Chapter 2: Channel Embedded Systems

2. Introduction

Channel Embedded Systems are wide and varied propositions to differing people. To anyone who has been taking care of servers, a credit card application developed for any phone is an embedded system. To somebody who has written code for tiny 8-bit microprocessors, anything with the operating system doesn't seem much embedded. To help explain Channel Embedded Systems to all non-technical people, they are things like microwaves and automobiles that run software but aren't computers. Most people recognize a pc as a general purpose device. Perhaps a good way to define the word without haggling over technology is: An embedded system is a computerized system that may be purpose designed for its application.

Because its mission is narrower than a general purpose computer, an embedded system has less support for issues that are unrelated to running the application. The hardware often has constraints: e.g., a CPU that runs more slowly to avoid wasting battery; a system that utilizes less memory so it could be manufactured more cheaply; processors which come only using some speeds or support a subset of peripherals. The hardware isn't the only main system with constraints. In a few systems, the software must act deterministically (similar whenever) or real-time (always reacting with an event fast enough). Some systems require that this software be fault tolerant with graceful degradation industry by storm errors. E.g., think about a system where servicing faulty software or broken hardware may be infeasible (i.e. a satellite or a tracking tack on a whale). Other systems require how the software cease operation at the very first hint of problems, often providing clear error messages (a cardiac monitor should not fail quietly).

2.1 Compilers, Languages, and Object-Oriented Programming

An additional way to identify Channel Embedded Systems is that they use cross compilers. While a cross compiler runs on your desktop or notebook, commemorate
code it doesn't. The cross compiled image is run on your target embedded system. Since the code are powered by your processor, the seller for the target system usually sells a cross compiler or comes with a set of available cross compilers to pick from. Many larger processors utilize the cross compilers from the GNU class of tools. Embedded software compilers often support only C, or C and C++. Furthermore, many embedded C++ compilers implement simply a subset in the language (commonly, multiple inheritance, exceptions, and templates are missing). There is a growing popularity for Java, even so the memory management inherent for the language works only with a larger system.

Depending on the language one utilizes as part of your software, you'll be able to practice object oriented design. The planning principles of encapsulation, modularity, and data abstraction may be given to any application in nearly any language. The goal is always to make the design robust, maintainable, and versatile. We must employ all what we can get from the object-oriented camp.

Taken in its entirety, an embedded system can be viewed as similar to an object, particularly one which works in a larger system (e.g. a remote control speaking with a set top box, a distributed control system inside a factory, an airbag deployment sensor in a vehicle).

Inside the higher level, things are inherently object-oriented, which is logical to give this into embedded software.

However, it is not recommend a strict adherence to all object-oriented design principles. Channel Embedded Systems get drawn in a lot of directions to set down this type of commandment. When one finally recognizes the trade-offs, he is able to balance the application design goals and also the system design goals.

### 2.2 Embedded System Development

Channel Embedded Systems are special, offering special challenges to developers. Most embedded software engineers produce a toolkit to help with the constraints. Before we can start discussing ours, here are the difficulties related to developing an embedded system. As soon as it is comprehended how the embedded system may be limited, we can start on some principles to guide us to higher solutions.
Should someone wish to debug software on a computer, he can compile and debug it on computer. The system can have plenty of resources to perform the program and support debugging it concurrently. Actually, the hardware wouldn't know that someone is debugging software, because it is all designed in software.

Channel Embedded Systems aren't like that. Plus a cross compiler, you'll need a cross debugger. The debugger sits on your desktop and 'talks' to the target processor from the special processor interface (see Figure below). The interface is focused on letting another person eavesdrop within the processor since it works. This interface can often be called JTAG, whether it actually implements that widespread standard or not.

![Fig. No. 2.1 Computers and Target Processor](image)

The processor must expend most of its resources to aid the debug interface, allowing the debugger to halt and run and supply the normal forms of debug information. Supporting debugging operations adds cost to the processor. To maintain costs down, some processors support a restricted subset of features. For example, adding a breakpoint causes the processor to change the machine code to express “stop here.” However, if your code is programmed to flash (or any other form of read-only memory), rather than modifying the machine code, the processor must set an interior register (hardware breakpoint) and compare it at intervals of execution cycle on the address being run, stopping if they match. This can alter the timing of the code, bringing about annoying bugs that occur only if one might be (or perhaps aren't)
debugging. Internal registers take up resources too, so there are often simply a small group of hardware breakpoints available.

So one can say that, processors support debugging, but is not always the maximum amount of debugging since any one from the software world would know. The unit that communicates relating to the PC and also the processor is often called an emulator, an in-circuit emulator (ICE), or possibly a JTAG adapter. These may refer (somewhat incorrectly) to the ditto or they are often three different devices. The emulator is specific for the processor (or processor family), and that means one can't go ahead and take emulator one has for just one project and assume it will eventually work on another. The emulator costs add together, notably if someone collects an ample amount of them or for those who have a substantial team fixing your system.

To avoid buying an emulator or handling the processor limitations, many Channel Embedded Systems are made to have their debugging primarily done via printf or some type of lighter weight logging for an otherwise unused communication port. While incredibly useful, this will also customize the timing of the system, possibly leaving some bugs to get revealed only after debugging output is powered down.

Writing software for an embedded system can be tricky, as one has to balance the needs on the system and also the constraints with the hardware. Now there is another item in the to-do list: making the software debuggable inside a somewhat hostile environment.

### 2.3 Additional Issues

An embedded system is built to start a specific task, removing the resources doesn't necessarily have to accomplish its mission. The resources include:

- Memory (RAM)
- Code space (ROM)
- Processor cycles or speed
- Battery life (or power savings)
- Processor peripherals

Somewhat, these are generally interchangeable. For example, one can trade code space for processor cycles, writing areas of your code to take up more space but run faster. Or one will lessen the processor speed to be able to decrease power
consumption. If someone does not have a certain peripheral interface, one might be capable to create it in software with I/O lines and processor cycles. However, despite the presence of trading off, one has merely a limited supply of each resource. The process of resource constraints is one of the most obvious for Channel Embedded Systems.

Another number of challenges come from working with the hardware. The additional burden of cross-debugging can be frustrating. During board raise up, the uncertainty of whether a bug is within the hardware or software will make issues hard to solve. Unlike the pc, the software program one writes may be able to do actual harm to the hardware. Best of all, it should be stated in regards to the hardware and the achievements competent at. That knowledge will not be applicable to another location system the programmer is working on. The programmer may be challenged to understand quickly.

Once development and testing are finished, the system is manufactured, something most pure software engineers never need to take into consideration. However, creating a system that could be manufactured for the reasonable expense is a mission that both embedded software engineers and hardware engineers ought to consider. Supporting manufacturing is why you can make sure that the system that you just created gets reproduced with good fidelity.

After manufacture, the units go into the field. With consumer products, this means they're going into a lot of homes where any bugs you created are enjoyed by many. With medical, aviation, or other critical products, your bugs might be catastrophic. With scientific or monitoring equipment, the field can be quite a place the spot that the unit cannot ever be retrieved (or retrieved only at great risk and expense consider the devices in volcano calderas) so that it ought to work. The life a system will lead after it leaves the programmer can be a challenge one needs to consider as the program is designed.

Once the programmer has established all of these issues and determined how to approach them for the system, there is certainly still the biggest challenge, one present with all branches of engineering: change. Nearly the item goals change, the requirements of the project change through its lifespan. In the beginning, one may like to hack something together, only to give it a try. As the programmer grows older and better understands and defines the goals of the product and also the hardware he might be using, he begins to construct more infrastructure to generate it debuggable, robust
and flexible. Within the resource constrained environment, the programmer will have to figure out how much infrastructure he can afford in terms of development time, RAM, code space and processor cycles. That which he started building initially isn't what he would actually end up with when development is complete. And development is rarely ever complete.

Creating a system that may be purpose intended for software comes with an unfortunate side-effect: the system may not support change since the application morphs. Engineering Channel Embedded Systems isn't just about strict constraints as well as the eventual lifetime of the system. The challenge is working out which of those constraints might be a problem later in product development. One has got to predict the likely course of changes and attempt to design software flexible enough accommodate whichever path the appliance takes.

2.4 Principles to Take on Those Issues

Channel Embedded Systems can seem just like a jigsaw puzzle, with pieces that interlock. Sometimes it is possible to force pieces together, but the resulting picture might not be what is about the box. However, we must abandon the idea of a final result as a single version of code shipped by the end in the project.

Instead, think of the puzzle carries a time dimension which shows the actual way it varies over its very existence: conception, prototyping, board boot, debugging, testing, release, maintenance, and repeat. Flexibility is not just about what the code can do at this time, but also precisely the code are designed for its lifespan. Our goal might be flexible enough to match the product goals while coping with the resource constraints along with other challenges inherent to Channel Embedded Systems.

There are several excellent principles the programmer can easily take from software design to generate the system more flexible. Using modularity he separates the functionality into subsystems and hides the data each subsystem uses. With encapsulation, he creates interfaces between the subsystems so that they don't know much about the other person. As the programmer has loosely coupled subsystems (or objects, when you prefer), he can change one region of software with confidence it won't have an impact on another area. Allowing him to break down the system and put it together again a bit differently whenever he has to.
Recognizing where you can break up a system into parts takes practice. An excellent rule of thumb is to consider which parts can transform independently. In Channel Embedded Systems, this really is helped because of the presence of physical objects that you can consider. If your sensor X talks over the line Y, those are separate things and good candidates for being separate subsystems and code modules. If the programmer breaks things into objects, you can do some testing to them. It has been found that bugs caught before software releases can be like gifts. The earlier at the same time errors are caught, the cheaper they may be to solve and also the better it can be for everyone.

The programmer won't have to lose time waiting for another person to give him presents. Testing and quality go hand in hand. Writing test code for the system could make it better; provide some documentation for one’s code, to make people think you're writing great software.

Documenting the code is another strategy to reduce bugs. It is usually challenging be aware of degree of detail when commenting your code.

```
i++; // increment the index
```

Lines like this rarely need comments in any respect. The goal should be to write the comment for someone like the programmer checking this code a year from the time the code was actually written. By that point, in future the programmer will likely be working on something different and also have forgotten just what exactly creative solution did he came up with. It is likely the programmer doesn't even remember scripting this code, so he should feel free out having a little bit of orientation (file and function headers). Generally speaking, though, assume people could have your brains plus your general background, so document just what code does, not how it can it.

Finally, with resource constrained systems, there is a temptation to optimize your code early and sometimes. Fight the urge. Implement the features, get them to work, test them out, and then make them smaller or faster as needed.

"We ought to forget about small efficiencies, say about 97% of times: premature optimization is the reason for all evil." – Donald Knuth

You have only a limited period of time: target where one can progress results by searching for the larger resource consumers have got a working subsystem. It doesn't do you high quality to optimize a function for speed whether it runs rarely and is particularly dwarfed by the point spent in another function that runs frequently. To make certain, coping with the constraints of the system will require some
optimization. Associated with you recognize where your resources are used before you begin tuning.

2.5 Scheduling and Operating System Basics

Structuring an embedded system without an operating system requires an awareness of a number of the stuff an operating system are able to do for the programmer.

2.5.1 Tasks

After you switch on your personal machine, should you be like we are, you group the email program, internet browser, and compiler. Many programs start automatically for example my instant message client. Each one of these programs works on the computer, seemingly in parallel even when you've only got one processor.

Three words that mean slightly different things, but that overlap extensively are often used interchangeably. A job is something the processor does. A thread is an activity as well as some overhead such as memory. A process is generally a complete unit of execution unit with its own memory space, usually compiled separately from other processes. Let us restrict this discussion centering on tasks; threads and processes generally imply an operating system.

The operating system that we run has a scheduler that does the switching between processes (or threads), allowing each to operate to use proper turn. The main factor is a scheduler is aware of everything your system has to do and chooses which and also today. Lacking operating system, the programmer will have to do the scheduling himself.

2.5.2 Avoiding Race Conditions

The programmer should look for a way to prohibit multiple tasks from writing to identical memory. It isn't really only writing that may be an issue: the principle loop reads the two variables which say the button is modified along with the worth of the button. If the interrupt occurs between those two reads, it might change the importance of the button between them.
Any time memory shared between tasks is read or written, it makes a vital section of code, meaning code that accesses a shared resource (memory or device). The shared resource should be protected so just one task can modify it each time. This is known as mutual exclusion, often shortened to mutex.

Within a system with the OS, when two tasks are running, but neither can be an interrupt, the programmer will use a mutex to indicate which task owns a resource. This is often as easy as a variable indicating if the resource (or global variable) can be found to use. However, when one of the two tasks can be an interrupt, it is well known that doesn't do obviously any good, so this resource ownership change needs to be atomic. An atomic action is but one that cannot be interrupted by anything from the system. Operating systems have heavier weight mutexes called semaphores that are equipped for situations where a thread or process may be preempted forcibly changed through the scheduler.

From this level on, the discussion is going to concentrate on systems certainly where a task is interruptible but otherwise runs until it gives up control. If so, race the weather is avoided by disallowing interrupts while accessing the shared global variables. This has a down side, though: once you go out on interrupts, the system can't respond as quickly to button presses given it has to wait to leave a crucial section. Turning off interrupts boosts the system latency time it takes to react.

Latency is essential even as we talk about real-time systems. An actual-time system must interact with a meeting in a fixed amount of time. Although the required latent period is dependent upon the system, usually it really is measured in microseconds or milliseconds. As latency increases, the time it requires before a meeting can be noticed because of the system increases and the total time between a conference and its response increases.

### 2.5.3 Priority Inversion

Some processors allow interrupts to possess different priorities like operating systems do for processes. As the programmer moves, the flexibility can be extremely useful, and he possibly can avoid a wide range of trouble for himself. Figure below shows the typical operating systems definition of priority inversion. It really is okay for a high priority process to quit as it needs having access to something a minimal priority
process has. However, if your medium priority process starts, it could block over the priority process from completing its call time resource essential for high priority task. The medium priority task blocks the high priority task.

![Fig. No. 2.2 Priority Inversion](image)

By way of example, say the programmer has now the high priority button press interrupt and our low priority main loop. When the main function accesses the button press variable, it turns journey button press interrupt avoiding a race condition. Now, we add another interrupt to output debug data through a communication port. This should actually be a background task of medium priority: an issue that should have completed but shouldn't block the system from handling a button press. However, when it interrupts the leading loop, it can be doing exactly that.

What's the most crucial thing for the processor to be doing? That is its highest priority. Or otherwise, it should be the biggest priority. If someone else had asked whether debug output is more important than button presses, one might have said no. If it is true, how come the processor running the debug interrupt and not the button handler? The simple fix in this instance is usually to disable all interrupts or all interrupts which might be reduced priority than the button handler.
Even as we take a look at types of methods task management lacking any operating system, there will be more techniques for finding into the position the spot that the processor is inadvertently running a task that is not the very best priority.

### 2.5.4 State Machines

A great way to keep the system organized as you have more than something occurring is by using a state machine. This can be a standard software pattern, the one which Channel Embedded Systems work with a lot. According to Design Patterns: Portions of Reusable Object-Oriented Software, the intent with the State pattern would be to allow an object to change its behavior when it’s internal state changes. The object will appear to alter its class.

Put more simply, whenever you call a situation machine, it is going to do whatever it thinks it should do, according to its current state. It doesn't do the same every time, but will base the change of behavior on its context, which includes the environment along with the state machine's history (internal state). If this all sounds clinical and theoretical, it comes with an easier solution to think about state machines: flow charts. Virtually any state machine may be represented like a flow sheet. Conversely, a problem the programmer solves with a flow diagram is most likely to be a state machine. State machines are also represented as finite state automata as shown in Figure below, where each state can be a circle and also a change between states a state transition can be an arrow.

![State Machines Diagram](image)

**Fig. No. 2.3 State Machines**
We'll investigate each portion of this determine these section. The arrows within the diagram are only as important as the circles. State machines are not only seen around the states that the system can occupy, but also regarding the events that the system handles.

### 2.5.5 Choosing a State Machine Implementation

Any of the representations for the state machine affords the same functionality, even if the implementation differs from the others. If the programmer thinks about the implementation as being lazy, then choose the option leading on the least number of codes. If they are all about the same, pick the one with the least volume of replicated code. If another person’s implementation enables someone else to reuse a section of code without copying it, that's a better implementation for the system. If that still does not help the programmer choose, consider which kind of code is going to be the most easily read by another person.

State machines are powerful because they let the code act in line with its history (state) along with the environment (event). However, given that they react differently depending on those actions, they might be quite hard to take care of, resulting in spaghetti code and dependencies between states which aren't obvious to the casual observer. Documentation is key, which is why the discussion here has focused in these sections about the human-readable representation in the state machine before showing a code implementation.

### 2.5.6 Interrupts

The state machine doesn't care whether there exists a person pushing buttons that say “stop” and “go” or there is a wireless Ethernet controller parsing a data stream seeking these commands.

Interrupts can be rather scary. These are one thing which makes Channel Embedded Systems not the same as traditional software. Interrupts often appear to swoop in from nowhere to vary the flow from the code. They could only call certain functions (but not usually the debug functions). Interrupts should be fast, so fast likely a bit of code
that's still sometimes written in assembly. And bugs in interrupts are often very
difficult to locate because, by definition, they occur outside the normal flow of code.
However, it isn't the bogeyman they've been thought to be. In case if we try to
understand a little bit with what happens when an interrupt occurs, we can discover
where and how they can be a useful part of the software design.
Processors and interfaces are just like software APIs and that function pointers aren't
scary, one should keep this in mind as the discussion moves on to what are the effects
when an interrupt occurs:
1. Interrupt request (IRQ) happens, inside processor with different peripheral, the
   software, or maybe a fault from the system.
2. The processor saves where it had been (the context).
3. The processor looks within the interrupt vector table to search for the callback
   function of this particular interrupt.
4. The callback function (aka interrupt utility routine (ISR) or interrupt handler) runs.
   Some interrupts are just like small, high-priority tasks. Once their ISR is complete, the
   processor restores the context it saved and continues returning just as if nothing had
   happened. Most peripheral interrupts are just like that: input lines, communication
   pathway, timers, peripherals, ADCs, etc.
   However, many interrupts are definitely more like exceptions, handling system faults
   rather than going back to normal execution. By way of example, an interrupt can
   occur if you have a memory error, we have a divide by zero error, the processor tries
to execute an invalid instruction, or the electricity is not quite sufficient to operate the
   processor properly (brown out detection). Since these errors mean the processor can't
   run properly, they are often handled with infinite loops or processor resets.
   For most interrupts it is advisable to treat them to be more like tasks rather than
exceptions. They are going to let the processor run multiple tasks, seemingly in
parallel. Before the discussion moves to the next issue, first let's glance at the
procedure for interrupt handling in additional detail.

2.5.7 As IRQ Happens

Usually an interrupt request is reality because the programmer has configured the
interrupt. If this wasn't the programmer, then the compiler's startup code probably did
the job. While often invisible to dangerous language programmers, the startup code configures some hardware oriented things for instance establishing the default interrupts, usually the fault interrupts.

For task-like interrupts, your initialization code can configure the crooks to occur beneath the conditions you specify. What precisely those conditions are that have to be determined by the processor. The processor's user manual is going to be critical to creating interrupts.

Step one to setting up an interrupt is usually, somewhat oddly, disabling the interrupt. Despite the fact that part of the power-on sequence is to disable and clear all interrupts, it's sensible to consider the precaution anyway. If the interrupt is enabled, it will fire prior to the initialization code finishes configuring it properly, possibly producing an accident. Creating interrupts uses registers that are memory mapped. Accessing the memory address directly can make for illegible code. Most processor vendors and compiler vendors offer header files of #define statements, often enabling the programmer gain access to individual registers as members of structures.

### 2.5.8 Restore the Context

After the ISR has been completed, it is time to come back to normal execution. Some compilers C/C++ extend to incorporate an interrupt keyword to indicate which functions implement interrupt handlers. The processors give these functions special treatment both whenever they start (some context is saved prior to the ISR starts running) and when they return.

The program counter points on the machine instruction that is just about to run. Whenever the programmer calls a function, the address from the next instruction (program counter + 1 instruction) is put in the stack since as the return address. Whenever the programmer returns in the function, the program counter is determined to that address. It can be unusual for several assembly languages to have similar opcodes.

However, rtsandrtitend to become pretty common. They stand for return from Subroutine and Return from Interrupt respectively. However, the interrupt isn't a standard function call; this can be a jump towards interrupt handler brought on by the processor. If the interrupt simply returned that it turned out a function, things that the
processor did to store the context would not get undone. So interrupts employ a special instruction (rti) to point actually returning from an interrupt. This lets the processor realize that it has to restore the context to its state before the function call before continuing returning.

This difference between a call as well as a jump is a crucial one, notably if the programmer turns out doing any assembly programming. Iteration will help recall the difference: a call has context, an increase just goes. If the programmer uses go to as part of his code, then he’s executing a jump.

Yes, people do use go to in modern code. Used sparingly, they are able to become exception handlers, taking care of memory or a peripheral when products have gone catastrophically wrong during its use. They often replace code that starts that has a variable that tracks the error condition, getting checked at intervals of stage within a series. When a great number of stages are preceded by if! Error do..., the code morphs to simply "go to an error handler" as so soon being an error occurs, leaving the standard flow code unburdened by if statements.

Should the compiler not need to indicate that a function is an interrupt, then it really is finding various other approaches to make the return from interrupt happen. This is, the compiler is most likely wrapping the interrupt function in assembly code that merely calls the function. When the handler returns from the function call, the assembly wrapper returns from the interrupt. The processor resets the stack the actual way it was and program execution continues from precisely the point it left off.

**2.5.9 When You Should Use Interrupts**

Now that we've established our stoplight to utilize an interrupt with the timer and made the code to take care of the interrupt, we should backtrack. There are several circumstances in which the simplest solution is an interrupt. Communication pathways often have buffers that require be filling (or emptying). An interrupt can behave as a background task to feed the buffers even though the foreground task generates (or uses) the data. Changes to input lines may need interrupts whenever they ought to be handled quickly. The greater will be the requirement to handle a big difference on the line, the greater an interrupt is proper for any solution.
A button press happens pretty rarely on a processor. However, if checking to see whether or not it has become pressed needs time to work from other considerations, it usually is better to produce an interrupt. The discussion has also shown that interrupts have serious downsides also.

Interrupts also make the system less deterministic. One of many great things about without having an operating system is it being able to say that after instruction \( x \) happens, \( y \) will happen. If the programmer has interrupts, it loses the predictability of this system. And, as the code has stopped being linear in flow, debugging is harder. Worse, some catastrophic bugs are going to be very difficult to locate as long as they be determined by an interrupt happening in a very specific time the code (i.e. a race condition). Plus, the configuration is largely compiler and processor dependent along with the implementation might be besides, so interrupts often make the code less portable.

In the end, the growth price of implementing (and maintaining) interrupts is pretty high, sometimes more than figuring out the best way to solve the situation accessible without them. One should save interrupts when one needs their special power: each time a system is time critical, when a celebration is pricey to check on for and happens very rarely, when a short background task will permit system run more smoothly.

### 2.5.10 Avoid Use of Interrupts

So, if the programmer does not need their special power, how might he avoid interrupts? Several things could be solved in hardware, such as employing a faster processor to maintain time critical events. Other items require more software finesse. The implementation of the system might be incredibly easier to maintain if one can always see easily what the code is anticipating.

At times, interrupts turn into a strategy to wake the processor from sleep, therefore they become necessity.
2.5.11 Polling

Asking an individual “are you done yet?” is usually considered impolite after the fourth or fifth query. The processor doesn't care should the code incessantly asks whether a conference is prepared. Polling adds processor overhead regardless if there won't be any events to process.

However, if the programmer is intending to stay quite some time loop anyway (i.e. an idle loop), there is no reason not to look for events. Polling is straightforward to code. There's just one single subtlety worth mentioning: if you are polling and looking forward to the hardware to perform complete something, it's advisable a time out in case. While Channel Embedded Systems have a reputation for being very fast, many systems spend an inordinate volume of their clock cycles looking forward to time to pass.

2.5.12 System Tick

Such as sound you hear in the event the clock's used moves, many systems have a very tick that indicates time is passing. As you move the amount of time in that tick varies, one millisecond is commonly a common choice. Ticks don’t have to be one millisecond. If you have a period of time that may be crucial that you your system for some other reasons (e.g. you would like to audio system that is certainly running at 44100Hz), you might use that instead.

The system tick solves a lot broader selection of problems. For example, it allows us to define this function: void Delay Ms (Time delay);

This will likely wait for a length of time indicated—well, for the amount of energy indicated. Observe that as a consequence of fence posting and jitter, Delay Ms isn't a fantastic measure of one particular millisecond. However, in case the programmer wants to delay ten or a hundred milliseconds, the error becomes small enough not to ever matter. Should the system be dependent upon one-millisecond accuracy, he could employ a shorter tick, though he should balance the overhead from the timer interrupt using the processing needs of the other parts in the system
2.5.13 Watchdog

Metaphorically speaking, it acts as a watchdog to prevent catastrophic failure. Actually, the phrase watchdog means something a great deal more specific inside embedded world. Most processors or reset circuits have a watchdog timer capability that may reset the processor if your processor does not perform an action for example toggle an I/O line or write to a unique register. The watchdog system waits for that processor to deliver a signal that the relationship is running nicely. If a real signal fails to happen in a fair (often configurable) period of time, the watchdog causes the processor to reset.

The goal is always that if your system fails, it fails within a safe manner (failsafe). No one wants the system to fail, but you've gotten being realistic. Software crashes. Even critical safety software crashes. A programmer designs and develops his systems, he operates avoid crashes.

But, unless the programmer is all-knowing, the software will fail in the unexpected way. Intergalactic rays and slack wires happen. Many Channel Embedded Systems ought to be self-reliant. They can't lose time waiting for you to definitely reboot the system if your software hangs. They might not even be monitored by way of a human. In the event the system can't recover from some varieties of error, it really is generally far better to restart and hang the system in the good state.

Employing a watchdog won't free the programmer from handling normal errors; instead, it exists only when the system is unrecoverable. There are ways to use a watchdog which render it far better. However, let's talk about some suboptimal techniques, determined by models that would result in the watchdog less capable: Starting a timer interrupt that beeps better often than the watchdog would choose to use expire.

In the event the programmer services the watchdog within the timer interrupt, your system won't ever reset even though your system is stuck in a infinite loop. This defeats the aim of the watchdog.

By scattering the watchdog code around, it waters down the strength of the watchdog while offering the chance that your code could crash in a of the people areas and hang the system. The purpose of the watchdog should be to supply a strategy to evaluate if any kind in the system has hung. Ideally, watchdog servicing is in only 1 place, an
area that this code must pass through that shows most of its subsystems are running not surprisingly.

Generally, this is actually the main loop. And sometimes it means that a watchdog needs a lengthier timeout, having it watch the entire system provides improvement over creating a shorter recovery time while trying to watch only the main system. Sadly, for many systems, the watchdog can't be segregated so neatly. When the signal to the watchdog has to be submitted to some lower degree of code, notice that the code can be dangerous, and become a place where an unrecoverable error might occur, creating the system to hold. That code will need extra attention to determine whether anything could go wrong and hopefully prevent it.

Generally it can be not be suggested that the watchdog be active during board bring-up or when using the debugger. Otherwise, the system will reset after having a breakpoint. A straightforward strategy to shut off the watchdog will facilitate debugging. If there is a logging method, one has to be sure to use a communication on boot if the watchdog is on. It really is a type of stuff that should not be forgotten to make it possible because production testing is performed. Alternatively, you can toggle an LED if the watchdog is serviced to supply the system a heartbeat that is certainly easy to understand externally, letting the user are aware that things are all being employed as expected.

Whether we are making a device that fits in the pocketbook or seeking to save the planet by cutting our company's carbon footprint, decreasing a system's power consumption may take an order of magnitude additional time than implementing the item features.

Choosing all the right hardware components is a huge component of setting up a system power efficient. But because the processor may just be among the largest consumers of power inside the system, software can enjoy an enormous role in preserving electricity.

The pressures to reduce power usage and cost are what bring this discussion to processors which do not have sufficient resources to comfortably perform the merchandise features. Then again, a processor using the minimum number of power cycles, RAM, and code space consumes less power than one using sufficient amount of resources.
2.5.14 Understanding Power Consumption

All electrical circuits may be modeled as resistors, though it is like saying all differential equations might be modeled linearly. It is true, but based on the circumstances; one could possibly expect to see a lot of errors. However, this can be a good start. Power is measured in watts and is proportional towards square in the current employed in the system:

\[ P = I^2 R \]

power = (current)² * resistance

Watts (W) = (Amps (A))² * Ohms (Ω)

As your system is modeled as being a resistor, in case you concentrate on decreasing the power consumption. A way to look at power is usually as the merchandise of voltage and current:

\[ P = V \times I \]

power = voltage * current

Watts (W) = Volts (V) * Amps (A)

That is why the processor core might run at 1.8V although other regions of your respective system run at 3 or 5V. Because the core uses a lot of current, the fewer voltage means less power consumption. Both the strategies to considering power are equivalent, good golden rule of electrical engineering, Ohm's law:

\[ V = I \times R \]

voltage = current * resistance

Volts (V) = Amps (A) * Ohms (Ω)

Another useful equation is designed for energy:

\[ W = P \times T \]

energy = power * time

Joules (J) = Watts (W) * seconds

Basically, it means that if someone is designing an electricity-efficient system, he is able to minimize the power it uses or time frame it truly is on.

Batteries possess a voltage rating and a capacity. An alkaline AA battery incorporates a nominal
voltage of just one.5V as well as a capacity of around 3000mAh (milliamp hours). So capacity isn't rated in power but also in current supplied after some time. But if the system takes 30mA and 1.5V only when it's on, with the AA battery, it should last about 100 hours (~4 days). But if the system takes 300mA, it will last about 10 hours. However, if the system consumes 3000 mA, the AA battery won't last an hour or so, because, just as with calling the system a resistor, this rule of thumb only goes hitherto. The battery can have a datasheet that explains its capacity, peak current draw, along with characteristics.

From both perspectives (power and battery capacity), minimizing current consumption is vital to reducing energy consumption or increasing battery life. Let down the sunshine if you Leave the area the way to reduce power consumption is turn off components that aren’t needed. The downside is those components will never be ready when you need them, and bringing them back will add both some power usage and a few latency in giving an answer to events. The programmer will need to investigate the trade offs. The discussion ahead details several of the approaches to switch off or lessen the power drain of components.

2.5.15 Let Down Peripherals

While one reviews the schematic and the design, consider what the processor has access to and ways to design the system to turn off peripherals that aren't needed. For example, for those who have external RAM, they could be able to preload the information into a local location so power down the RAM rather than leaving it powered on for longer durations.

A chip external for the processor will consume no current if it doesn't have power. If the processor doesn't always have access to the power lines, holding the peripheral in reset will nearly get rid of the current consumption. This isn't as optimal as turning from the power, but a lot better than leaving the processor running.

Many chips need time to return to functioning after reset, so there is certainly some overhead in powering them down. The goal would be to go out the component for just a reasonable amount of energy. ADCs are some of the peripherals with longer reset times. Nevertheless they also often big power consumers, so it's often worth spending some time to balance the facility consumption versus the electricity-in time.
2.5.16 Turn Off Unused I/O devices

If there are some spare I/O devices, on saves a small amount of current by configuring them to become inputs with internal pull-downs. If the chip doesn't need internal pull-downs, set it to be a low output. In the event that doesn't work available for you, an input with pull-ups is still equipped with reasonably small quantities of leakage current. Don’t member pull-up assay thing besides gym torture?

Wholesome diet I/Os attached to the component you've got powered off, the preferred order is the similar input with pull-downs, output low, input with pull-ups. Configuring one I/O in this way isn't prone to save a great deal, but when a couple of them are done, the tiny savings can also add up. Then again, if some I/Os in the system provide an external pull-up or pull-down, set the inner that matches.

2.5.17 Fighting External Resistors Wastes Power

Should there be a peripheral that is certainly powered off maybe in reset; be careful how to connect other processor I/Os thereto. One should take care to never leave the lines pulled-up or set high. Chips usually have internal protection diodes that might use the high signal to back-power and damage the chip.

2.5.18 Switch off Processor Subsystems

In addition to setting I/O devices for being low power, one should cut whole subsystems from the processors that are not needed. The processor manual will show whether that is a supported feature with the processor. Sometimes it is possible to switch off all functionality; sometimes it is possible to only cut the hands of time on the peripheral.

2.5.19 Slowing to Save Energy

Cutting the clock frequency saves power. This is true of all of the clocks that can be controlled, but specially the processor clock. However, a slower processor clock gives
fewer processor cycles to run in. This is why one should spend much amount of time on optimizing code, despite the fact that optimizing for processor speed will decrease the qualities like robustness, readability, and debug ability of one's code. Often, probably the most straightforward approach to decrease power use of the system by 10% would be to decrease the clock by about 10%. In other words, the facility usage of the chip is proportional towards frequency it runs at.

If the processor must be on at all times, by way of example, monitoring its environment, to be able to slow down can be vital to achieving low power. Some embedded operating systems will do frequency scaling to suit the needs, consuming power efficiently when possible yet still getting the speed available when it's needed. This can be without an operating system, rather than scaling within the whole range; one should create slow, medium and fast modes and affect the performance level depending on events. However, similar to running a race, there's a balance between slowing the processor down and keeping it on for a longer period to be a tortoise or sprinting for just a shorter period and sleeping like a hare.

### 2.5.20 Putting the Processor to Rest

Even at the slow clock speed, the processor is consuming power. Scaling down can help, but imagine the code spends time and effort waiting for what to happen?

Many processors intended for low-power systems will go into a power-conserving sleep if they aren't needed. They normally use interrupts to express to the processor when to wakeup.

On the computer, sleep is a standby mode where processing is off however the memory is still powered. This lets the operating system awake quickly, whereby it had left off. Alternatively, in hibernation, the contents of RAM are written to the disk or other non-volatile memory. It will require longer waking from hibernation, though the system uses significantly less power within the state may be minimal power, based on the maker of the computer.

Most embedded processors present with a similar variety of sleep possibilities, by processors having a lot of granularity among others opting to own merely one low power mode. In decreasing numbers of power consumption, some sleep modes many times with the processor manual include:
Reduce
Going beyond frequency scaling, some processors assist the programmer to slow the time down to many hertz.

Idle or sleep
This turns journey processor core but keeps enabled timers, peripherals and RAM alive. Any interrupt can return the processor to normal running.

Deep sleep or light hibernation
As well as the processor core being disabled, some and possibly all peripheral scan be configured to show off. Try not to go out the subsystem that generates the interrupt that can arise the processor.

2.5.21 Deep Hibernation or Power Down

The processor chooses which peripherals being switched off (usually most of them). RAM is usually left in the unstable state; even so the processor registers are retained, to ensure the system doesn't need initialization on boot. Usually, just a wakeup pin or a small subset of interrupts can restart the processor.

Power off
The processor will not retain any memory, and starts at a completely clean state. Deeper sleep modes position the processor into a lower power state but take more time to wake from, increasing system latency. The programmer should invest some time while using user manual to discover how the programmer can reach the lowest power level feasible for will see the product's needs.

The wake-up interrupts may be buttons, timers, other chips (i.e. ADC conversion complete), or traffic with a communications bus. This will depend on the processor along with the sleep level.

Once the programmer has determined that the processor can sleep, how does he design a system to take advantage by using it? Let us turn the discussion to that now.

2.5.22 Interrupt-Based Code Flow Model

Understanding the interrupt-based code flow model is crucial for power sensitive applications. Deteriorating the long name inside the title in this section, interrupt-
based suggests that the code will hear interrupts to accomplish its business. And when there is a sub system which doesn't normally cause an interrupt, the programmer should make one, probably with a timer that can do all of the regular housekeeping. Another part of the name is code flow to show which the program is likely to flow along being a waterfall. Doesn't necessarily have multiple tasks and it is often linear. Button handlers are likely to spend considerable time idling, awaiting control button to be pressed. Once is, the handler calls the appropriate function, and after it returns, the handler returns to expecting something to occur. Rather than idling, the microprocessor spends time sleeping; waking up not until it would need to, then handling the interrupt all night returning to sleep. The goal should be to maximize the amount of time the processor is asleep, thereby maximizing efficiency. See Figure below for a flowchart describing the interrupt-based code flow model.

![Interrupt-based code Flow Chart](image)

Fig. No. 2.4 Interrupt-based code Flow Chart
The interrupt might be generated from the peripheral to indicate it needs processor attention just like an ADC finishing featuring its data, needing the processor to see the result and start another conversion. In other systems, the interrupt can be generated by a button the user presses or even a sensor rising above a threshold. The interrupts will also be generated internally, for instance, by way of a timer.

When an interrupt occurs, the processor awakens and calls the right interrupt utility routine (ISR). The ISR sets a flag to indicate the reason for the wakeup and modifies the sleep register to keep the processor awake. If the ISR returns, because the sleep register is set to awake, the leading loop runs. The key loop checks each possible flag and, when it finds a flag set, runs the appropriate handler also clearing the flag. If the main loop completes and every one of the flags is already cleared, the processor returns to rest mode. Figure 2.4 illustrates the typical idea with two handlers to check.

Let's imagine the programmer will have a processor that could sleep. To experiment, commence with a timer interrupt. These include common in the interrupt flow model for the reason that timer enables the programmer to waken early go necessary housekeeping. Run the processor without sleep, and hang up an interrupt running therefore it toggles an output line or LED. The regularity no matter, though a slow interrupt is much better. I'd recommend about twice per second (2Hz). Measure the present state of the system. Now set the processor to get to sleep between interrupts. On the surface, the processor behaves the identical. However, when the programmer measures the system current, it should be smaller. The amount smaller depends upon many areas of the system.

While it can be allowed to enter a decreased power mode as an alternative to idling inside the while loop, the outcomes were dramatically different. Remember, energy efficiency is time in addition to current. The processor on TI's Launch pad kit was created to be super low power only when it's sleeping, therefore the results might not be so dramatic on another device.

### 2.5.23 Processor Watchdog

Seeing that the system is entirely interrupt based, how's it going to make sure the watchdog remains content? Within the lightest degrees of sleep, the processor will probably permit watchdog continue running. The processor may allow the
programmer to configure whether to choose the watchdog to own as the code moves
the processor is asleep consuming some power while asleep and forcing the processor
to wake up to service it or develop the watchdog sleep if the processor does lose a
failsafe mechanism for the system. Recognizing the trade off will help the
programmer in making the suitable decision for the system.
When the programmer leaves the watchdog on, he will need to set a timer to wake the
processor up before the watchdog expires. This goes against the grain of my earlier
advice. However, here the watchdog is not only verifying that this system is running
properly, it can be verifying how the system is wakening from sleep properly.

2.5.24 Avoid Frequent Wake-ups

Since each wakeup requires some overhead on the chip (as well as the deeper the
sleep, a lot more overhead is necessary because it awakens), avoid frequent wake-ups.
For those who have a low-priority task that doesn't use a hard real-time requirement,
piggyback it upon another task. This spreads the overhead of wakening the processor
over several tasks.
While a timer to try and do housekeeping may be valuable, it can be safer to check
whether time has passed to perform over the priority tasks and reset the timer to help
the programmer skip a wakeup.

2.5.25 Chained Processors

It is pretty common to use a small, very efficient processor monitor quite signals,
stumbling out of bed a bigger processor if needed. In this instance the larger processor
requires more chance to arise and perchance more time if it's running an operating
system. To ensure the small processor does the housekeeping: checking for buttons
pressed, seeking lower power conditions that indicate the system should close up, or
waking the big processor caused by a preset alarm.
In such systems, both processors spend as often time asleep as is possible, even so the
system is made so the interrupts get triaged from the small processor, which often can
opt to handle them itself or arise its larger partner.