embeddedworld 2015
Exhibition&Conference
... it´s a smarter world

Design of SoC for High Reliability Systems with Embedded Processors

Joseph Yiu
Senior Embedded Technology Manager
CPU Product Group, ARM
Cambridge, United Kingdom

Abstract— Reliability is vital for many embedded applications including industrial controllers and automotive electronics. There are many well-established techniques for creating reliable high-end control systems, and now these techniques are also appearing in small embedded systems including many microcontroller products based on ARM® processors. This whitepaper covers an overview of system design techniques and processor specific features that are commonly used in such applications including dual-core lock-step and the processor's internal memory protection (e.g. parity, ECC), as well as system level enhancements like bus level timeout monitors and hardware monitoring units. The whitepaper also covers various areas that chip designers need to watch out for when developing their designs, such as how these features impact on system behavior.

Keywords— Processors, reliability, safety, ECC, Cortex-M7

I. BACKGROUND

A. Scope of high reliability systems has expanded

There is a wide range of embedded systems requiring high reliability. Traditionally, industrial controllers and automotive electronics are key areas of the electronics industry that demand high reliability. Today, however, embedded systems are also deployed in medical equipment, smart building management, and other areas that have higher reliability requirements than traditional consumer electronics such as home appliances and entertainment systems.

Technology trends like the Internet of Things (IoT) are also driving demand for improved reliability. With IoT, there are new implications for a range of electronic product designs. For example, a home broadband router built into the IoT infrastructure allows a fire alarm and a security system to be linked with a home owner's smartphone and local emergency services. As a result, such home alarm systems and the IoT gateway must utilise a number of reliability features in order to ensure that the correct information is distributed. As a result of various IoT usage implications, embedded system designers are demanding more reliability feature support in the embedded systems underneath these products.

Previously, the design for high reliability in System-on-Chip (SoC) products, including microcontrollers, was a very specialized topic. Today, as the demand for higher reliability has increased, many standard off-the-shelf microcontroller designs are built with enhanced system reliability. As ARM Cortex®-M processors are used in many of the modern microcontroller products, this paper will also cover how ARM processors are designed to enable better system reliability.

B. Typical requirements of a high reliability system

In general, we can generalize the technical requirements for high reliability systems into several key areas:

1) Reducing the possibility of errors.
2) Detection of errors.
3) Support correction of errors.
4) Robustness – a single point of failure should not be able to manifest into a complete system failure.

Depending on the application, these requirements may be optional. Memory error detection and correction, for example, typically requires error-correcting code (ECC) support, an uncommon feature in low-cost microcontroller products.

C. Common sources of system failures

Before we look into enhancing system reliability, we must first look at what can go wrong. In most SoCs with embedded

processors, the causes of failures can be classified into the following categories:

- Memory: This is the most common concern for SoC designers. As memory density gets higher and higher, the energy required to toggle a memory bit is reduced, and therefore accidental changing of memory states can be triggered easily by various scenarios like a transient pulse in power wire or signal connections, electrostatic discharge, a hit by a radiation particle or even interference from nearby RF transmitter.

- Logic: This is a less common cause as modern chip design tools provide very good signal integrity checking, and most hardware failure can be detected by manufacturing tests (e.g. a scan test). However, logic failure can still be caused by electron migrations and transient pulses in a power supply or signal connections.

- Software: This is possibly the most common cause of system failures in finished products. Common mistakes like inadequate checking of external inputs, incorrect setup for stack and heap memory size or simple programming bugs can all result in different types of system failure.

At the system level, there can be additional factors to consider:

- The chip packaging can be an important factor for applications that need to endure mechanical shocks and vibration.

- PCB design and physical connectors can seriously affect the stability of the power supply and signal integrity.

The rest of this paper will focus on what can be done at the SoC design level to prevent, detect and correct errors, and therefore these system level factors will not be covered in details.

II. MEMORY PROTECTION

A. Reducing risk of memory errors

The first step in getting better memory reliability is by using the right memory macrocells. For example, some SRAM has advanced built-in test features that can help detect faults and prevent field failures. In SRAM and ROM IP products from ARM, the flexible margining features also enable chip designers to create memory systems with larger margining to enhance reliability (reference [1]).

Many microcontrollers use embedded flash memories. Some of the embedded flash suppliers (e.g. Cypress SONOS embedded flash) are designed to support high reliability by providing extended lifetime in a wide temperature range (reference [2]), and reduce reliability concern using uniform tunnel erase (reference [3]).

B. Parity and ECC

In some embedded systems, the memory contents are protected with either parity bits or ECC bits. Parity bits require a smaller memory size overhead, and simpler computation hardware. However, a parity bit cannot handle error correction and typically can only detect single bit errors. ECC can often support SECDED (Single-bit Error Correction, Double-bit Error Detection). However, it requires more memory overhead and more complex computation.

For systems with high reliability requirements, ECC is generally preferred over parity due to its ability to correct single bit errors and detect multi-bit errors.

In many systems, each 32bits of data has a 7-bit ECC overhead, and each 64bits of data has an 8-bit ECC overhead. For a parity scheme, however, a designer can choose 1parity bit per byte, 1parity bit per 32bits of data or even 1 parity bit per 64bits of data, although this can affect the ability to detect errors. For cache memories, additional ECC overhead would be required for cache tag memory.

In order for memory errors to be handled, the processor in use must provide a suitable interface. For example, the TCM (Tightly Couple Memory) interface of the ARM Cortex-M7 processor (reference [4]) provides the essential signals for ECC error handling (figure 1).

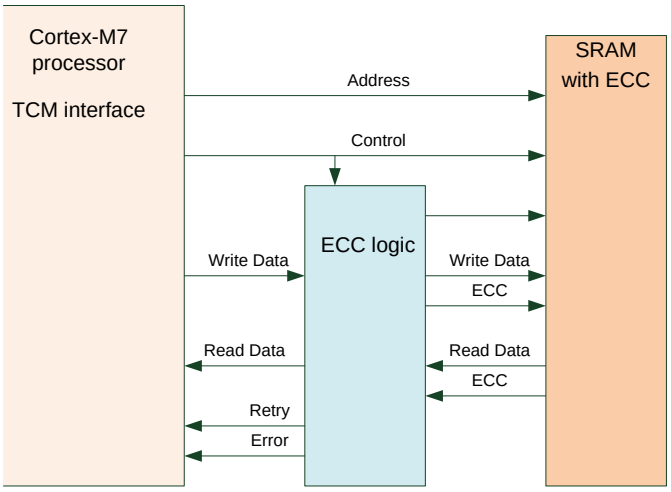

Fig. 1. Connections between the TCM interface of the Cortex-M7 processor and the TCM.

- If an ECC error is detected, the ECC computation logic can signal to the processor that the read operation needs to be retried (a retry signal). If the

error can be corrected, the corrected data is forwarded to the processor, and can also be written back to the SRAM at the same time.

- If the ECC error cannot be corrected (a multi-bit error), then the ECC computation logic can send an error status back to the processor to indicate a fault (an error signal). In this case, a fault exception could be triggered on the Cortex-M7 processor and the error is dealt with inside the fault exception handler.

Since the ECC computation takes time, the retry signal in the Cortex-M7 processor is one cycle behind the read data cycle. This enables ECC function without slowing down the memory clock speed.

The cache memory system in the Cortex-M7 processor can also be protected by ECC. This is a configurable option and the ECC is handled internally within the processor.

In the ARM Cortex-M3 and Cortex-M4 processor, a signal is also available to allow ECC logic to signal to the processor so that an instruction fetch can be retried if an ECC error is detected.

C. ECC feature implications

While ECC is great for enabling data integrity, it has some implications. In addition to the memory size overhead, it can also cause performance reduction. Because a processor can write data of different sizes to the memory (e.g. 8-bit, 16-bit, 32-bit or 64-bit), and typically the ECC value is generated for the whole 32-bit word or 64-bit doubleword, a Read-Modify-Write (RMW) sequence might be required to allow the ECC to be recomputed, unless the ECC code is on a per byte basis.
In order to support this, the TCM interface of the Cortex-M7 processor can be configured to enable the RMW sequence for each write operation that does not cover the whole 32bits or 64bits of data. This is completely transparent to software, but it can result in extra clock cycles for some of the write operations.

D. ECC initialization aspect

When a system with ECC memory is powered up, the memory is likely to contain invalid ECC values. This means that the system may require additional initialization steps to avoid memory locations being speculatively read before they are initialized. This can trigger unnecessary retry sequences or even accidentally trigger a fault exception in some processor architectures.

On the Cortex-M7 processor:

- The ECC for the cache memory is initialized when it is being invalidated (part of the cache system initialization), so there is no additional software overhead to support cache ECC initialization.

- The first write to each TCM word / doubleword / byte / halfword initializes the ECC for the whole word (for the data TCM) or doubleword (for the instruction TCM), so again, there is no additional software overhead.

Due to the high performance nature of the Cortex-M7 processor, speculative read operations can occur and this means an uninitialized TCM memory locations could be read by the processor speculatively. However, the TCM interfaces ignore the ECC retry/error if the read is speculative and if the read data ends up unused. So as long as the software (including C runtime library code) does not read an uninitialized memory location, it should work just fine.

However, one particular area that chip designers need to pay attention to is accesses from debuggers, as they can read memory locations that are uninitialized. In the Cortex-M7 processor, an additional TCM side band signal is available to allow ECC logic to ignore ECC errors caused by debugger accesses. This is not an issue for the cache because debug access would not happen to uninitialized cache locations.

For other processors that cannot distinguish ECC errors caused by speculative reads or debug reads, or if you cannot guarantee the software does not read uninitialized memory locations, you might need to initialize the memory's ECC before starting the application. For example, the processor can initialize the memory by writing a constant (e.g. 0) to certain memory ranges to initialize the ECC values during the startup code execution. The disadvantage is that, of course, it could take longer for the system to start up.

ECC initialization can also affect unused memory locations in NVM (Non-Volatile Memory, e.g. flash). For this situation the development tool chain can fill the unused program space with known values so that the whole NVM is programmed with valid ECC values.

E. Error management features

1) TCM

The TCM ECC error handling depends on the design of ECC logic attached to the TCM interface. Typically, the ECC calculation result is available one clock cycle after a data read operation. As a result, the retry signal is designed in a way such that it should be asserted one clock cycle after the data read operation if an ECC error is detected. The retry signal triggers the processor to retry the read operation, which by that time the ECC computation should have been able to calculate the corrected data, if the error is single bit. This corrected data can then be written back to the TCM memory location as convenient. When the accesses is retired, the processor should

hopefully then read the correct data. However, the retry of the read access may be delayed if an interrupt occurs during the retry phase. (See figure 2).

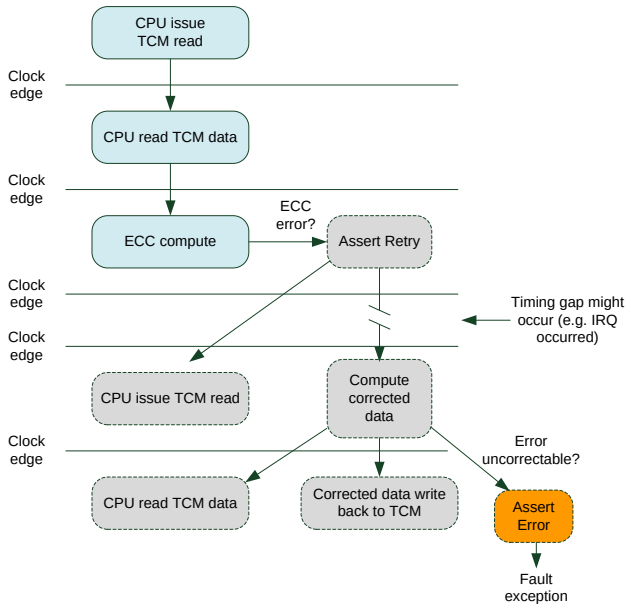

Fig. 2. An example of TCM ECC error handling sequence

If the TCM contains an uncorrectable ECC error, an error response is needed and the implementation could trigger a fault exception, or it could continually retry the access resulting in a "deadlock" situation so that the system does not do anything else. It could also generate an NMI, which could be the exception used to handle all safety related errors. In the Cortex-M7 processor, an uncorrectable error will trigger a fault exception.

2) Caches

The Cortex-M7 handles cache ECC errors automatically. If ECC is used and a correctable error is detected in a data read, the cache is cleaned and invalided (correcting the data on the fly), and the data is read back in a retry cycle.
If an uncorrectable ECC error is detected, there are several ways to deal with it.

- If the ECC error is in the instruction cache memory, the corrupted cache line can be discarded and the program is fetched from the main memory system again.
- If the ECC error is in the data cache memory, the affected cache memory could be cleaned and invalidated
    a. If the cache line is clean (tag RAM is not corrupted and tag indicates that data is not dirty), the corrupted cache line can be discarded and the data is fetched from the main memory system again.
    b. If the ECC error is in the data cache memory, and the cached data is dirty, a fault exception would be needed.

Additional cache error indication signals are available on the processor's interface to indicate other ECC scenarios.
The Cortex-M7 processor also has a couple of error bank registers for the instruction cache and data caches. These registers allow the cache line that suffered from an ECC failure to be locked down, preventing it from being used again.

F. Memory BIST

Majority of memory macros designed for high reliability systems support some forms of BIST (Built-in Self-Test) features. Normally such BIST can be carried as part of manufacturing tests and as a part of device startup sequence. In order to support such feature, some processors such as ARM Cortex-M7 processor included MBIST interface for its cache memories.

The Cortex-M7 processor also allows the RAMs to be tested using the MBIST interface during normal execution. This is known as online MBIST.

III. PROTECTION AGAINST HARDWARE LOGIC ERRORS

A. Reducing the risk of logic errors

There are a number of ways to enhance the reliability of the logic hardware. Typically, in automotive applications, the logic cell libraries are only characterized for operation conditions with minimum risk of logic failure. As a result, the same design using the same semiconductor process could run slower (e.g. with larger margin for timing), but can run with higher reliability (i.e. lower possibility of logic failure).

In some cases, chip designers can also use special processes that are radiation-hardened (RAD–hardened) for specialized IC design (e.g. aviation electronics – which have a higher chance of being affected by cosmic ray and alpha particles). However, this is typically much more expensive than standard semiconductor processes.

For other chip designers, it might be impossible to use either of these two methods. However, we can still improve the reliability of the system by:

- Ensuring good floor planning and power rail design to reduce IR drop (internal power rail voltage drop).
- Using EDA tools to ensure good signal integrity.
- Enhancing the error detection features in the system.

Fortunately, the possibility of logic hardware failure is often much lower than memory failure.

B. Logic error detection with Dual-Core Lock-Step

In order to detect errors in logic during run-time, some embedded systems deploy two instantiations of the same processor, and compare the outputs as a way of detecting hardware errors. This is commonly known as dual-core lock-step. In such designs, the two processors starting with the same state (reset) and receive the same inputs (execute the same program), and therefore the outputs should be identical. If the outputs from the two processors mismatch, then we know that a logic failure has occurred and can be handled by a separated mechanism (with two cores, it is impossible to tell which one is correct).

The Cortex-M7 processor supports dual-core lock-step configuration option. In this configuration, the core logic is instantiated twice, but the cache memory and TCM interface are shared because they can be protected by ECC which has a lower silicon area overhead (figure 3). Since the two processor cores' logics execute the same program, it does not enhance the performance of the systems by having two processor cores.

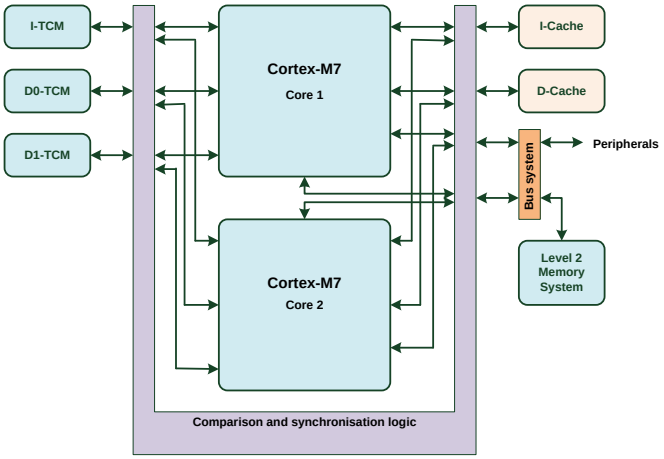

Fig. 3. Dual-Core Lock-Step configuration in the Cortex-M7 processor

When looking at the dual-core lock-step in more details, there is a way to enhance the error detection rate. If the two core blocks are implemented in the same way and the failure is caused by a transient pulse on power or signal interface, it is possible to yield the same failure on both cores. In order to enhance the success rate to detect errors, the execution cycle of the two core logic blocks are actually offset by two clock cycles (See figure 4). If an error triggers a fault in one core, it is unlikely for it to happen to the second core in the same way, as it is less likely that they are doing the same operations.

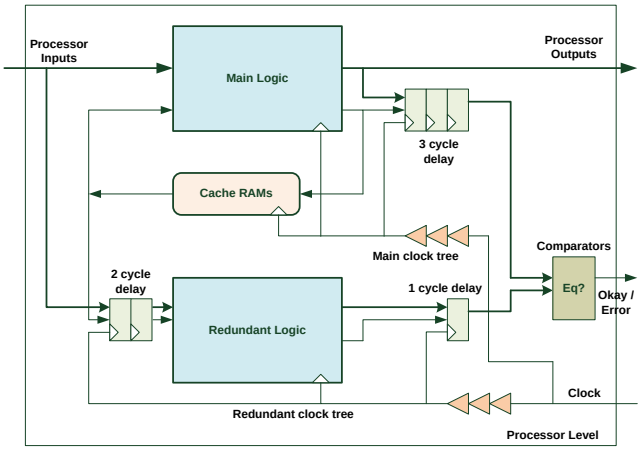

Fig. 4. Delay elements used in dual-core lock-step to reduce risk of common mode failures.

Of course, it means that the inputs to the redundant core logic needs to be delayed by two clock cycles, and the output from the main core logic needs to be delayed by two clock cycles to make the output matches.

Another requirement of a dual-core lock-step is that both processors need to start up with the same state. As a result, we need to add one more feature to the processor – all the flip-flops inside the processor need to be able to reset to a known state. In processor designs, it is typically acceptable to have some flip-flops not having reset (e.g. registers for the register bank). This arrangement reduces the silicon area as well as power, but for dual-core lock-step systems, this can cause false positives. For example, the undefined initial value of a register in the register bank could propagate to the bus system as the system starts. As a result, the Cortex-M7 processor supports a configuration option to reset all registers inside the core to ensure that the initial states of the two core logic blocks are identical. This is the default setting if dual-core lock-step configuration is implemented.

The same 'reset all registers' option is available for other Cortex-M processors allowing designers to implement a dual-core lock-step design with existing Cortex-M0/M0+/M3/M4 processors. In these processors, there is no special configuration option for dual-core lock-step as there is no memory block inside the processor, making the creation of dual-core lock-step systems fairly straight forward.

C. Logic Error Detection with a hardware monitor unit

In some SoC designs, duplicating the whole processor core logic is not suitable due to silicon size or power constraints. For the Cortex-M3 processor, Yogitech has designed their own third party technology called faultRobust.

The "fRCPU_armcm3" (reference [4]) is a licensable IP developed by YogiTech which is designed to be tightly coupled to a Cortex-M3 processor configured with a special observation interface. This interface allows the fRCPU

module to observe the internal operations of the Cortex-M3 processor and flag up any unexpected operation results. This monitor unit is much smaller than the extra hardware required for a full dual-core lock-step arrangement.

Failure Mode and Effect Analysis (FMEA) has been carried out by YogiTech to ensure that the fault coverage of the fRCPU can reach over 99%, and this helps products based on this technology to be certified to the IEC61508 SIL3 standard.

IV. PROTECTION AGAINST SOFTWARE FAILURES

A. What can we do at software level?

There can be many causes for software related failures. The majority of them are simply bugs in software components, such as inadequate validation of external inputs, poor software design that leads to race conditions, or in some cases, a failure to reserve enough memory for stack or heap memories.

There are also failure conditions that a software designer cannot have foreseen. For example, if an internet connected device is attacked via DoS (Denial of Service), or if it suddenly receives unexpectedly high amounts of input packets, a system could fail due to performance or memory size limitations. Potentially, a significant performance margin or memory size margin is required.

Since Cortex-M programming requires no or very little tool chain specific extensions, you can use most of the existing code analysis tools for quality checks (e.g. MISRA-C compliance tests, formal code analysis tests).

For some of these cases, the software bugs can be avoided with sufficient testing and coding quality checks, but it is not uncommon for some of these potential issues to remain undetected for a long period of time. Fortunately, additional protection can be implemented at software level to reduce the severity of the failures, by preventing a single failure from manifesting into a complete system failure, for example.

In some cases, certain data processes can be repeated, potentially using different algorithms, to verify the consistency of the data processing result. However, this verification technique can only protect the data processing part. Many of the software operations cannot be protected in such an arrangement.

B. Stack protection

Stack overflow is a common cause of failure in many embedded systems, typically caused by over optimistic stack memory allocation, and in some other cases, software bugs that result in stack leaks. Obviously software designers need to obtain a correct estimation of required stack size, including stack memory used by the application code and exception handlers. Once the required stack size is established, there are several ways to implement stack protection:

I. **Using the Memory Protection Unit (MPU)** – The MPU is an optional component for most of the Cortex-M processors and this feature should be included for SoC products that might be used in applications that require high reliability. Software developers can use the MPU to define memory regions for program code, data, stack, peripherals, etc. If the processor tries to access a memory location not defined with a valid MPU region, a fault exception is triggered. This method can be used with bare-metal systems (no embedded OS) or systems with an RTOS that supports an MPU. If the stack memory access goes beyond a certain memory space, a fault exception would be triggered and remedy actions could be taken (e.g. terminating an offending task and restarting it).

II. **Using stack checking features in embedded OS / RTOS** – many operating systems have optional stack limit checking functionality and this can be very useful for systems that require high reliability. The only limitation is that checks usually only take place during context switching and therefore a stack overflow might not be detected or prevented.

III. **Separating the application's program stack and exception handler stack, and creating a hard limit on the application's stack usage** – Cortex-M processors have two stack pointers to support efficient OS operations. However, for embedded systems without an OS, it is still possible to utilize this feature so that the exception handler's stack is separated from the application code stack. In these cases, you can place the application program's stack at the bottom of valid RAM space so that a stack overflow would trigger a fault exception. The exception handlers (including the fault exception triggered, and other interrupt handlers) that use the handler's stack can still execute and carry out fault handling actions.

C. Fault exception handlers

In many cases of Cortex-M based embedded systems going wrong, fault exceptions are triggered. Situations causing embedded systems to go wrong typically include:

- Execution of undefined instructions (e.g. flash memory corruption).
- Attempting to access invalid memory locations (e.g. stack overflow)
- Violation of access permissions defined by the MPU or default permission settings (e.g. attempted attack from hacking activities).
- System integrity check failures (e.g. exception return to thread state when there are still active interrupt/exception. This type of checking is available

for processors based on the ARMv7-M architecture, including Cortex-M3, Cortex-M4 and Cortex-M7 processors.)

- Incorrect data address alignment (e.g. all unaligned data accesses for Cortex-M0 and Cortex-M0+ processors and unaligned data accesses in multiple load/store instructions in ARMv7-M processors).
- Optionally, a divide by zero operation (software can setup a control register to enable/disable a fault exception for this condition).
- Optionally, ECC errors that are not correctable on a Cortex-M7 processor (if the ECC feature is implemented and enabled).

For ARMv6-M architecture (Cortex-M0 and Cortex-M0+ processors), all fault exceptions trigger the HardFault exception handler. This has higher priority than other interrupts and system exceptions, apart from the Non-Maskable Interrupt (NMI). For ARMv7-M processors (e.g. Cortex-M3, Cortex-M4 and Cortex-M7 processors), there are three more configurable exceptions that can be programmed to enable separate handling of BusFault (e.g invalid memory locations), UsageFault (e.g. invalid instructions), and MemManage (Memory Management Fault for MPU permission violations). In addition, ARMv7-M architecture also adds a number of fault status registers and other registers to help analyse causes of fault exceptions.

Software developers can utilize the fault exception to:

- Handle remedy actions when the system goes wrong.
- Report errors to the users or the system or other systems connected to it.
- Optionally perform a self reset if needed.

Care should be taken in creating fault exception handlers. For example, the fault could have been caused by issue with the stack (stack pointer pointing to an invalid memory location) and therefore the BusFault, MemManage and HardFault exception handlers should check the Main Stack Pointer (MSP) is pointing to a valid memory space before attempting to perform additional stack operations.

D. Built-in Self Test software

Software developers can add Built-in Self-Test features in their software. Typically, such self-test operations can be carried out as the system started. Runtime test library (fRSTL Software Test Libraries, reference [6]) for Cortex-M processors is also available from YogiTech.

The fRSTL is a set of software components that can be integrated into application code for detection of hardware failures. The fault coverage has been verified by means of intensive fault injection simulations. It can be used on standard off-the-shelf microcontroller products. A generic version of the fRSTL called fRSTL_arm is available that focuses on the testing of the processor, and an MCU vendor specific version of the fRSTL called fRSTM32 that covers the whole microcontroller (the test library covers the testing of peripherals, memory systems of the STM32 microcontroller, not just the processor) has been developed for STMicroelectronics's STM32 product range. The libraries can be used to target applications that requires SIL2 (IEC 61508 Safety Integrity Level) or SIL3.

V. OTHER SYSTEM LEVEL PROTECTION

A. System Design for reliability

There are a number of system level protection components that SoC designers can integrate in their chip design. For example, a watchdog timer and brown out detector can be used to generate a NMI (Non-Maskable Interrupt) and reset requests.

In order to prevent a single bus slave from locking up a system completely, bus timeout monitoring units can be added to the bus system. For example, in the ARM Cortex-M System Design Kit (CMSDK) there is an AHB timeout monitor and an APB timeout monitor. These monitor components can be placed between the bus interface of bus slaves and the rest of the bus interconnect. If a bus slave is dysfunctional (asserting too many wait states when being accessed), the monitor can trigger a timeout and return an error response to the bus master (e.g. the Cortex-M processor) and the fault handler can then optionally reset the failed peripheral or carry out other remedy actions. In these monitoring units, the number of clock cycles to trigger the time out is configurable.

In many applications that need to operate over a wide temperature range, the choice of embedded flash memory, SRAM, oscillators and voltage reference can also be crucial. You can also integrate temperature sensors in the system and trigger an alert/interrupt if the temperature range is beyond the acceptable range.

B. Automatic reset

The Cortex-M processor has a LOCKUP status output signal that can be optionally used to reset the system automatically if a LOCKUP state is reached (LOCKUP state can occur for different reasons, e.g., a result from a fault event inside HardFault or NMI handlers). An automatic reset feature should be programmable, and should be disabled during software development to enable easier debugging of program failures.

C. Auxiliary Fault

Some hardware failures can come from other components on the chip. On the Cortex-M3 and Cortex-M4 processors there is an Auxiliary Fault status input and an Auxiliary Fault Status

Register. Chip designers can connect these additional failure status signals to the Auxiliary Fault status input and also use one of the interrupt inputs to handle these failure events.

VI. Other Considerations

A. *Certification requirements*

Certification for functional safety is a complex topic that cannot be fully covered in this document. It is worth noting, however, that ARM is working on various Safety Documentation packages for ARM processors. These are designed to help SoC designers to get their SoC products certified to a specific standard (e.g. ISO26262). A documentation package is also available for the Cortex-R5 processor and a range of Cortex-M processors.

B. *On chip firmware*

Many SoC or microcontrollers contain on-chip firmware. As a result, when creating these firmwares for high reliability systems, the software components and toolchains being used should also be considered.

1) *Tool chains*

A number of development suites are certified for use in safety related development. For example, both ARM Compiler (shipped with Keil MDK and ARM DS-5, reference [7]) and the IAR Embedded Workbench for ARM (reference [7]) have been certified by TÜV SÜD. In addition, ARM Compiler also has a Qualification Kit package available to help the certification process (reference [7]).

2) *Middleware*

A number of RTOSes are designed to support applications with functional safety requirements. For example, a number of RTOSes for ARM Cortex-M processors have been certified for safety-critical applications (e.g. ThreadX, FreeRTOS-MPU and a number of other RTOSes have been certified by SGS-TÜV).

VII. Conclusions

The design of high reliability SoC products requires many areas to be specially handled. For example, on the hardware side the memory as well as the logic might need special arrangements like memory ECC and dual-core lock-step feature. On the software side there is also a need to protect the system against common software failures and handling of various error cases. Fortunately, a number of features on the Cortex-M processors are available to help designers to create SoC products that can reduce risk of failure, or to enable errors to be detected quickly and to be handled such as by fault exception handlers.

There are also various other IP solutions available to enable better reliability at system level, as well as software product solution for enable high quality software development for safety critical systems.

References

[1] ARM Embedded SRAM and ROM IP (http://www.arm.com/products/physical-ip/embedded-memory-ip/sram.php, http://www.arm.com/products/physical-ip/embedded-memory-ip/rom.php)

[2] Cypress SONOS Technology (http://www.cypress.com/?docID=45736)

[3] An Embedded 90nm SONOS Nonvolatile Memory Utilizing Hot Electron Programming and Uniform Tunnel Erase (https://www.freescale.com/files/technology_manufacturing/doc/IEDM_2002_CRAIG_SWIFT_SONOS.pdf)

[4] ARM Cortex-M7 Processor (http://www.arm.com/products/processors/cortex-m/cortex-m7-processor.php)

[5] Enabling Increased Safety with Fault Robustness in Microcontroller Applications (http://www.arm.com/files/pdf/Enabling_Increased_Safety_with_Fault_Robustness_in_MCU_Applications.pdf)

[6] Yogitech fRCPU (http://www.yogitech.com/en/frstl-0) Generic versions for Cortex-M processors: http://www.yogitech.com/en/frstl-arm , STM32 specific version: http://www.yogitech.com/en/frstlstm32

[7] ARM Compiler Verification (www.arm.com/products/tools/software-tools/mdk-arm/compilation-tools/compiler-verification.php)

[8] IAR Embedded Workbench certified for function safety development (http://www.iar.com/About/Pressroom/Press-releases/2013/4/IAR-Embedded-Workbench-certified-for-functional-safety-development/)

[9] ARM Compiler Qualification Kit (http://ds.arm.com/ds-5/build/arm-compiler-qualification-kit/)