Tecnológico de Monterrey

# Hardware and Software Optimizations to Satisfy the Response of Embedded Systems in real time.

Isaac Benjamín Ipenza Retamozo A01228344

Perla Vanessa Jaime Gaytán A00344428

Gabriela Hernández López A01634320

**Embedded Systems**

**(TE3059)**

**November 30th, 2020.**

## TABLE OF CONTENTS

## TABLE OF FIGURES

# 1. Introduction

In this document it will find information related to how to do a better optimization of embedded systems to have a better response on real time. It will include hardware and software optimizations. It also contains some conclusions regarding this investigation at the end and the references that were used to do this report.

## 1.1. Document Organization

This document is organized as follows:

| | |
|---|---|
| **Section 1 Introduction** | It contains a short description of what content it can be find on this document and the most important point on it. It also contains information relatable to understand this document. |
| **Section 2 Reduce the Size of a Program.** | Techniques use nowadays to reduce the size of a program in an embedded system. These techniques are divided in two subsections which are: HW Level and SW Level. |
| **Section 3 Reduce the Execution Time of a Program.** | Techniques use nowadays to reduce the execution time of a program in an embedded system. |
| **Section 4 Reduce the Energy Consumption.** | Techniques use nowadays to reduce the energy consumption on an embedded system. |
| **Section 5 Conclusions** | In this section some conclusions are shown regarding the investigation on this document. |
| **References** | It contains the references that were used during the making of this document. |

## 1.2. TERMINOLOGY

The following acronyms are used all along the document.

| **Acronym** | **Description** |
|---|---|
| ADC | Analog to Digital Converter |
| CISC | Complex Instruction Set Computer |
| CPSR | Current Program Status Register |
| DAC | Digital to Analog Converter |
| DDR | Double Data Rate |
| DMA | Direct Memory Access |
| DSP | Digital Signal Processing |
| EMIO | Extended Multiplexed I/O |
| FIQ | Fast Interrupt Request |
| FPGA | Field Programmable Gate Array |
| GEM | Gigabit Ethernet MAC |
| GUI | Graphical User Interface |
| HDL | Hardware Description Language |
| HLS | High Level Synthesis |
| HW | Hardware |
| IRQ | Interrupt Request |
| LED | Light Emitting Diode |
| LSB | Least Significant Bit |
| LVDS | Low-Voltage Differential Signaling |
| MAC | Medium Access Control |
| Mbps | Mega Bits per Second |

| | |
|---|---|
| MMCM | Mixed-Mode Clock Manager |
| MSB | Most Significant Bit |
| NCO | Numerically Controlled Oscillator |
| OTR | Out of Range |
| OS | Operating System |
| PC | Personal Computer |
| PL | Programmable Logic |
| PLL | Phase Locked Loop |
| PS | Processing System |
| RISC | Reduced Instruction Set Computer |
| RMII | Reduced Media Independent Interface |
| Rx | Reception |
| SFP | Small Form-Factor Pluggable Transceptor |
| SoC | System on Chip |
| SoW | Statement of Work |
| SP | Stack Pointer |
| SVC | Supervisor Call |
| SW | Software |
| Tx | Transmission |

## 2. Strategies to Reduce the Size of a Program.

Laptops, PCs, and servers have a very huge amount of resources at their disposal. Figure 1 shows and example of an embedded system. They are usually small and build to do a certain task. Therefore, they have very limited amount of resources. Some features that embedded system have include are clock frequencies reaching only few MHz, word-lengths of 8 or 16 bits and memories with a store of a few mega or kilobytes. This is the main reason why optimization on the design of an embedded system plays an important role and why code must be optimized to complete all the requirements but also with the limited amount of resources they have. [11]

Code size optimization is part of the fine-tuning phase on the develop of embedded system which is usually do after debugging has been complete. This part is where an exploration is made with the purpose of finding better options which minimize the code size and maximizes the performance. The idea is to shrink code size without affecting the performance. There are two ways of do the optimizations of a program which are at HW Level and SW Level. [7]

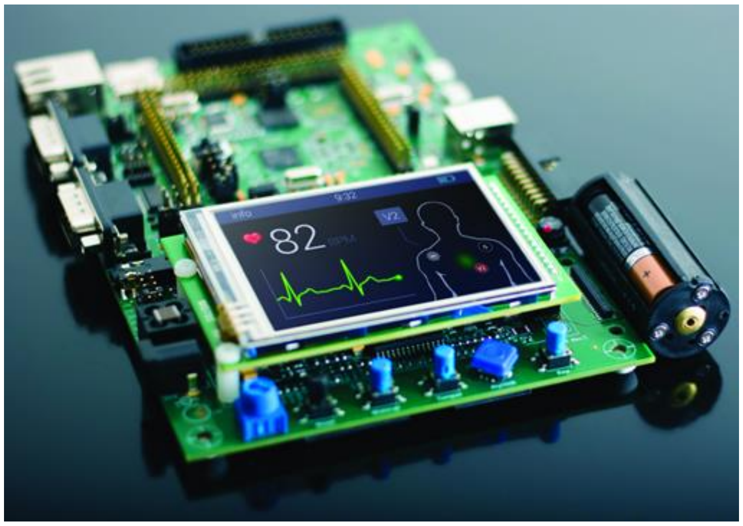


**Figure 1. Example of an Embedded System.**

## 2.1. HW LEVEL

The most of the HW Level techniques involve several levels of memory hierarchy to save the most. Some research shows how cache control mechanisms will work for this purpose. Some examples are victim cache, column-associative caches, hardware prefetching mechanisms, and cache bypassing using memory address table. [5]

Victim cache is a hardware technique to improve performance of caches which is fully associative cache placed in the refill path of a CPU cache that stores all the blocks evicted from that level of cache. It was designed to decrease conflict misses and improve hit latency for direct-mapped caches. An implementation of this technique is shown in Figure 2. [3]

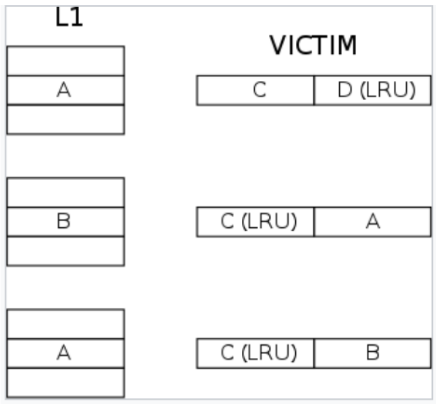


**Figure 2. Implementation of victim cache.**

Column-associative caches predicts the percentage of interface misses removed from a direct-mapped cache using only one measured. This model makes the assumption that blocks have a uniform probability of mapping to any cache set, and that the mapping for different blocks are independent of each other. An example of this is shown in Figure 3. [1]

Cache prefetching is used by fetching instructions or data from their original storage in slower memory to a faster local memory before it is actually needed. A dedicated HW mechanism in the processor that watches the stream of instructions or data being requested by the executing program, recognizes the next few elements that the program might need based on this stream and prefetches into the processor's cache. [14]

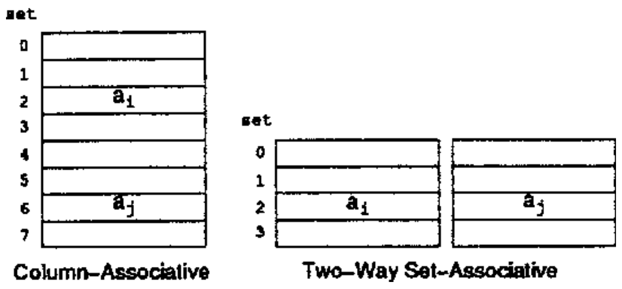


**Figure 3. Column-associative example.**

## 2.2. SW Level

Embedded systems programmers usually are dealing with low-cost-8-bit microcontrollers which have a small limited flash memory. Therefore, it is recommended to optimize the C code and reduce the program memory bytes used on each microcontroller. There are some strategies that can help optimizing the code and some of them are mention below. [2]

The first strategy is to build your code in layers as shown in Figure 4 because it will help with the understanding of it. It is important to remembered that this helps with the maintainability of the code. One common mistake made while working coding is not using the appropriate conditional statement. The most often used is the if/else statement, the else statement it is often forgotten and, in some cases, could replace an if statement as shown in Figure 5. However, there is also the switch case statement, which allows the processor to check the statement and then select the best answer from a list. This conditional statement is recommended while using more than one conditional to safe clock cycles. In the same line as keeping the code in layers is using descripting variable names. Writing great software that is understandable and easy to maintain is essential por embedded systems, therefore it is important to keep variables names obvious not only for the developers, but also for further maintenance. Some examples of this variables are shown in Figure 6. [5]

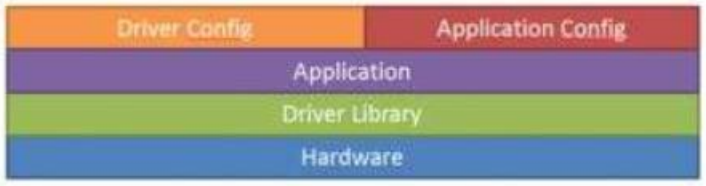
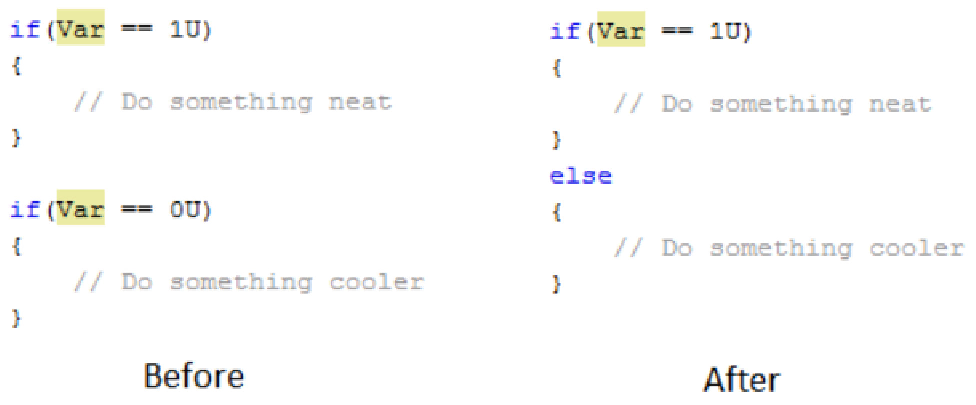


**Figure 4. Example of using layers on a code.**

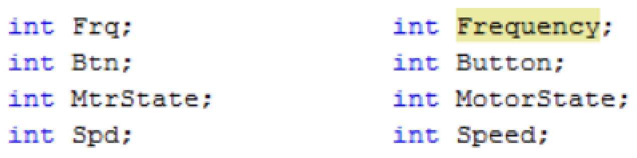


```
if(Var == 1U)
{
    // Do something neat
}

if(Var == 0U)
{
    // Do something cooler
}
```

```
if(Var == 1U)
{
    // Do something neat
}
else
{
    // Do something cooler
}
```

**Figure 5. Using if-else statement examples.**



```
int Frq;
int Btn;
int MtrState;
int Spd;
```

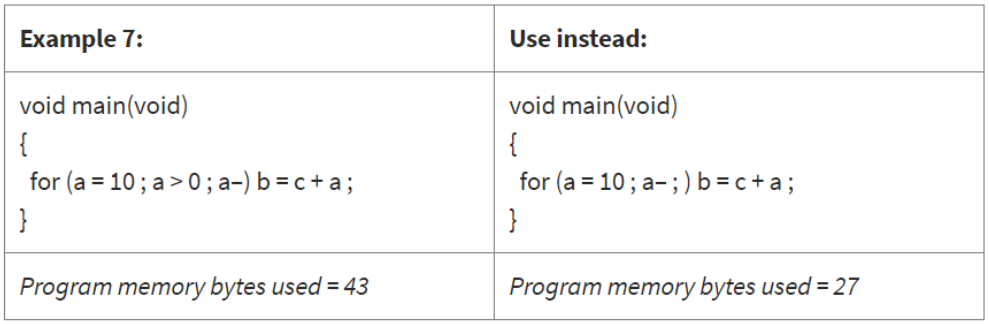
```
int Frequency;
int Button;
int MotorState;
int Speed;
```

**Figure 6. Example of declaration of variables.**

It is important to remember that fewer lines do not mean better program or short in size. Size in software means how many memory does it required and not how many lines does the code has. Using short mathematical expressions (Figure 7) is an example of a very simple trick that can be applied in to save some memory space, even though it will mean more lines. Another simple trick can be shown in Figure 8 where a for instruction is written in different ways but doing the same task. [5]

| **Example 1:** | **Use instead:** |
|---|---|
| void main(void) { int a, b ; a = (b – 1) * 3 ; } | void main(void) { int a, b ; a = (b – 1) ; a = a + a + a ; } |
| *Program memory bytes used = 50* | *Program memory bytes used = 25* |

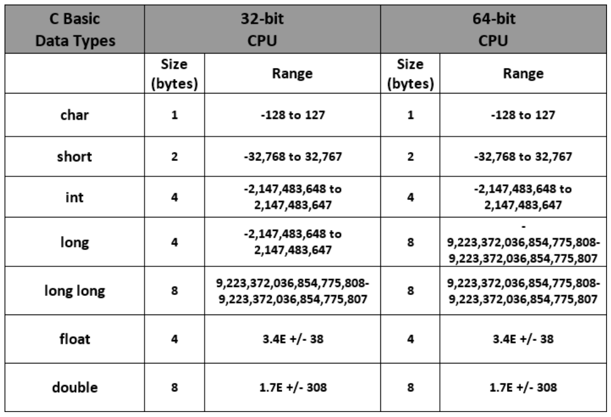
**Figure 7. Mathematical expressions.**



| **Example 7:** | **Use instead:** |
|---|---|
| void main(void)<br>{<br> for (a = 10 ; a > 0 ; a–) b = c + a ;<br>} | void main(void)<br>{<br> for (a = 10 ; a– ; ) b = c + a ;<br>} |
| *Program memory bytes used = 43* | *Program memory bytes used = 27* |

**Figure 8. Different using of a for.**

There are some other methods that can be implemented to keep in mind when developing embedded systems. One of them is Data Type Selection which can play an important role while reducing code size. Do the selection of the data type of the variable can be tricky, but one way to do it is by according to its maximum value held on the variable. Data types can be shown in Figure 9 and it is important to keep them in mind to do not waste memory by choosing one of a 16 bits when we only need 8 bits. Another method to keep in mind is Loop Jamming which refers to merch the statements and operations from different loops into only one. The use of const keyword is important to initialized values that does not need to change. Using const will save program memory reducing RAM memory for the program. Last but still important is the correct declaration of local and global variables. Declaring a variable as global requires a separate memory location and on the other hand, local variables are store in stack memory only when it is required. [15]



| C Basic Data Types | 32-bit CPU | | 64-bit CPU | |
|---|---|---|---|---|
| | Size (bytes) | Range | Size (bytes) | Range |
| char | 1 | -128 to 127 | 1 | -128 to 127 |
| short | 2 | -32,768 to 32,767 | 2 | -32,768 to 32,767 |
| int | 4 | -2,147,483,648 to 2,147,483,647 | 4 | -2,147,483,648 to 2,147,483,647 |
| long | 4 | -2,147,483,648 to 2,147,483,647 | 8 | - 9,223,372,036,854,775,808- 9,223,372,036,854,775,807 |
| long long | 8 | 9,223,372,036,854,775,808- 9,223,372,036,854,775,807 | 8 | 9,223,372,036,854,775,808- 9,223,372,036,854,775,807 |
| float | 4 | 3.4E +/- 38 | 4 | 3.4E +/- 38 |
| double | 8 | 1.7E +/- 308 | 8 | 1.7E +/- 308 |

**Figure 9. Data types.**

# 3. Strategies to Reduce the Execution Time of a Program.

Embedded systems consist of programmable software part (SW) and application specific hardware part (HW). Software part is much easier to develop and modify, and it consumes less power compared to the hardware part but it requires extra time to give final response. In fact, compared to the software which is less expensive in terms of cost and power consumption, the hardware provides better performance because it offers a faster treatment. For that reason, the purpose of HW/SW partitioning is to design a balanced system that accomplishes all system constraints. [8]

When working with time-execution optimization, speed at which the said embedded systems perform their task tend to be of critical interest to the developers. There are some cases where timing is indispensable to fulfill hard real time systems requirements, while in other cases time optimization can produce another one at reducing energy consumption that will eventually lower costs since cheaper hardware can be used.

There are several ways to approach the problem of timing optimization. The first one to be described here is some techniques fully related to code itself. Once execution time is stated as the primary issue to be solved over some other things like code size or power consumption the developer can try applying some or all of the following techniques.

## 3.1. Inline functions

This refers to a function that is commonly used with classes where the compiler places a copy of the code of that function at each point where the function is called at compile time. The keyword inline can be then added to any function declaration and it'd eventually ask the compiler to replace all calls to the indicated function with copies of the code that are inside, allowing the runtime overhead associated with the function call to be eliminated. It's extremely helpful especially when the function is used frequently but is short in lines of code.

## 3.2. Table look-ups

This term refers to the action of building an array or matrix data that contains data that can be searched where key-value pairs are formed having the items being searched as the data items and the values being the actual data or pointers to where the data is located. Also called Lookup Functions, Tables, Lookup Tables or Table Functions, one uses them to create their own specialized functions.

Within a program, a switch statement could be used to decrease execution time by putting the individual cases in order by their relative frequency of occurrence. This will reduce the average execution time, though it will not improve at all upon the worst-case time (Barr & Massa, 2009). Thus, if there is a lot of work to be done within each case, it might be more efficient to replace the entire switch statement with a table of pointers to functions.

## 3.3. Register variables

When using the register keyword to declare local variables, the compiler is asked to place variables into a general-purpose register rather than on the stack. If used correctly this will provide hints to the compiler about the most frequently accessed variables and will enhance the performance of the function given that the more frequently the function is called, the more this change will end up improving the code's performance.

## 3.4. Global variables

By using these we can achieve efficiency and therefore avoiding the passing of parameters to functions. Doing so eliminates the need to push the parameter onto the stack before the function call and pop it back-off once the function is completed. In fact, the most efficient implementation of any subroutine would have no parameters at all.

## 3.5. Fixed-point arithmetic

If the application contains one or two Floating-point instructions then they can be implemented by using fixed-point arithmetic. For example, two fractional bits representing a value of 0.00, 0.25, 0.50, or 0.75 are easily stored in any integer by multiplying the real value by 4 (e.g., << 2). Doing so reduces the very large penalty a processor pays when manipulating a float data type, in contrast to the integer counterparts.

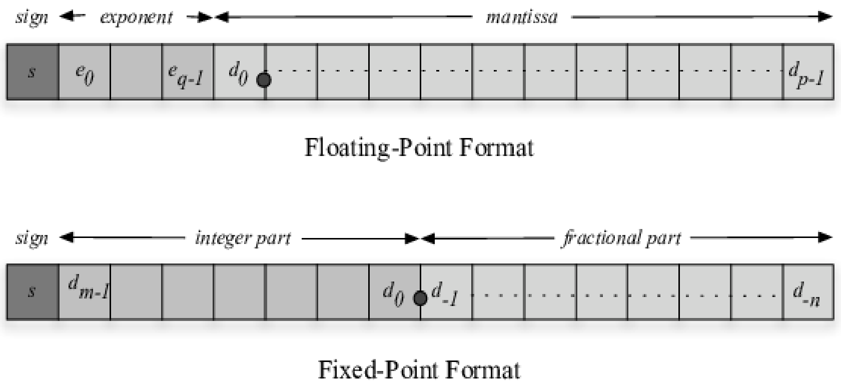


**Figure 10. Floating-Point vs Fixed-Point.**

## 3.6. VARIABLE SIZE

Using the processor's native register width for variable size (8, 16 or 32-bits), allows the compiler to produce code that leverages the fast registers built into the processor's machine opcodes (Barr & Massa, 2009). This makes the number of external memory accesses to be limited if the variable size is being tailored to the processor's registers.

The second part of this section refers more to the hardware side of an embedded system. It's indeed a task that involves a little more trouble than the software part since several things can affect the performance of the numerous components that will build your embedded system.

The first thing and most remarkable thing we found while doing research was to check on the clock configuration and to use the clock output feature that many processors (just as the DE10 Standard board) include, which allows you to route the processor clock to a port pin so it can be monitored with an oscilloscope.

For high-speed processors it's also important to have the correct number of wait states set for the distinct memory spaces and also if clock speed was to be increased, it'd be also necessary to increase the number of wait states. This leads to the statement that not increasing clock speed doesn't necessarily mean an increase in performance and implying that it’s always important to understand and analyze the tradeoff between clock speed and the final throughput.

It is indeed hard not to find articles and several sources mentioning that one of the main thing developers have to be sure while working with a certain processor is that they’re using its hardware features at its fullest. For example: most processors include FIFOs on communication interfaces that when turned on could impact on performance particularly for high speed interfaces. The same happens let’s say whenever a processor includes a DSP multiply-accumulate unit that will perform certain calculations in a much faster way that a C code would.

# 4. Strategies to Reduce the Energy Consumption in an Embedded System.

An embedded system is a computing system that  is designed for a specific control function, and  to take care the power consumption of this is an important aspect when we design a battery-powered product or any device that will be consuming electricity continuously. There are different methods that allow reducing energy consumption, however the application of these will depend specifically on the device in which you work. Some strategies are presented below.

## 4.1. Use Deep Sleep Mode

Some powerful microcontroller also consumes higher current when all the peripherals are turned on, and this results in the need for a  higher capacity battery if the microcontroller is continuously running at full power, so to prevent the microcontroller from  drawing maximum  power is to put it into  deep sleep mode when it is idle; in this  mode the  microcontroller often consume a tiny fraction of the maximum current. Firmware engineers can then use interrupts to wake up the microcontroller as needed. [Altium]

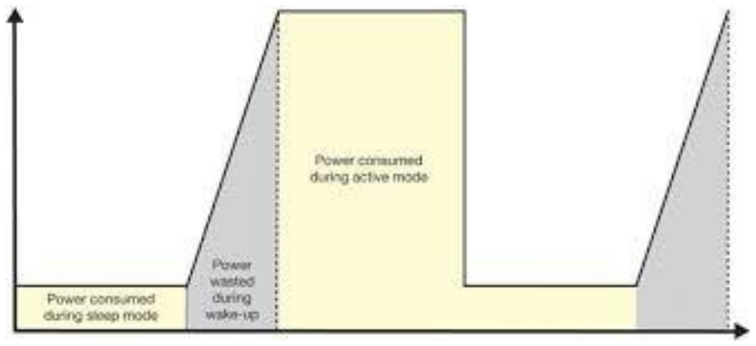

Power consumed during active mode
Power consumed during sleep mode
Power wasted during wake-up

**Figure 11. Power consumption in the Embedded systems modes**

## 4.2. Use Switching Regulator

In some Embedded systems, unnecessary power wastage occurs  in the form of heat, an this  is  true when you  are using linear regulator in the design, naturally, a linear voltage regulator is the cheapest solution for some embedded systems but this are not know to be efficient, as  the dissipate the difference between the voltages as heat. The heat dissipation results in extra current being drawn from the battery. If this is the case, you can opt to use a switching regulator instead, switching regulators and associated components increase the overall unit cost, this is a more effective tactic than using linear regulators. Less heat dissipation means a longer lasting battery, which is useful in cases when the device has to operate without sunlight. In power sensitive applications, every milliwatt saved can make a huge difference.[13]

**Figure 12. Heat dissipation in a PCB.**

## 4.3. POWER OFF UNUSED COMPONENTS (INTELLIGENT SHUTDOWN)

Putting the microcontroller into deep-sleep mode is a brilliant technique for reducing total power consumption. But other components like logic integrated circuits (IC) or communication IC can still drain significant current even when your microcontroller is in deep sleep mode. This issue can be minimized, instead of having a single voltage regulator on the PCB, you can use two regulators to separately power the microcontroller and the other components. This method allows the microcontroller to turn off the power of other components before entering deep sleep mode.

Minimizing sleep mode power consumption requires analyzing the hardware in your system and determining how to set it into the lowest-possible power state. Most battery-operated systems continue to power their general-purpose I/O pins during sleep mode. As inputs, these I/O pins can be used as interrupts to wake up the device; as outputs, they can be used to configure an external peripheral. Careful consideration of how these pins are configured can have a large effect on sleep mode power consumption. [13]

## 4.4. Keep operating voltage low in the embedded system design

Keep the overall operating voltage low for the whole board as power consumption is directly proportional to the operating voltage. Use the lowest voltage level possible; if all the chips can go as low as 2.7, we can keep a small margin and set that voltage for the whole board. If there is a need to have two sets of power rails on the embedded board and considerable power saving is there then it makes sense to do that, although an extra DC-DC converter and some digital level translation chips will be required to do so. [Aggarwal]

## 4.5. Select low power consumption components, chips, and modules.

Selecting the right ICs considering overall power consumption budget is very important, pick ICs with low power consumption (active / Idle) and with a low operating voltage rating. It is good to compare their power-up time, active power consumption, idle power consumption parameters. Selecting the right technology is also critical. Many times, if you select the wrong technology it will be difficult to optimize power consumption later. [1]

## 4.6. Select the right input voltages to the embedded system (Power Supply).

Selecting the right input voltage to the embedded board is important. Whether it is a battery input or a DC input from a power supply adaptor. If all the circuitry on the board is powered by 5v or 3.3V then using 6V power input is better than using a 12/24V DC input or battery input. Power loss will be proportional to voltage difference. [1]

## 4.7. Remove LEDs on board.

LEDs onboard can easily consume 1–5mA per LED, if there are LEDs on board you can do the following:

- Remove LEDs if possible
- Reduce the brightness by using higher series resistance value
- Use blink LED instead of making it full time ON
- Only shows when the user interacts with  the device or when its required

[1]

## 4.8. Pull-ups

The optimal value of pull-ups can help reduce power consumption. Pull-ups are mainly used for I2C, keys, etc.. Each pull-up can contribute to a few milliamps saving. For example, consider using 22k or 10K pull-up instead of 4.7K or 1K pull up, if performance is not affected. At 3.3V when the pin is at zero, the pull-up will drain 3.3mA if the value is 1K and 330uA if pullup is 10K. [1]

## 4.9. Using unconventional Cores

The performance of modern computing systems is primarily shaped by power concerns. In such a regime, “unconventional” cores such as GPUs, FPGAs, ASICs and DSPs etc. hold a great promise for improving application performance and energy efficiency. For this reason, several researchers have used unconventional cores for power management of embedded systems. Evaluate sliding window program on multi-core CPU, FPGA and GPU. This program has applications in digital signal processing. They have observed that FPGA provides an order of magnitude better performance while using an order of magnitude less energy compared to both CPU and GPU. [9]

## 4.10. Utilize the DMA controller

Within a microcontroller, the CPU is typically the power-hungry device that uses the most current. Peripherals certainly contribute to the energy consumption especially if there is a USB or Ethernet controller, but one of the first steps to optimizing a device for energy is to keep the CPU in its lowest state as long as possible. Applications will periodically need to wake-up though to move data around such as receiving bytes from a UART and storing them in a buffer. A way to avoid constantly waking the CPU up to handle moving data around is to utilize the direct memory access (DMA) controller.

DMA allows a developer to keep their CPU asleep and instead use a lower power peripheral to do what the CPU would have done otherwise, move data around the microcontroller. The specifics on how to set up DMA will vary from one microcontroller to the next, but the benefit is that the CPU can be kept in a low power state until it needs to actually wake-up and process the data that has been transferred by DMA. [6]

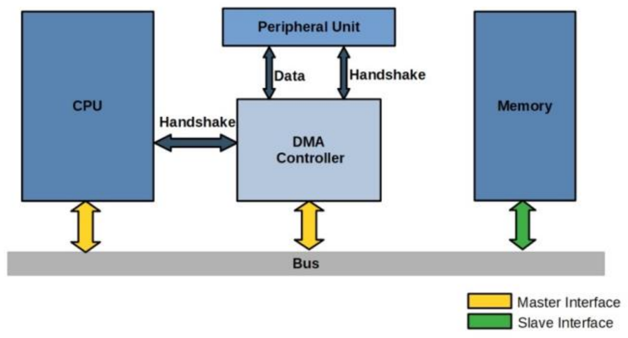


**Figure 13. DMA in a Embedded system**

## 4.11. USE CLOCK THROTTLING

The energy consumption of a microcontroller is dependent upon the clock frequency. The higher the frequency, the higher the energy consumption. An application doesn't necessarily always need to be running full tilt. In fact, throttling the clock based on what needs to be done can be an effective technique for minimizing the energy that will be consumed.

When and how to throttle the clock frequency is completely application dependent. Some applications may still want to run full tilt all the time. In others, it may be okay to cut the frequency in half from the maximum and run the part that way. In still others, it may make sense to dynamically change the clock on the fly based on what the application is doing. Just beware that if you decide to use clock throttling dynamically, you may also need to dynamically change peripheral register settings to compensate for the changes. [6]

## 4.12. USE COMPILER OPTIMIZATIONS

Modern compilers provide embedded optimization settings that can improve the execution time for a given piece of code. Most compilers provide developers with optimization settings to perform loop unrolling and inline functions. Developers can use these techniques to generate code that executes faster. Code that executes faster, even if it does use more code space, will give the opportunity to spend more time in a low power mode which in turn will decrease the energy consumed by the application. [6]

## 4.13. UTILIZE LOW POWER PERIPHERALS

Microcontrollers designed for low power operations will often include low-power peripherals that are separate from the “standard” peripherals. For example, the STM32 microcontrollers include a LP timer. This is a timer peripheral that doesn’t include all the extra bells and whistles but is designed to be operated in a low-power environment. When optimizing, or even starting a low-power design, watching for these low-power peripherals and utilizing them can be a great way to decrease the products energy consumption without a lot of extra effort. [6]

## 4.14. EVENT REDUCTION

Event reduction attempts to keep the processor in idle as long as possible. It is implemented by analyzing your code and system requirements to determine if you can alter the way you process interrupts. Your operating system should be able to set variable timeouts for its scheduler. The operating system knows whether each thread is waiting indefinitely for an external or internal event or is scheduled to run again at a specific time. The operating system can then calculate when the first thread is scheduled to run and set the timer to fire accordingly before placing the processor in idle mode. Variable timeouts do not impose a significant burden on the scheduler and can save both power and processing time. [12]

## 4.15. PERFORMANCE CONTROL

Processors that allow dynamic reductions in clock speed provide a first step toward power savings: cut the clock speed in half and the power consumption drops proportionately. It's tricky, however, to implement effective strategies using this technique alone since the code being executed may take twice as long to complete. Dynamic voltage reduction is another story. An increasing number of processors allow voltage to be dropped in concert with a drop in processor clock speed, resulting in a power savings even in cases when a clock-speed reduction alone offers no advantage. In fact, as long as the processor does not saturate, the frequency and voltage can be continually reduced. [12]

Although there are different known methods to reduce energy consumption in our embedded system, it is necessary to understand that each system will have particular characteristics, and you cannot apply the same method to all and hope that it works just as well, since it is necessary before initiating work on low power embedded systems design, these two things:

- To know the available power budget, understanding the overall power budget of the design, the consumption of the system or if it is powered by battery the back up time required

- To estimate the power consumption, a block diagram of the embedded design should be created, and  this diagram will help to identify the main components and their power consumption; the  information about the device power consumption could be found in the datasheet and application notes.

## 5. Conclusions

Overall this research has allowed us to understand in a better way the importance of the development of an embedded system's design and how if we take care of certain aspects from the beginning of the structure of this we can improve some major aspects that will eventually allow us to have an optimized system.

Throughout this whole text several ways to improve different aspects of an embedded system have been mentioned and described. One thing that stands out is that one of the first steps to have a proper optimization of our system is to fully know what we are working with. This means that we need to know what and how many components or units we have that will allow us to perform some functions especially from the hardware side. Once we know what and how many things are available we can begin to choose and rearrange things that could help us improve one or more aspects of the overall system.

Although this research was splitted between the members of our team, when redacting this conclusion and sharing insight about the information we found there were certain things all of us were aware and came up with a list of point we all agreed on:

1. It's highly important to know not only the hardware but also the software you're working with so one can take full advantage of the tools available so its use can be fully maximized.
2. Different language constructs exist, therefore we shall use them. It's clear that constructs such as inline, static, volatile, and so most of the time allow the compiler to generate optimal code.
3. Writing clean, simple code it's always a good idea. Not only when it's required to work with others and codes need to be shared but also for oneself future use. It's always easier to know what you were working on and how you were implementing some things to avoid some mistakes that could go undercover and may even pass the compile test since they can become a timing-bomb ready to cause problems when it's less expected.

Understanding the techniques that are mentioned in this research allow us to reflect on the way in which we design something. From now on, we will seek to contemplate several aspects mentioned here and we will try to implement some techniques that are not so complicated in each of the designs and codes that we do. With this kind of research a personal initiative is created, so we can continue searching the internet for other aspects and details related to embedded systems we can find and thus be able to continue improving the way we work, we realized that the bases of embedded systems already we have learned it in the course but there are still more details and tools that we can start to learn on our own.

# 6. References

[1] Aggarwal, P. (2019). *10 Most important tips for low power embedded system design.* Available at: https://medium.com/@pallav.aggarwal/10-most-important-tips-for-low-power-embedded-system-design-303dc051c194

[2] Amine, A. (2004). *Embedded Software for SoC.* Available at: http://160592857366.free.fr/joe/ebooks/ShareData/Software%20Books%20-%20Embedded%20Software%20for%20SoC.pdf

[3] Banks, S. (2010). *What is a victim cache?.* Quora. Available at: https://www.quora.com/What-is-a-victim-cache

[4] Barr, Micheal and Massa, Anthony.(2009). PDF format. *Programming Embedded Systems*, Second Edition with C and GNU Development Tools.

[5] Beningo, J. (2013). *10 C Language Tips for Hardware Engineers.* Available at: https://www.edn.com/10-c-language-tips-for-hardware-engineers/

[6] Beningo, J. (2019). *five techniques to lower energy consumption.* Available at: https://www.embedded.com/five-techniques-to-lower-energy-consumption/

[7] *How to Reduce Code Size (and Memory Cost) Without Sacrificing Performance.* (2005). Texas Instruments, Inc. Available at: https://www.embedded.com/how-to-reduce-code-size-and-memory-cost-without-sacrificing-performance/

[8] J. Teich, "*Hardware/software codesign: the past, the present, and predicting the future,"* Proceedings of the IEEE, vol. 100, pp. 1411–1430, 2012.

[9] Mittal, S. (2015*). A survey techniques for improving energy efficiency in embedded computing systems. HAL.* Available at: https://hal.archives-ouvertes.fr/hal-01101854/document

[10] Mohammed, S. (2019). *10 simple tricks to optimize yout C code in small embedded systems.* Available at: https://www.embedded.com/10-simple-tricks-to-optimize-your-c-code-in-small-embedded-systems/

[11] *Optimization in Embedded Systems.* (2017). Institute of Embedded Systems. Available at: https://www.tuhh.de/es/embedded-systems-design/teaching/seminars/optimization-in-embedded-systems.html

[12] Tennies, N. (2016). *How to reduce Power Consumption by writing better Software. BARR Group*. Available at: https://barrgroup.com/embedded-systems/how-to/low-power-management

[13] *Tips for Minimizing Power Consumption In Your Embedded System. PCB Design.* (2018). Altium Designer. Available at: https://resources.altium.com/p/tips-minimizing-power-consumption-your-embedded-system

[14] Vandewiel, S. (2000). *Data Prefetch Mechanisms.* Available at: https://www.csd.uwo.ca/~mmorenom/CS433-CS9624/Resources/p174-vanderwiel.pdf

[15] *Versatility of Embedded system.* (2016). Embedded System Learning. Available at: http://embedded-system-learning.blogspot.com/2016/12/how-to-reduce-code-size-in-embedded.html