IOT
Chapter 8
TECHNIQUES FOR WRITING EMBEDDED CODE

Q1) Explain MEMORY MANAGEMENT issues in Embedded code?

- 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.
- 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 that 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.
- **The different types of memory you might encounter are:**
- **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.
- **Flash**
- Flash is a semi-permanent type of memory which provides all the advantages of ROM, 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.
- 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.
- **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.
- You can also provide hints to the compiler 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.
- It's much better to get this storage out of RAM and into flash.
- 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"));

Q2)How to make THE MOST OF YOUR RAM ?

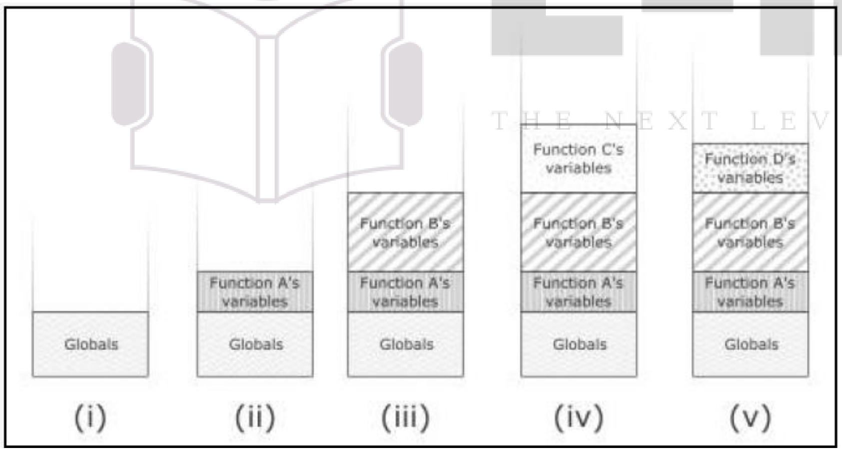
- Now that you've moved everything that you can out of RAM and into flash, all that remains is to work out ways to make better use of the free memory you have.
- When you have only a few kilobytes or tens of kilobytes of RAM available, it is easier to fill up that memory, causing the device to misbehave or crash.
- Yet you may want to use as much of the memory as possible to provide more features.
- There is a 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.
- For example when you're downloading some information from the Internet how could you possibly know beforehand exactly how large it is going to be?
- 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 while 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.
- 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.
- 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.
- 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.

Q3) Explain Organising RAM: Stack versus Heap.

- Two general concepts for arranging memory are used: the stack and the heap.
- 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.
- The processor 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.
- 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.

- 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()
```

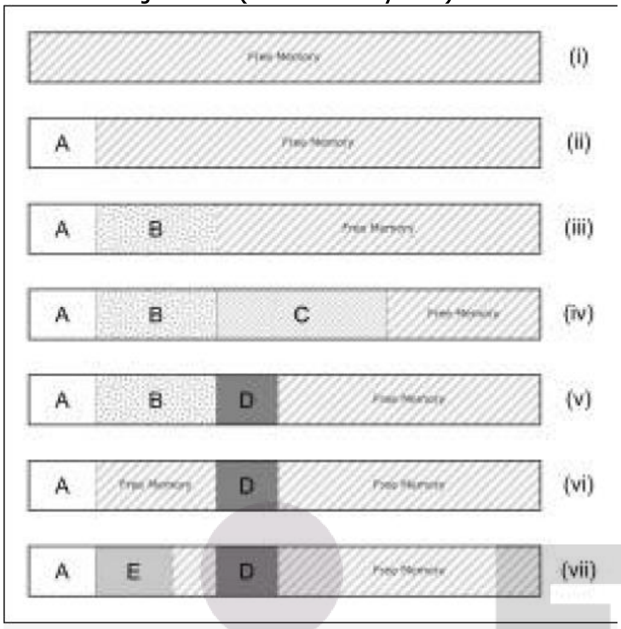


HEAP

- 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.
- 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.

- Following 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)
```



- *Deterministic memory usage suggests* 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 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 , it could outgrow this fixed space.
- One way to reduce the chance of this situation occurring is to keep the number of global variables to a minimum.
- 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.
- 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.

Q4) Explain the concept of 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.
- 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.
- Similarly, if you're building something which doesn't have to react instantly 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.
- 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.
- 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.
- But if 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, Bubblino 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 Bubblino device.
- The downside to that is you would need to write another service and maintain infrastructure to support it.

Q5) list and explain habits that help make your code generally more efficient (to increase performance).

- **Following are a few habits that help make your code generally more efficient:**
- **1]W**hen 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.
- 2] Declaring data as constant where it is known never to change can help the compiler to place it into flash memory or ROM.
- 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.
- 3] 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.
- 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.
- 4] 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, use processor instructions that copy more than one byte in a single operation, thus considerably speeding up the process.

Q6)Why we need LIBRARIES? List few libraries available.

- 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
- 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 its better finding one that's already written with microcontroller limitations in mind.
- Here are a few libraries that are available:
- ▪ **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.

- ▪ **uClibc:** uClibc is a version of the standard GNU C library (glibc) targeted at embedded Linux systems.
- It requires far fewer resources than glibc.
- 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.
- ▪ **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.

Q7) Explain various methods of DEBUGGING software and hardware.

- 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 integrated development environments (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.
- 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.
- 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.
- 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.
- JTAG has been extended to provide more advanced debugging features such as 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.
- 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.

Q8) what is Logging? Explain issues with logging.

- 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.
- Issues with this are:
- 1] 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.
- 2] 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.
- 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.
- 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.
- 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.
- 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 can be: 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.