Safety on Hardware MCU and Microprocessor System (MS) Design

Mohamed Fezari Faculty of Technology BMAU

Reading for CSM

Safety points to take care of in MCU or MSD Design are

- SRAM parity error check
- □ Flash area protection
- □ ADC self-diagnosis function
- □ Clock Frequency Accuracy Measurement Circuit (CAC)
- □ Cyclic Redundancy Check (CRC) calculator
- □ Data Operation Circuit (DOC)
- □ Port Output Enable for GPT (POEG)
- □ Independent Watchdog Timer (IWDT)
- □ GPIO readback level detection
- □ Register write protection
- □ Illegal memory access detection

Security and Encryption :

- AES128/256
- □ True Random Number Generator (TRNG)

Flash area protection :

Memory Array Write Protection Two types of protection methods exist for protecting the memory array. Their objective is to prevent changing the contents of parts of the memory which the user wants to protect. In other words, they block erase and program operations in certain memory sectors or blocks.

Protection methods for Status Registers prevent the user from writing to those registers and changing their values. Note that in many cases certain bits of the Status Registers configure the protection state of the memory array; thus, protecting the Status Registers can also indirectly protect the memory array.

Certain methods exist to protect from accidentally sending commands that change the state of the flash, or otherwise reset the flash.

Hardware protection uses a flash input pin named WP (Write Protect) to signal the flash chip to protect or unprotect certain resources.

Software protection uses flash commands to protect or unprotect certain resources. Examples are: ▪ Writing to Status Register protection bits that control the protection state of the memory array. ▪ Protect/unprotect commands (sometimes called lock/unlock) that change the protection state of certain sectors/blocks of the memory array, or the entire memory array. ▪ Mandatory write-enable command (or reset-enable command in the case of reset) before sending another command that makes a change to a certain resource (memory array, register, flash state). This extra step protects from accidental or unintentional changes.

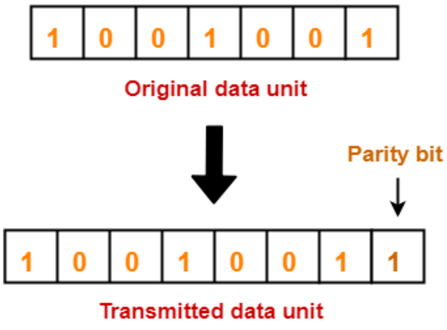
SRAM parity error check

A parity check is an error-correction process in network communication that ensures data transmissions between communication nodes are accurate. In this process, the receiver agrees to use the same even parity bit or odd parity bit scheme as the sender

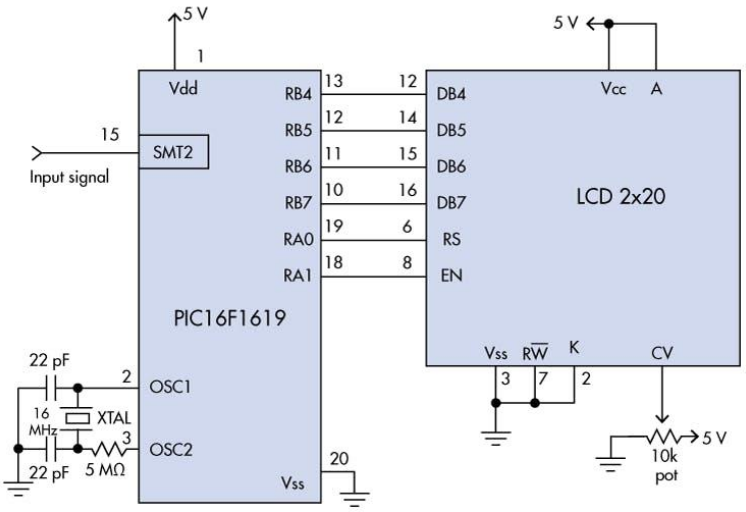
It is used to validate the integrity of the data. The value of the parity bit is assigned either 0 or 1 that makes the number of 1s in the message block either even or odd depending upon the type of parity. Parity check is suitable for single bit error detection only.

Parity checking adds an extra parity cell to each 8-bit byte of memory, thus creating a nine-bit structure. In an "even parity" system, a 0 is stored in the parity bit if there is an even number of bits in the byte; if an odd number, a 1 is stored to make the total number of bits even.

We can detect single errors with a parity bit. The parity bit is computed as the exclusive-OR (even parity) or exclusive-NOR (odd parity) of all of the other bits in the word. Thus, the resulting word with a parity bit will always have an even (for even parity) or odd (for odd parity) number of 1 bits in it.
The simple parity check can only detect an odd number of bit errors, but it will detect it 100% of the time. There are more sophisticated error correction techniques. The Hamming code can detect up to two-bit errors or correct one-bit errors.



Clock Frequency Accuracy Measurement Circuit (CAC)



Microcontrollers are reliant on their clock source. The processor, the bus, and the peripherals all use the clock to synchronize their operations. The clock determines how fast the processor executes its instructions, so it is fundamental to performance. But how important is the clock source? What is clock frequency in microcontroller?Does it matter how accurate it is? The short answer is that it depends… it depends on what the microcontroller is doing and its interfaces.

Two considerations need to be taken into account: the clock's speed, which determines how fast things happen, and the clock's accuracy, which determines the consistency of the period between each clock tick and how the clock speed can change over time.

The microcontroller's central processor can be thought of as a synchronized chain of logic blocks that perform a specific function. If thesystem clock in microcontroller runs too slow, the processing takes longer. If the clock runs too fast, there may not be enough time to complete the required operations before the next set begins—the processor interfaces with a range of different component blocks, from dynamic memory to interface pins. Any significant error in clock speed will have unpredictable consequences for internal microcontroller operations.

The microcontroller clock signal will govern the conversion rate of any analog-to-digital operations. The clock's speed will determine the maximum rate at which the analog signal can be sampled; the clock's accuracy will determine the sampling rate's accuracy. Suppose you are recording a sample twice a second with a timestamp. In that case, it doesn't take long before a one percent error in the clock's frequency (not uncommon with internal oscillators) removes any correlation between your sample's timestamp and the time shown on your wall clock. With a 1% constant offset of the clock source, your sample's timestamp will be out by over 14 minutes every day.

Ref: https://resources.altium.com/p/how-important-your-microcontroller-clock-source-0

A critical application for the microcontroller clock signal will be to manage asynchronous communications where the clock signal determines when the incoming data stream is sampled; once the start bit is received and the waveform of the outgoing data stream in terms of when transitions between each bit of data occur.

With asynchronous communications, the transmitter and receiver are reliant on having the same clock speed for encoding and decoding data streams. However, these clocks do not need to be synchronized; they just need to have sufficiently equal clock rates. This is because the receiver starts processing the incoming data stream when detecting the first edge on the signal line. It then needs to maintain the correct clock speed for the data stream's duration to sample the data bits at the correct times. The required accuracy will be dependent upon the window where the data has to be sampled. Each data bit will potentially have a rising edge and falling edge to its signal where the data's value is indeterminate, leaving the period between the edges when the data is valid and can be sampled.

Conclusion

To sum up, what type of clock signal you need to use for your microcontroller will depend mainly on the nature of the device it's embedded in and its operating environment. Interfaces with high-speed asynchronous communications buses and high-frequency analog signals will drive the need for an accurate clock signal. Suppose the device needs to operate in a harsh environment, whether that is over a wide temperature range, in high levels of electromagnetic interference, or subject to mechanical vibrations. In that case, it can limit the choices available. A microcontroller clock that doesn't have such time-sensitive or environmentally challenging requirements can get away with a cheaper solution.

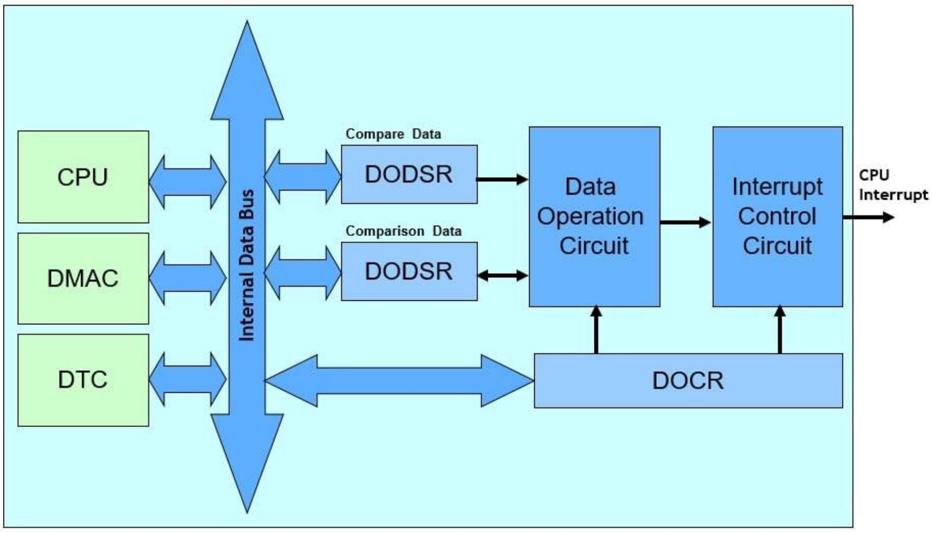
Data Operation Circuit (DOC)

https://www.renesas.com/us/en/blogs/what-s-doc-smart-way-manage-smart-peripherals

My favorite peripheral, and probably one of the most interesting but less understood peripherals inside a typical RA microcontroller, is the Data Operations Circuit (DOC). This can provide some significant performance advantages in real time applications, allowing tasks to be offloaded from the CPU, improving the response time to asynchronous events, and potentially reducing power consumption. This is especially true when the DOC is used in conjunction with some of the other more advanced features available on the RA family.

At the heart of the DOC, is a simple Arithmetic Logic Unit (ALU). This simple ALU has only three basic functions, it can make a 16-bit data comparison, a 16-bit addition, or a 16-bit subtraction, and then to generate an interrupt based on a specified condition. These tasks can all be executed without any intervention from the CPU. See diagram below.

Image



When using the 16-bit comparison mode, an initial reference value is loaded into the DOC, the 16-bit data to be compared in then loaded and compared with the reference value in hardware. The DOC can be programmed to generate an interrupt on match true or match false.

When using the 16-bit addition mode, an initial 16-bit value is loaded into the DOC, additional 16-bit values are then loaded (can be one or more) into the DOC and are added to the original value. When all the required values are loaded, the count is checked for overflow, and an interrupt generated if required. This simple mechanism allows for a decision to be made if a specific threshold value has been exceeded, ideal for instance for automatic level sensing using the ADC.

When using the 16-bit subtraction mode, an initial 16-bit value is loaded into the DOC, additional 16-bit values are then loaded (can be one or more) into the DOC and are subtracted from the original value. When all the required values are loaded, the count is checked for underflow, and an interrupt generated if required. This simple mechanism again allows for a decision to be made if a specific threshold value has been breached.

However, the true power of the Data Operation Circuit is in the fact that these three simple functions can be used to make simple decisions as to how the system will operate, without any CPU intervention. This means that we can make simple decisions, directly in the hardware of the microcontroller, allowing peripherals to decide how to manage the data they produce based on a simple comparison.

When combined with peripherals such as the Direct Memory Access Controller (DMAC) or the Data Transfer Controller (DTC), which can be used to automate the passing of data to the DOC, we can see how this can be used to create a system that’s capable of making decisions based on data from almost any source. This is done without any CPU intervention, even when the CPU is asleep. In many systems it can also provide a much faster response to changing data, rather than waiting for the CPU to be interrupted to then respond to the event.

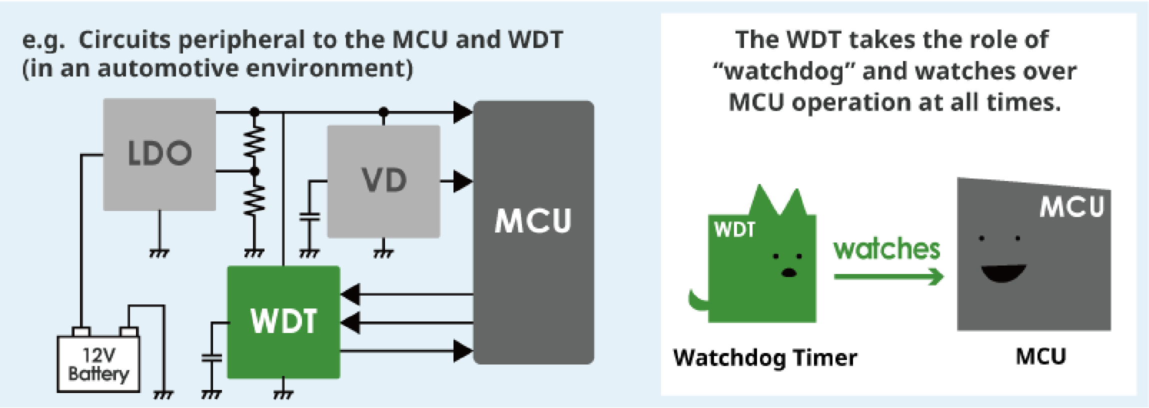
We can imagine many uses for the DOC, for instance, when used with the UART interface, the DOC could be used to automatically detect and incoming address and alert the CPU when the address is valid. The DOC could be used with the Analogue to Digital convertor (ADC) in a level sensing system, to detect automatically when the level exceeds a programmed threshold. There are many advantages to using the DOC for functions like these. For instance, the CPU could be dedicated to other high priority tasks, only alerted by the DOC, by interrupt, when a particular condition is reached. The CPU could even be placed into sleep, to reduce power consumption, and only awakened again by an interrupt on a valid alarm condition detected by the DOC comparison.

You can find the Data Operating Circuit on all the members of the RA family, and we have continued to enhance the functions of the DOC and the latest versions have additional functions allowing more complex decisions to be made on data automatically.

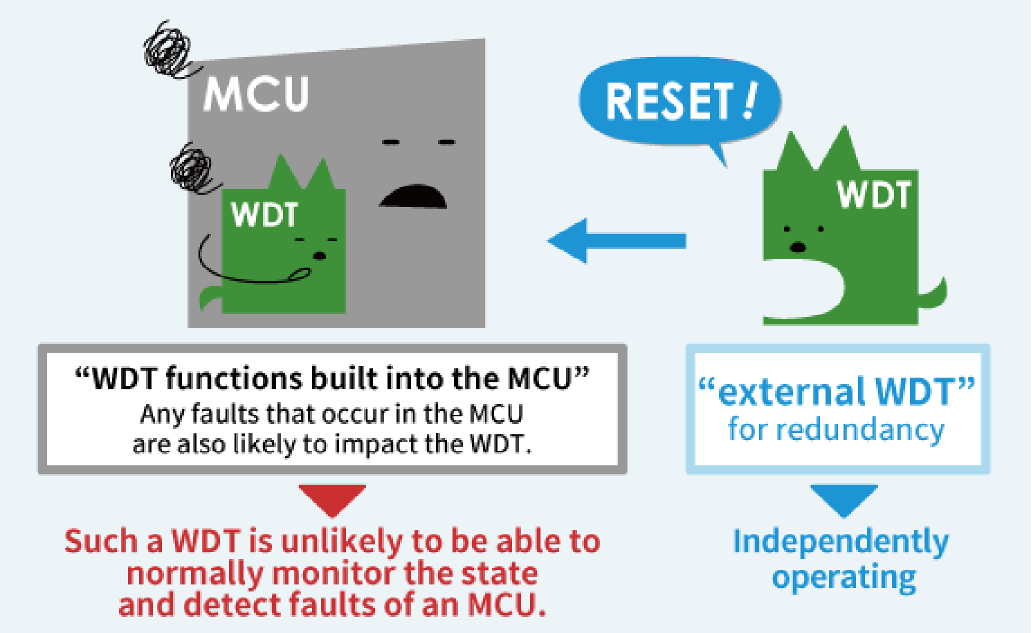
In the next blog, I'll look at the Data Transfer Controller and how this can be combined with the DOC to provide a complete ADC subsystem that operates without any CPU intervention, unless an alarm condition is met, when the CPU can be interrupted if required.

Independent Watchdog Timer (IWDT)

The independent watchdog is **used to detect and resolve malfunctions due to software failures**. It triggers a reset sequence when it is not refreshed within the expected time-window. Since its clock is an independent 32-kHz low-speed internal RC oscillator (LSI), it remains active even if the main clock fails.


e.g. Circuits peripheral to the MCU and WDT
(in an automotive environment)
LDO
VD
MCU
WDT
12V Battery
The WDT takes the role of "watchdog" and watches over MCU operation at all times.
WDT
watches
MCU
Watchdog Timer
MCU

https://www.ablic.com/en/semicon/products/automotive/automotive-watchdog-timer/intro/


MCU
WDT
RESET!
WDT
"WDT functions built into the MCU"
Any faults that occur in the MCU are also likely to impact the WDT.
"external WDT" for redundancy
Such a WDT is unlikely to be able to normally monitor the state and detect faults of an MCU.
Independently operating

References

1. https://resources.altium.com/p/how-important-your-microcontroller-clock-source-0
2. https://www.renesas.com/us/en/blogs/what-s-doc-smart-way-manage-smart-peripherals
3. https://www.ablic.com/en/semicon/products/automotive/automotive-watchdog-timer/intro/