded systems, this situation can be doubly frustrating because there tend to be fewer ways to inspect what is going on so that you can track down the issue.

Building devices for the Internet of Things complicates matters further by introducing both custom electronic circuits (which could be misbehaving or incorrectly designed) and communication with servers across a network. Troubleshooting electronic circuits is outside the scope of this book, but we'll cover some ways to debug the network communication.

Modern desktop integrated development environments (often shortened to IDEs) have excellent support for digging into what is going on while your code is running. You can set breakpoints which stop execution when a predefined set of conditions is met, at which point you can poke around in memory to see what it contains, evaluate expressions to see whether your assumptions are correct, and then step through the code line by line to watch what happens. You can even modify the contents of memory or variables on the fly to influence the rest of the code execution and in the more advanced systems rewrite the code while the program is stopped.

The debugging environment for embedded systems is usually more primitive. If your embedded platform is running a more fully featured operating system such as Linux, you are better placed than if you're developing on a tiny microcontroller.

Systems such as embedded Linux usually have support for remote debugging with utilities such as `gdb`, the GNU debugger (www.gnu.org/software/gdb/). This utility allows you to attach the debugger from your desktop system to the embedded board, usually over a serial connection but sometimes also over an Ethernet or similar network link. Once it is attached, you then have access to a range of capabilities similar to desktop debugging—the ability to set breakpoints, single-step through code, and inspect variables and memory contents.

Another way to get access to desktop-grade debugging tools is to emulate your target platform on the desktop. Because you are then running the code on your desktop machine, you have access to the same capabilities as you would with a desktop application. The downside of this approach is that you aren't running it on the exact hardware that it will operate on in the wild.

Although it is a useful way to flush out initial bugs and get things mostly working, there's likely to be the odd problem that you don't catch this way and that reveals itself only when you run it on the final hardware.

Emulation is a good approach if the software is particularly involved and/or complex because that's the scenario in which you need the most development and debugging time. However, the further the target hardware is from a desktop PC—particularly when it comes to having specialised sensors or actuators—the harder it is to write software subsystems for the emulator which accurately reflect the behaviour of the electronics.

If you need on-the-hardware debugging and your platform doesn't allow you to use `gdb` (or if the serial port is in use for another part of the system), JTAG access might give you the capabilities you need. JTAG is named after the industry group which came up with the standard: the Joint Test Action Group. Initially, it was devised to provide a means for circuit boards to be tested after they had been populated, and this is still an important use.

However, since its inception, JTAG has been extended to provide more advanced debugging features. Of particular interest from a software perspective are those features available when connected to some software on a separate PC called an *in-circuit emulator* (ICE). These allow you to use the additional computer to set breakpoints, single-step through the code running on the target processor, and often access registers and RAM too. Some systems even allow you to trigger the debugger from complex hardware events, which gives you even better control and access than debuggers such as `gdb`.

If you don't have access to any of these tools, you have to fall back on some of the simpler, yet tried-and-tested techniques.

The most obvious, and most common, poor-man's debugging technique is to write strings out to a logging system. This approach is something that almost all software does, and it enables you to include whatever information you deem useful. That could be the value of certain variables at key points in the code. Or if you suspect the system is running out of RAM, writing out the amount of free space at startup and then at various points throughout the code can help you work out whether that is the case. And if your code appears to hang during execution, something as simple as some "reached line X" debug output can let you narrow in on the offending point with a binary chop approach.

If you have access to a writable file system, for example, on an SD card, you can write the output to a file there, but it is more common to write the information to a serial port. This approach enables you to attach a serial monitor, such as HyperTerminal on Windows, to the other end of the connection and see what is being written in (pretty much) real time.

You should be aware of a couple of gotchas, though, but in many cases you won't encounter them. The first is obviously the amount of space needed for any logging information: all the strings and code to do the logging have to fit into the embedded system along with your code.

The second gotcha is alluded to in the “pretty much” modifier to the previous real-time line. As the serial communication runs at a strict tempo, typically a small buffer is used to store the outgoing data while it is being transmitted. If your code hangs or crashes very soon after your logging output, there's a chance that it won't be written out over the serial connection before the system halts. So, if you are printing out “got here” messages to work out how far through your program things got before encountering the bug, there's a chance that it actually got a tiny bit further than your serial log would suggest. This isn't just an issue for serial logging; the file system tends to have a buffer for writing out data to the file too, so a log file may have the same problem.

Because most Internet of Things devices have a persistent connection to the network, you could add a debugging service to the network interface. This way, you can create a simple service which lets you connect using something as basic as telnet and find out more about what is happening in real time. At its simplest, this service could output the logging data which would otherwise be directed to the serial port, or it might understand a number of simple programmer-defined commands to query parts of the system status or trigger functionality or test code. Obviously, one warning with this approach is to take care that you don't inadvertently open a backdoor security hole. We don't recommend leaving such a system in place in the production models.

Actually, even without a dedicated debug interface on the network, you can use the fact that your device has network connectivity to help figure out what has gone wrong.

TCP/IP stacks generally offer basic capabilities such as responding to ICMP ping requests, even if the higher-level code isn't doing what you expect. If you know the IP address of the device and it responds when you query it with the `ping` network utility, you can infer that at least some part of the system is still functioning.

Taking that thinking further, if you can connect a computer somewhere on the network path between the device and the service it communicates with, running a *packet sniffer* enables you to see what is happening at the network level. (Wireshark, <http://www.wireshark.org/>, is the authors' usual choice here.) Usually, you monitor the network at one end of the connection or the other—with a computer connected to the same LAN as the device or running the packet sniffer software on the remote server.

Unless the network has very little traffic, you need to use the filtering options to reduce the amount of information flowing down to something manageable. Filtering on the MAC address of the device is a good approach if you're at the device end; otherwise, use its IP address to restrict the display just to traffic to and from the device. This approach enables you to see whether it's registering with the network (if there's DHCP traffic) and if it's succeeding in connecting to the remote service and sending out the relevant data and getting the correct response—all of which, we hope, helps you narrow down where the problem lies.

At a slightly higher level, you can also use the logging and software on the server to gather information about the device's activity. If the transport between the device and the server is a standard protocol, such as the HTTP communication with a web server, there is a good likelihood that any requests were recorded in the server log file. If you can modify the server API too, you can add analytics to keep track of the time the client last accessed the service, for example, or the count of API accesses.

If all else fails, the debugging tool of (often not-so-) last resort is a variation on what was probably your first step in building hardware: flashing an LED. As long as you have one GPIO pin free, you should be able to connect an LED and have your code turn it on at a given point. As with the string logging approach, you can use this technique to narrow in on a problem area if the system hangs or possibly use different patterns of flashing to indicate different conditions.

You also are able to reuse an LED that's normally used elsewhere in the system, during debugging at least, as long as the different uses can be distinguished from each other. Adrian remembers using this method when debugging the network stack on a mobile phone web browser; the only means of indicating the activity lower down was to flash the (fortunately) two-colour status LED one colour for incoming packets and the other colour for packets heading out.

SUMMARY

Although this chapter by no means provides a comprehensive list of ways to improve your embedded coding, we hope it provides some useful tips and pointers. To make it easier to refer to and refresh your memory, revisit the main points here:

- Move as much data and so forth as possible into flash memory or ROM rather than RAM because the latter tends to be in shorter supply.
- If items aren't going to change, make them constant. This makes it easier to move them into flash/ROM and lets the compiler optimise the code better.
- If you have only tiny amounts of memory, favour use of the stack over the heap.
- Choose your algorithm carefully. A single-pass algorithm enables you to process much more data than reading it all into memory, and iterative rather than recursive options make memory use deterministic.
- For best power usage, spend as much time as possible asleep.
- If you aren't using it (whatever "it" is), turn as much of it off as you can. This advice applies to the processor (drop into low-power mode) as much as other subsystems of the hardware.
- Optimisations can live on the server side as well as in the device. A nonpolling or reduced amount of data transferred improves both sides of the solution.
- Avoid premature optimisation. If you hit performance problems, profile to work out where the issues lie.
- Copying memory is expensive, so try to do as little of it as you can.
- Work *with* the compiler, rather than against it. Order your code to help the likely execution path and use constants to help it optimise.
- Choose libraries carefully. One from a standard operating system might not be a good choice for a more embedded environment.
- Tools such as `gdb` and JTAG are useful when debugging, but you can get a long way with just outputting text to a serial terminal or flashing an LED.
- Careful observation of the environment surrounding the device can help you sniff out problems, particularly when it's connected to the Internet and so interacting with the wider world.