SYLLABUS

Unit	Details
IV	**Techniques for Writing Embedded Code:** Memory Management, Types of Memory, Making the Most of Your RAM, Performance and Battery Life, Libraries, Debugging. **Business Models:** A Short History of Business Models, Space and Time, From Craft to Mass Production, The Long Tail of the Internet, Learning from History, The Business Model Canvas, Who Is the Business Model For? Models, Make Thing, Sell Thing, Subscriptions, Customisation, Be a Key Resource, Provide Infrastructure: Sensor Networks, Take a Percentage, Funding an Internet of Things Startup, Hobby Projects and Open Source, Venture Capital, Government Funding, Crowdfunding, Lean Startups.

Books and References

Sr. No.	Title	Author/s	Publisher	Edition	Year
1.	Designing the Internet of Things	Adrian McEwen, Hakim Cassimally	WILEY	First	2014
2.	Internet of Things Architecture and Design	Raj Kamal	McGraw Hill	First	2017
3.	Getting Started with the Internet of Things	Cuno Pfister	O'Reilly	Sixth	2018
4.	Getting Started with Raspberry Pi	Matt Richardson and Shawn Wallace	SPD	Third	2016

Chapter 8

Techniques for Writing Embedded Code

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. 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. Aside from resource constraints, connected devices are, by their nature, likely to be something that people turn on and then "forget about". 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. The aim with ubiquitous computing devices should be to take that idea of automated or self-maintenance further still.

8.1 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.

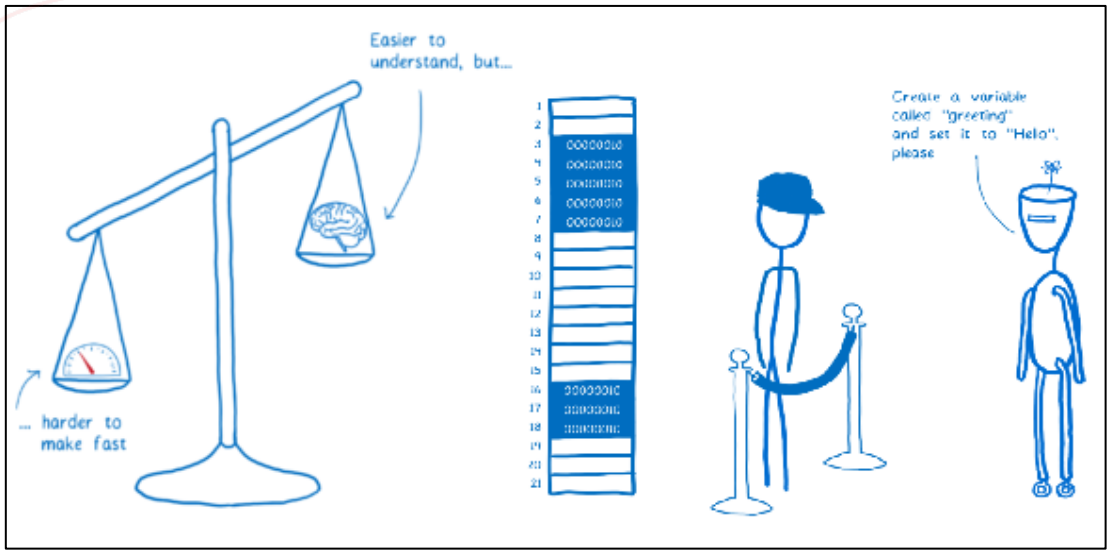

Figure 8-1: Memory Management

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.

8.1.1 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.

Video URL: https://www.youtube.com/watch?v=c4NPZcfRyV0

Video 8-1: Types of Memory

8.1.1.1 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.

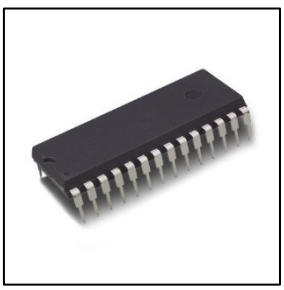

Figure 8-2: Typical ROM chip

8.1.1.2 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.

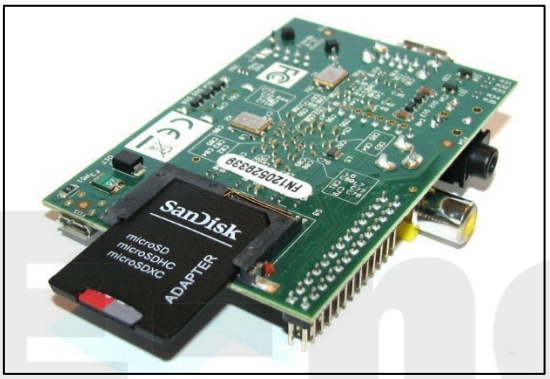

Figure 8-3: Flash Memory attached to an Embedded Board

8.1.1.3 RAM

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. 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. 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. 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.

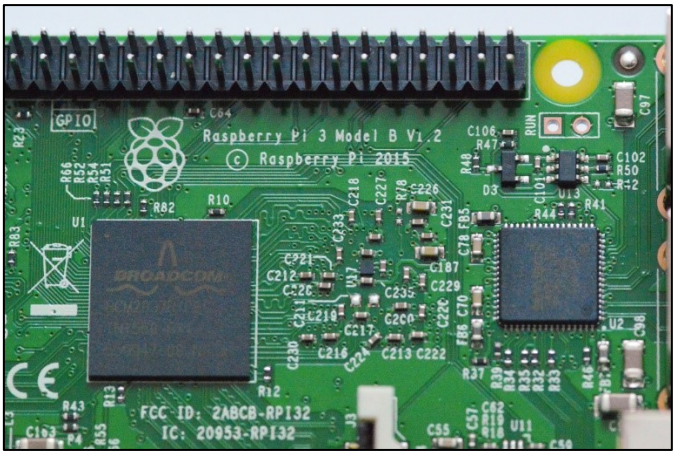

Figure 8-4: Broadcom RAM on Raspberry Pi 3 Model B

8.1.2 Making the most of your RAM

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. This consideration is important, and it's easier to make the best trade-off between maximizing 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.

In a deterministic model, 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 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. 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).

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.

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.

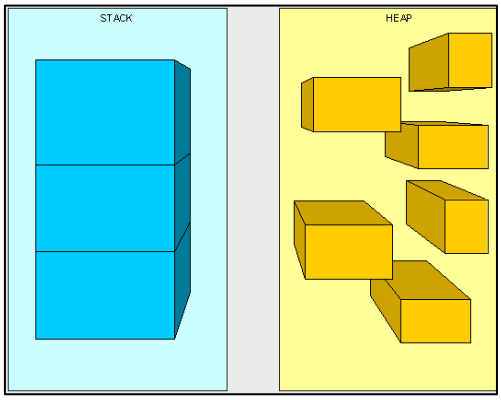

Figure 8-5: Stack v/s Heap

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, because 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. However, because they are always allocated, any global variables take up valuable RAM at all times. In comparison, local variables exist only during the function in which they're declared 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.

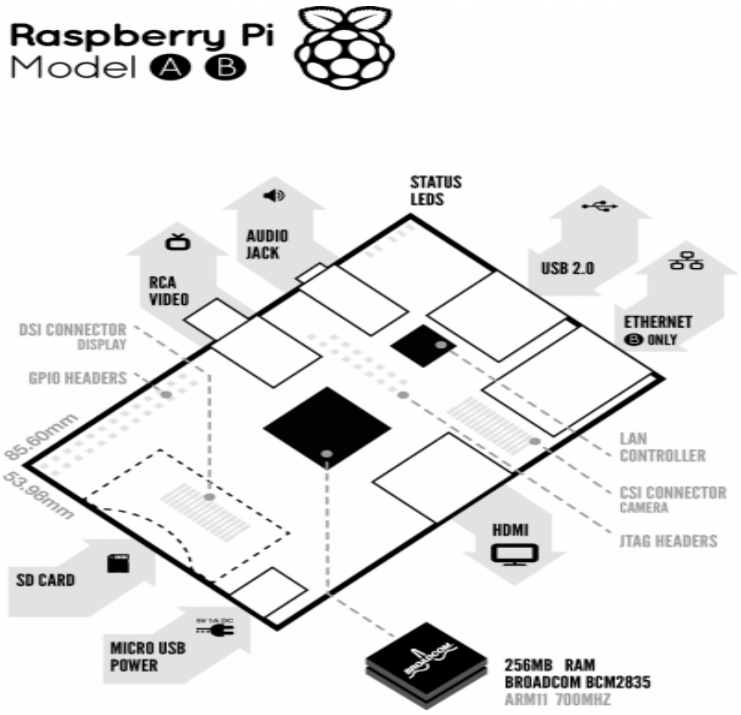

Figure 8-6: Memory and Other peripherals layout on Raspberry Pi

8.2 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 optimize depends on your application. 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. 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.

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. If 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. 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.
- 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 optimize 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.
- 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.

8.3 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. 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.

Here are a few libraries which might be of interest:

- **lwIP**: lwIP, or LightWeight IP, 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, 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. 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.
- **uClibc**: uClibc 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 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 (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 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.

8.4 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.

Modern desktop 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. 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.

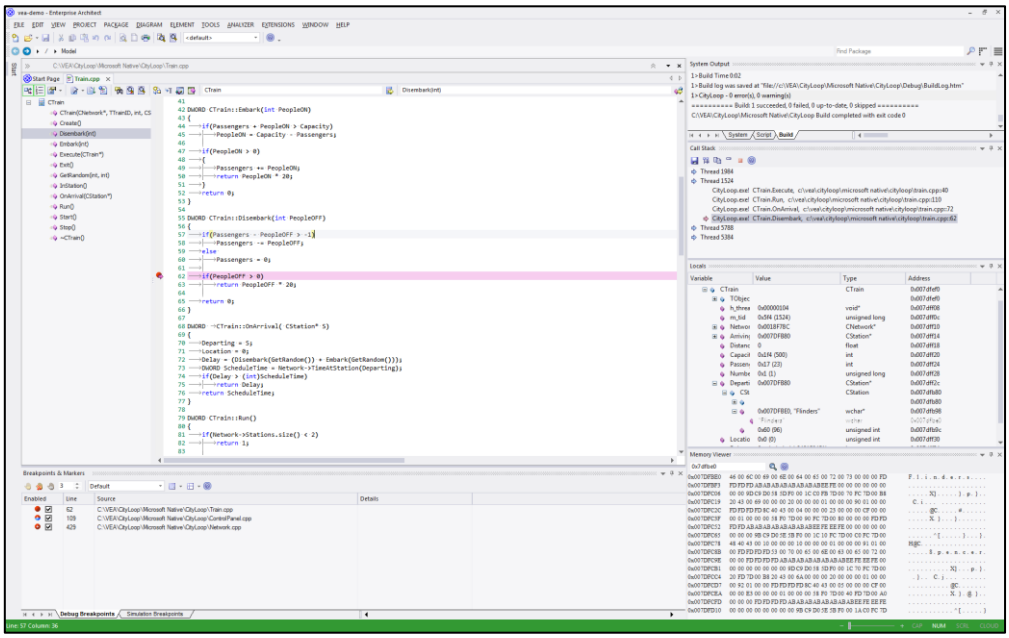

Figure 8-7: Typical debugging interface for Python

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.

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, 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.

Figure 8-8: JTAG device

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. 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 real time.

You should be aware of a couple of problems, 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 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. 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. 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.

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. 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 all

else fails, the debugging tool of 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.

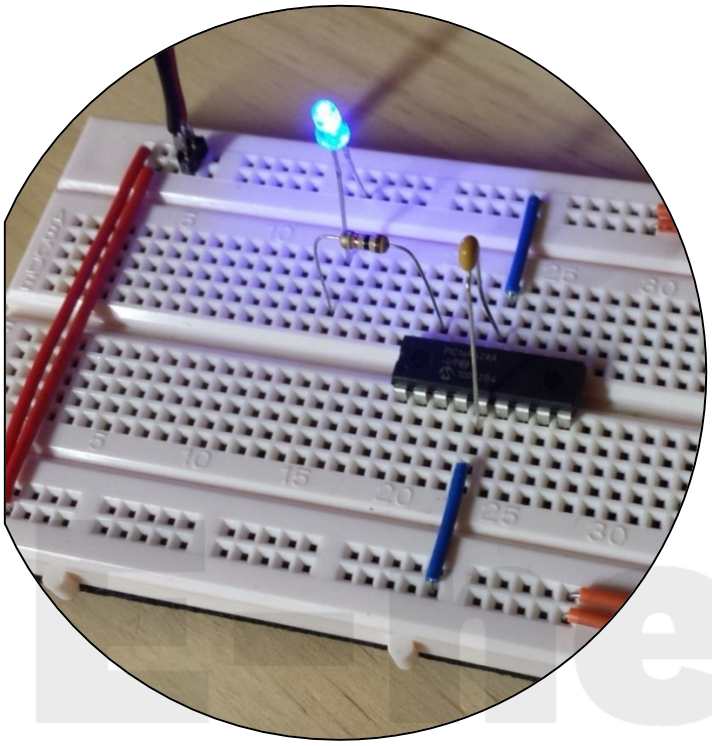

Figure 8-9: Flashing an LED

8.5 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.

8.6 Review Questions

1. Explain the different types of memory.
2. How can usage of RAM be optimised in embedded systems?
3. Compare between stack and heap in terms of organising data in RAM.
4. How can performance and battery life help in improving the efficiency of code?
5. Which libraries will be useful for developing code for embedded systems?
6. Explain the various techniques used for debugging an embedded environment.

Chapter 9

Business Models

Business model can be defined as "hypothesis about what customers want, how they want it, and how an enterprise can organize to best meet those needs, get paid for doing so, and make a profit". This definition brings together a number of factors:

- A group of people (customers)
- The needs of those customers
- A thing that your business can do to meet those needs
- Organisational practices that help to achieve this goal—and to be able to carry on doing so, sustainably
- A success criterion, such as making a profit

All these aspects are relevant as much to hobbyist or not-for-profit projects as they are to commercial enterprises, though for the last point profit might be substituted for "improving the world" or "having fun" as criteria for success.

9.1 A Short History of Business Models

From the earliest times, and for the great majority of human existence, we have gathered in tribes, with common property and shared resources. This is an almost universal pattern amongst hunter-gatherers, as it means that every member of the tribe can find food and shelter even if they have not been lucky foraging or hunting that day. We could describe this form of collectivism as a basic gift economy. Gift economies develop where those with the appropriate skills can provide their products or services—hunting, pottery, livestock, grain, childcare—and expect repayment of this obligation not immediately but with a gift of comparable worth later. This is not a written debt but a social obligation, which the recipient will repay in due course.

Development of systems such as barter and money developed only at the edges, between different tribes. We could argue that the first of what we could recognize as modern business models developed at these borders and resulted from the technology required to move products and obligations through space and time.

9.1.1 Space and Time

While neighbouring tribes might have discovered variants in the local area's resources—animal, vegetable or mineral—it is when trade develops with others from far-off lands that it becomes really interesting. A merchant might sell silks made in his village to a region where these cloths are rare and in demand in exchange for aromatic

spices which will be highly prized back home. But long-distance trade brings with it a whole set of problems: Merchants have to carry larger quantities of goods for sale and want to maximize the time travelling rather than doing the myriad tasks required for subsistence and shelter. Their goods and food carried will have to last far longer, so they will need to be protected and preserved. Above all, they need to have a reliable means of transport for themselves and their merchandise. Technological advancements such as waterway navigation and portage of boats over land opened up new possibilities, as did the domestication of animals such as the camel, which unlocked trade routes through the Western Arabian deserts. The preservation of food was done, either through salting or smoking, or simply better storage technology such as grain silos.

As well as facilitating transportation through space, preservation is also a way of transporting goods through time. A farmer or trader who can afford to not eat or sell all his produce during the glut of harvest can fetch a better price months later at a higher price. So, a merchant trader's business is transporting goods through space and time, and his suppliers, the producers, benefit from that by being able to sell a bulk of their produce in one go, after which they can continue with their daily life and work. Money, then, abstracted trade further, setting an easy-to-calculate exchange rate between a fixed currency and the product being exchanged. In the original gift economies, producers could pay their obligations only periodically or intermittently, according to the rhythms of hunting, farming, or craft. With money, this obligation was abstracted and could be paid back at arbitrary times. In this sense, money is another technology which allows travel through time. The versatility and ease of calculation which this development brought with it made it easier to develop new business models, such as investing in other merchants' trade expeditions in return for a given share or the development of interest on loans.

Video URL: https://www.youtube.com/watch?v=eUyRI66JkgQ

Video 9-1: History of Business

9.1.2 From Craft to Mass Production

When Gutenberg demonstrated his printing press circa 1450, books changed from being priceless treasures, hand-crafted by monks and artisans, to a commodity that could be produced. Soon every bourgeois family could afford their own books, at least a copy of the Gutenberg Bible, the first mass-produced book. It is no exaggeration to suggest that the invention laid the foundations for an information culture which is currently exemplified by the Internet and the World Wide Web. Trade routes would play their part here too, as the printing press spread to the New World and India via the sea routes that would be discovered by the end of the century. The cost of printing would become ever smaller as the technology spread, leading to new business models with the rise of newspapers and pamphlets.

In 1884, the British company Lever Brothers launched Sunlight Soap, the first household soap to be sold not by weight, to be cut in the shop by the grocer, but packaged in bars and branded with a logo. This was an innovation in mass consumerism, whereby the brand established a link of trust direct with the consumer, relegating the middleman, the grocer, to becoming just a way to deliver the product to the consumer. Mass production, perfected by Ford Motor Company, was another major change in business model, driven not by how Henry Ford sold his cars but by how he made them. Ford moved away from the "craft production" of cars sold by commission to highly custom requirements and made by skilled craftsmen. Rather he insisted on standard gauges for parts so that the cars could be assembled and fitted together, ending up identical. This approach made it simple to maintain and repair a Ford car, so the average person could afford to buy one without employing a mechanic to keep it working. The fact that mass production also drove down the costs to produce these cars also helped keep them affordable.

Figure 9-1: First packaged household soap

The transition to mass production had its own cost, not least that semiskilled factory labour may not be as fulfilling as the more varied craftsman role that it displaced. As well as social cost, the typical operation can reach bottlenecks in efficiency. The method of lean production pioneered by Toyota in the 1950s retains many aspects of mass production (efficiency, automation, and high volume of production) but instead of producing masses of a single part, assembly, or finished product, can be run to

produce them to order, at a specified date. This approach allows the company a much greater degree of customization than does mass production, and the emphasis on continuous improvement of efficiency is believed to lead to a more fulfilling and varied environment for the factory worker.

In other areas, the ethic of mass production resulted in new business models such as supermarkets, which pioneered both "self-service shopping" and the sale of a whole range of products under one roof. The first recognizable supermarkets appeared in the 1930s, evolved into the hypermarkets of the 1960s, and now the concept of self-service shopping has evolved to the automated tills where every shopper can be his own checkout assistant. Fast-food franchising began in the 1930s and exploded with McDonald's and Burger King in the 1950s. Standardized menus, pre-prepared ingredients, and standard practices for each franchisee to follow meant that you could now eat exactly the same meal in any of a chain restaurant's stores in your country (local tastes, laws, and religious observances mean that menus are tweaked globally).

In an interesting turn, many new fast-food chains are fighting against this movement by flavouring sauces made by hand from freshly sourced ingredients instead of mass-produced ones. The U.S.- based Chipotle chain was one of Fast Company's 50 most innovative companies of 2012. Here, injecting the ethical sourcing of food and more responsibility in the hands-on cooking for the employees into the successful fast-food business model has revitalized it.

Similarly, Lush created a soap empire in the 1990s by selling natural soap in long strips, to be cut into blocks by weight, just as it had been before Lever Brothers' intervention a century before.

9.1.3 The Long Tail of the Internet

As we have seen, huge changes in business practice are usually facilitated by, or brought about as a consequence of, technological change. One of the greatest technological paradigm shifts in the twentieth century was the Internet. From Tim Berners-Lee's first demonstration of the World Wide Web in 1990, it took only five years for eBay and Amazon to open up shop and emerge another five years later as not only survivors but victors of the dot-com bubble. Both companies changed the way we buy and sell things.

Chris Anderson of Wired magazine coined and popularized the phrase "long tail" to explain the mechanism behind the shift. A physical bricks & mortar shop has to pay rent and maintain inventory, all of which takes valuable space in the shop; therefore, it concentrates on providing what will sell to the customers who frequent it: the most popular goods, the "hits", or the Short Head. In comparison, an Internet storefront exposes only bits, which are effectively free. Of course, Amazon has to maintain warehouses and stock, but these can be much more efficiently managed than a public-facing shop. Therefore, it can ship vastly greater numbers of products, some of which may be less popular but still sell in huge quantities when all the sales are totaled across all the products. Whereas a specialist shop may or may not find enough

customers to make its niche sustainable, depending on the town's size and cultural diversity, on the Internet all niches can find a market.

Figure 9-2: Bricks & Mortar Shop v/s Internet Storefront

Long tail Internet giants help this process by aggregating products from smaller providers, as with Amazon Marketplace or eBay's sellers. This approach helps thousands of small third-party traders exist, but also makes money for the aggregator, who don't have to handle the inventory or delivery at all, having outsourced it to the long tail.

E-books and print-on-demand are also changing the face of publishing with a far wider variety of available material and a knock-on change in the business models of writers and publishers that is still playing out today. Newer business models have been created and already disrupted, as when Google overturned the world of search engines, which hadn't even existed a decade previously. Yet although Google's stated goal is "to organize the world's information and make it universally accessible and useful", it makes money primarily through exploiting the long tail of advertising, making it easy for small producers to advertise effectively alongside giant corporations.

9.1.4 Learning From History

We've seen some highlights of business models over the sweep of human history, but what have we learnt that we could apply to an Internet of Things project that we want to turn into a viable and profitable business? First, we've seen that some models are

ancient, such as Make Thing Then Sell It. The way you make it or the way you sell it may change, but the basic principle has held for millennia. Second, we've seen how new technologies have inspired new business models. We haven't yet exhausted all the new types of business facilitated by the Internet and the World Wide Web. If our belief that the Internet of Things will represent a similar sea change in technology is true, it will be accompanied by new business models we can barely conceive of today. Third, although there are recurring patterns and common models, there are countless variations. Subtle changes to a single factor, such as the manufacturing process or the way you pay for a product or resource, can have a knock-on effect on your whole business. Finally, new business models have the power to change the world, like the way branded soap ushered in mass consumerism and mass production changed the notion of work itself.

9.2 The Business Model Canvas

Video URL: https://www.youtube.com/watch?v=QoAOzMTLP5s

Video 9-2: Business Model Canvas

One of the most popular templates for working on a business model is the Business Model Canvas by Alexander Osterwalder and his startup, the Business Model Foundry. The canvas is a Creative Commons–licensed single-page planner. At first sight, it looks as though each box is simply an element in a form and the whole thing could be replaced by a nine-point checklist. However, the boxes are designed to be a good size for sticky notes, emphasizing that you can play with the ideas you have and move them around. Also the layout gives a meaning and context to each item.

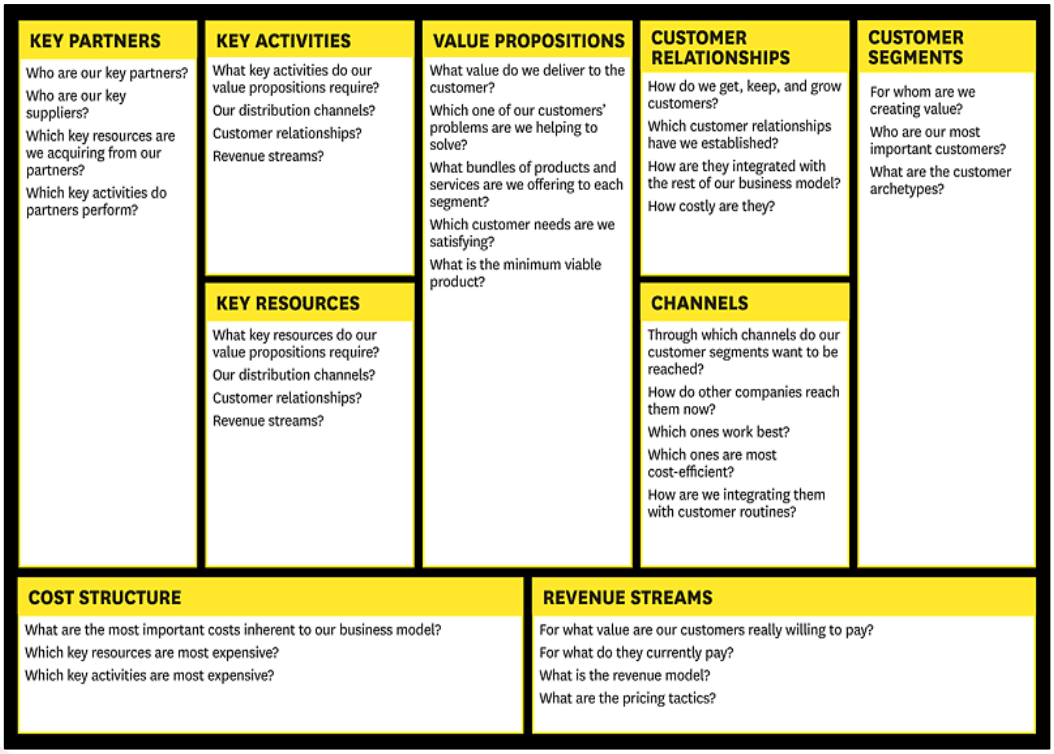

Figure 9-3: The Business Model Canvas

Let's look at the model, starting with the most obvious elements and then drilling down into the grittier details.

- At the bottom right, we have **Revenue Streams**, which is more or less the question of "how are you going to make money?"
- The central box, **Value Propositions**, is, in plainer terms, what you will be producing—that is, your Internet of Things product, service, or platform.
- The **Customer Segments** are the people you plan to deliver the product to. That might be other makers and geeks, the general public, families, or businesses.
- The **Customer Relationships** might involve a lasting communication between the company and its most passionate customers via social media. This position could convey an advantage but may be costly to maintain. Maintaining a "community" of your customers may be beneficial, but which relationships will you prioritize to keep communicating with your most valuable customer segments?
- **Channels** are ways of reaching the customer segments. From advertising and distributing your product, to delivery and after-sales, the channels you choose have to be relevant to your customers.
- On the left side, we have the things without which we have no product to sell. The **Key Activities** are the things that need to be done. The Thing needs to be manufactured; the code needs to be written. Perhaps you need a platform for it to run on and a design for the website and the physical product.

- **Key Resources** include the raw materials that you need to create the product but also the people who will help build it. The intellectual resources you have are also valuable, as are the finances required to pay for all this!
- Of course, few companies can afford the investment in time and money to do all the Key Activities themselves or even marshal all the Key Resources. You will need **Key Partners**, businesses that are better placed to supply specific skills or resources, because that is their business model, and they are geared up to do it more cheaply or better than you could do yourself. Perhaps you will get an agency to do your web design and use a global logistics firm to do your deliveries. Will you manufacture everything yourself or get a supplier to create components or even assemble the whole product?
- The **Cost Structure** requires you to put a price on the resources and activities you just defined. Which of them are most expensive? Given the costs you will have, this analysis also helps you determine whether you will be more cost driven (sell cheaply, and in great volume via automation and efficiency) or more value driven (sell a premium product at higher margins, but in smaller quantities).

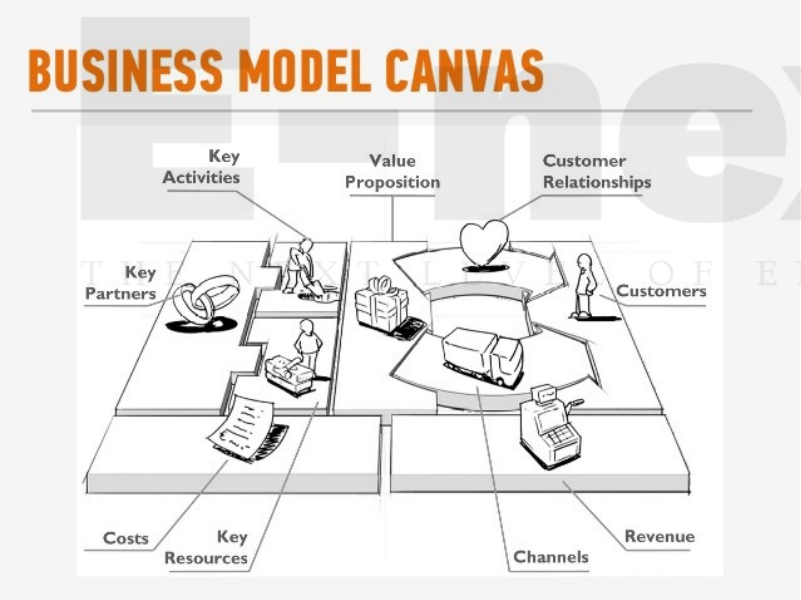

Figure 9-4: The Business Model Canvas

9.3 Who is the Business Model for?

Primarily, the reason to model your business is to have some kind of educated hypothesis about whether it might deliver what you want from it. Even if you don't use a semi-formal method like the canvas, anyone who starts up any business will have thought, at least briefly, about whether you can afford to do it, what the business is, and whether you'll get paid. As a programmer or a maker, you might believe it counter intuitive to think of a piece of paper with nine boxes in it as a "tool", but when you have a well-tested separation of factors to consider, the small amount of

structure the canvas provides should help you think about the business and give you ways to brainstorm different ideas. Many great product ideas turn out to be impractical, ahead of their time, or unprofitable. Being able to analyse how the related concepts mesh will help you challenge your product idea and either make it stronger or know when to abandon it.

The model is also useful if you want to get other people involved. This could be an employee or a business partner…or an investor. In each of these cases, the other parties will want to know that the business has potential, has been thought out, and is likely to survive and perhaps even go places. With a new business startup, you have no track record of success to point to. Although what will sell the business is primarily the product itself, and of course your passion for it, being able to defend a well-thought-out model of the business is an important secondary consideration for someone who is planning to sink time and perhaps money into your business. Perhaps to a lesser extent, your customers will also be considering whether to invest their time and money in your product. They will ask themselves certain questions about it. Let us look at some of these likely questions, from the wider field of Internet products in general.

- Why should I waste time trying out Yet Another Social Network? I think I'll wait and see whether all my friends join it first.
- This first question is about your "Value Proposition" and a reasonable concern if you are trying to get into a market that already has good or popular solutions.
- Will my Internet-connected rabbit become an expensive paperweight if you go bust?

This question is asked with a degree of consumer savvy about business risk. Potential customers have seen other companies fall under and don't want the inconvenience or waste it entails for them. Your online document collaboration looks great, but is it worth my moving my whole business to it? If you stop trading or change the platform, we may have to redo all the work again. Such customers may well be interested in the details of your business model to calculate whether the risk they've identified is worth their commitment. This isn't just a concern about viability of the company. This free service is fantastic, but why don't you let me pay for it, so I can get consistency, receive support, and avoid adverts? Lastly, many customers are aware of alternative charging models that they would prefer and might prefer a different one.

As Cegłowski says, "You don't really know that the cool project you signed up for is in a skyscraper in Silicon Valley, or like me: one dude in his underpants somewhere who has five windows open to terminal servers". But partners, investors, and informed customers will want to know. The business model provides, among other things, a useful tool for understanding what plans there are to keep the service running in both cases.

It has been stated about "free" products: "If you're not paying for something, you're not the customer; you're the product being sold". This formulation was popularized by Andrew Lewis in 2010 but builds on a long line of commentary about

consumerism. But as elegant as such a phrase may be, is it true? Derek Powazek, CEO of social startup Cute-Fight, challenges several assumptions often made:

- Not paying means not complaining.
- You're either the product or the customer.
- Companies you pay treat you better.
- So startups should all charge their users.

9.4 Models

The Business Model Canvas is a tool for generating and analyzing models. It is a good idea to have a look at some of the models that Internet of Things companies have used or might use and consider some of the parameters these models relate to on the canvas.

9.4.1 Make Thing, Sell Thing

The simplest category of models, "make a Thing and sell it", valid for the Internet of Things. Electrical products sold in shops (physical or online) may be subject to legislation and certification (RoHS, Kitemarks, and so on), which is an additional factor and cost to consider. Many small-scale projects take the option of selling the product in "kit" form, with some assembly required. Because kits are assumed to be for specialists and hobbyists rather than the general public, the administrative burden may be lower. However, making a decision to limit your target market may well limit the potential revenue also.

9.4.2 Subscriptions

A Thing would be a dumb object if it weren't for the important Internet component which allows the device to remain up to date with useful and current content. But, of course, this ongoing service implies costs to the provider—development, maintenance of servers, hosting costs, and in some cases even connection costs. A subscription model might be appropriate, allowing you to recoup these costs and possibly make ongoing profit by charging fees for your service. Many products could legitimately use this method, but perhaps the more complex, content-driven services would find it more convincing. People happily pay subscriptions to music services, corporate groupware, and of course, mobile phones, so perhaps Internet of Things products in these spaces will find subscription more appealing to their consumers. The so-called freemium model (a portmanteau of "free" and "premium") has always been a way to encourage paying customers while not alienating free ones. In this model, a smaller or larger part of your product is free, while the users are also encouraged to pay a

premium to get additional features or remove limits. This model could be combined with our first two models: Buying the physical device gives free lifetime access to the associated Internet service, but additional paid services are also available.

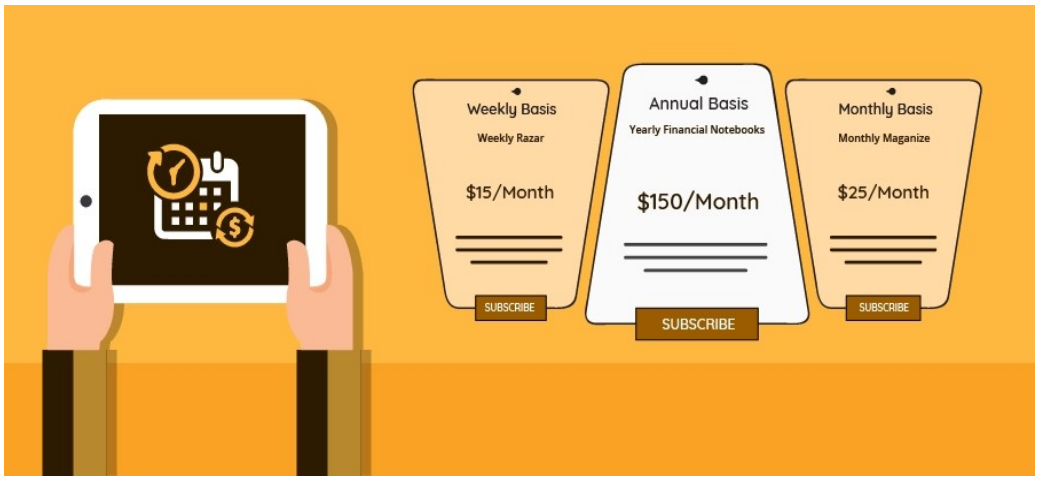

Figure 9-5: Subscription Model

9.4.3 Customization

For an Internet of Things device, at the intersection between solid thing and software, there are options for customization that we believe may lead to new business models. For a mass-produced item, any customization must be strictly bounded to a defined menu: a selection of different colours for the paintwork, options for fittings such as tyres, the trimmings and upholstery inside, and for features like the onboard computer control and display. Fordian logic dictates that all these components must be optimized for manufacture and fit well together. The world of software is, by contrast, pathologically malleable, if we let it be. Early websites explored the new medium of HTML to its garish extremes, with <blink> tags and animated .gif images. Yet today's equivalent of home pages, offered by incumbents such as Facebook, Twitter, and Pinterest, offer small degrees of customization within strictly defined boundaries: a selection of (tasteful) colour schemes and a choice of image to use as your avatar. Many Internet of Things products have some possibility of customization. The new manufacturing techniques, such as laser cutting and 3D printing, should allow great possibilities for customizing even the physical devices. MakieLab make dolls that can be designed online. Built to your specification, they are therefore unique and entirely yours in a way that a mass-produced doll couldn't be.

9.4.4 Be a Key Resource

Not every Internet of Things business will be selling a product to the mass market. Some will sell components or expertise to other companies—that is, component manufacturing or consultancy services. Effectively, in this kind of business, you are

positioning yourself as a “key resource” or a “partner” in somebody else’s business model. Small companies such as Adafruit and Oomlout sell electronic components to hobbyist makers. Manufacturers produce printed circuit boards (PCBs) and other custom electronics for the producers of gadgets such as Things. On the consultancy side, work will be available either simply providing your skills for hire or indeed in providing vision and expertise for strategic planning to a company that wants to engage with the Internet of Things.

9.4.5 Provide Infrastructure: Sensor Networks

Sensor data is a fascinating topic in the Internet of Things: Although there are official data sources, often very accurately calibrated and expensive to create, they may be hard to access and of course can exist only where a government body or company has chosen to apply its large but finite resources. The long tail of third-party data sensor enthusiasts can supplement or sometimes outclass the official streams of information. What is needed is a platform to aggregate that data, and one of the companies competing to fulfil that role is Xively. They allow any consumer to upload a real-time feed of sensor data and for the data from many feeds to be mapped, graphed, and compared. Xively have, since the beginning, intended to provide a free, public infrastructure for open source data while also providing enhanced commercial offerings with enhanced capacity and privacy options and formal service level agreements. Sensor data is information, which can be shared freely or might simply be sold. Many energy suppliers are rolling out “smart meters”, which promise greater efficiency and therefore cheaper bills but also aggregate huge quantities of information. As regards the business model, you need to consider the legality of such collection (now and in the foreseeable future) and whether it fits with your company values.

9.4.6 Take a Percentage

In the example of sensor networks, if the value of the data gathered exceeds the cost of the physical sensor device, you might be able to provide that physical product for free. In fact, energy companies quite often do this with their smart meters. Even without charging the end user of your Internet of Things device, there will be many options to make a profit from somewhere (ad revenues, payment for data services from companies or state organizations, commission for data bandwidth incurred, etc.). Within the burgeoning field of the Internet of Things, exactly what the “product being sold” consists of is a field that remains to be explored.

9.5 Funding an Internet of Things Startup

The problem of how to get initial funding is a critical one, and looking at several options to deal with it is worthwhile. If you have enough personal money to concentrate on your new Internet of Things startup full time without taking on extra

work, you can, of course, fund your business yourself. Apart from the risk of throwing money into a personal project that has no realistic chance of success, this would be a very fortunate situation to be in. And luckier still if you have the surplus money to bankroll costs for materials and staff. If you aren't Bruce Wayne, never fear; there are still ways to kick off a project. If the initial stages don't require a huge investment of money, your time will be the main limiting factor. If you can't afford to work full time on your new project, perhaps you can spare a day in the weekend or several evenings after work. Many people try to combine a startup with a consultancy business, planning to take short but lucrative contracts which support the following period of frantic startup work.

9.5.1 Hobby Projects and Open Source

If your project is also your hobby, you may have no extra costs than what you would spend anyway on your free-time activity. This is perfectly valid, and to turn your product into a successful business or community, you may wish to proceed at a less leisurely pace than a pure hobby might entail. One way to make a project grow faster might be to release all the details as open source and try to foster a community around it. This approach can be hard work and can benefit from a natural talent, experience, or luck in attracting and maintaining good collaborators. After you have open-sourced a project, you can't close-source it again. Yes, you can probably fork the project and continue to work on it in secret, but the existing project may carry on if your collaborators are enthusiastic enough about it. Indeed, your idea, code, and schematics could be used by others in their own commercial offering. Careful consideration of the license used may be critical here: A more restrictive license such as the GPL requires those who build on your work to share their source code also under the same terms. Running an open source project takes work, and the risk of losing control of your project may not be for everyone, but it is certainly an option to consider.

9.5.2 Venture Capital

Getting funding for a project from an external investor presents its own work and risks. The process of applying for funding takes time, and although much of this time can be justified as thrashing out the business model, it's not directly related to the work you actually want to be doing on the product itself. Startups often concentrate their fundraising activities into rounds, periods in which they dedicate much of their effort into raising a target amount of money, often for a defined step in their business plan. Before any official funding round comes the informal idea of the friends, family, and fools (FFF) round. Although it's important to consider the possible impact on your personal relationships, this round of funding may be the most straightforward to get hold of. A common next step would be an angel round. The so-called angels are usually individual investors, often entrepreneurs themselves, who are willing to fund some early-stage startups which a more formal investor might not yet touch. The reason might be that these angels have a technical or business background in your product or simply that, as individual investors, they may have more scope to go with

their own intuition about your worth. The personal interest and experience that angels can bring to your company means that their advice, contacts, and other help may well be as useful as any money they provide. They usually want equity in your company, a percentage of the value of the company, that will pay back their investment if and when you do well. These angels might also demand a place on your board of directors, to oversee their investment, but also out of interest in helping the company to succeed. The venture capital (VC) round is similar, but instead of your courting individual investors, the investor is a larger group with significant funds, whose sole purpose is to discover and fund new companies with a view to making significant profit. VCs may be interested if angels have already funded you and will certainly be interested if other VC companies are already looking at funding you. VCs will certainly want equity, probably a significant amount of it, and a position on your board of directors. Again, this last role may be as much to help fill gaps that your management team don't cover as much as it is to keep an eye on you and their money. A related option, especially in the early stages, would be an accelerator, which might be run by a venture capital firm. In this case, part or all of the money that could be awarded to your company is paid in kind, in the form of free office space, consultancy, and specific training and mentoring in areas that the investor believes will make you a success. The fact of being collocated in an incubator with other smart new companies may be of great benefit, and the training and contacts you gain could well be valuable. Even though funding may sound like "free money", we've already seen that getting investment comes with conditions: equity and some measure of control via your board of directors. The trade-off is obvious: You couldn't get the money to grow your business otherwise. Even if losing some control in the company is heart-wrenching, a smaller percentage of a valuable company will be worth more than a large percentage of nothing.

We've seen some of the considerations in funding. In addition, you need to be aware that by accepting investment through venture capital, you are committing yourself to an exit. An exit strategy is a "method by which a venture capitalist or business owner intends to get out of an investment that he or she has made". Typically, you have only two exits:

- You get bought by a bigger company.
- You do an IPO (initial public offering)—that is, float on the stock market.

9.5.3 Government Funding

Governments typically want to promote industry and technological development in their country, and they may provide funds to help achieve particular aims. Although governments can and do set up their own venture capital funds or collaborate with existing funds in various ways, they generally manage the majority of their funds differently. The money provided still has "strings attached", but they are likely to be handled differently:

- Outputs: Deliverables (aka outputs) are the metrics that an awarding body may use to tell if you are doing the kind of thing that the body wants to fund. This metric may simply be a test that you are managing the money well or may be related to the goals that the body itself wishes to promote. You might be required to write regular reports or pass certain defined milestones on schedule.
- Spending constraints: Some funding may require you to spend a proportion of the money on, for example, business consultancy or web development, perhaps with the fund facilitator's company or associates. This requirement may be highly valuable, of course, but the practice appears slightly misleading; it would be better to clarify this as a funding "in kind" rather than cash.

9.5.4 Crowdfunding

Getting many people to contribute to a project isn't exactly a new phenomenon. Walter Gervaise built the first stone bridge over the River Exe in 1238 through public subscription by approaching friends and other wealthy citizens. Earlier, after 27 BC, Augustus Caesar sponsored a public subscription for a statue of his physician Antonius Musa. Over millennia many civic and religious monuments and constructions have been funded at least partly by the public. However, such projects have been mostly sponsored and given focus by some influential person or body. With the efficiencies of the Internet's long tail, people can seek funding to print limited editions of their latest comic book, CDs of vocal-only covers of every song by the Smiths, or perhaps your exciting new Internet of Things product. As of 2013, the main options for crowdfunding are Kickstarter and Indiegogo.

Indiegogo is open to all types of projects, including community and charitable ones, whereas Kickstarter is only for those creative projects that end up with a product, be it artistic or technological. Your funders are real people and will have all the variety of concerns and foibles that any group of real people have. This interaction with a large and diverse group is a key part of the interest of this method of funding: It is far more than just the money. Crowdsourcing allows you to find consumers who are interested in the product even before you invest time and money in the project.

Figure 9-6: Crowdfunding

9.6 Lean Startups

The mentality of running a startup on a low budget includes spending time and money only when it's really necessary—staying hungry and lean. The concept of a "lean startup," pioneered by Silicon Valley entrepreneur Eric Ries, springs from this idea. The option in the preceding section of crowdfunding a project presented an even more appealing step on this route: running the project only if there is a demonstrable niche market for it. Many lean proponents suggest setting up a landing page for a project with a simple form to register interest. This is quick and simple to do, especially as numerous startups do exactly this. These simple pages allow you to propose many projects and focus only on the ones that have most feedback. However, if you've already done some prototyping work and have a good feeling about a single idea, taking things a step further and creating a project on a crowdfunding site may be even more appropriate. Doing so represents more work than creating a simple form, but you will learn far more from it! In many ways, this "laziness"—doing the minimum now and putting off the hard work till later—is also the reason to have a prototype separate from the final product. There is a time to market your project, a time to ensure that the idea works, and a time to build a sellable product.

If you are thinking "lean", you should be applying this idea at all stages. For example, at the first stages of production and marketing, you should be working towards the "Minimum Viable Product". This is still a sellable product rather than a prototype, but with all extraneous features removed, it may feel like a prototype of your final vision for the product. All the initial efforts are towards making this product because it can be sold. If you have time and money afterward to add additional enhancements to the product, service, packaging, and so on, this will add more value. But adding those enhancements to an incomplete prototype would not result in a working business model. The essence of lean is to be able to iterate, performing the tasks that are required to get things moving at this stage, without investing time upfront to make everything perfect. The fact that your business model is a hypothesis and not set in stone can encourage you to tweak it in response to the feedback you get from iterating your product in the real world. Such tweaks are known as pivots and usually work by changing one part of your model—think one of the boxes on the Business Model Canvas. For instance:

- Zoom-in pivot: Focus on what was only a part of the value proposition, and turn that into the whole Minimum Viable Product.
- Customer segment pivot: Realize that the people who will actually buy your product aren't the ones you were originally targeting. While you can continue to make exactly the same product, you have been marketing it to the wrong people.
- Technology pivot: Accomplish the same goals as before, but change the implementation details. While prototyping will almost certainly involve many changes in technology while you establish the best way to make the product from

an engineering perspective, this pivot would be a business decision, made to improve manufacturing costs, speed, or quality.

9.7 Summary

A business model is a hypothesis about how to run a project well, for commercial profit or some other success criteria, to develop a product that solves problems for a specific group of users. Throughout history, people have invented new ways of doing business; new technology is the factor that is most likely to bring about entirely new models. Faced with the certainty that the Internet of Things, as a technological paradigm shift, will facilitate entirely unexpected new business models, it is increasingly important that you analyse, discuss, and iterate your own model. It is, after all, only a hypothesis and can be changed in the face of the existing evidence. This is one of the important factors of the "lean" approach. In the fast-moving, competitive, and increasingly business- as well as tech-savvy world of the Internet, it is vital to be able to show your business model to potential investors, partners, and customers. The Business Model Canvas is one useful set of shared categories and terminology that may facilitate your communication and discussion with these groups. This is important because if you are planning to scale up your project into a product, making your vision a reality will rely on their involvement. We looked in particular at investment, from friends and family to angels, government funds, venture capital, crowdfunding and accelerators.

9.8 Review Questions

1. Write a short note on the history of Business models.
2. Explain the business model canvas.
3. Write a short note on various models.
4. Write a short note on different ways to fund an Internet of Things startup.
5. Write a short note on government funding.
6. Write a short note on crowdfunding.
7. Why are lean startups a good idea for an IoT startup?