Fault-tolerant satellite computing with modern semiconductors

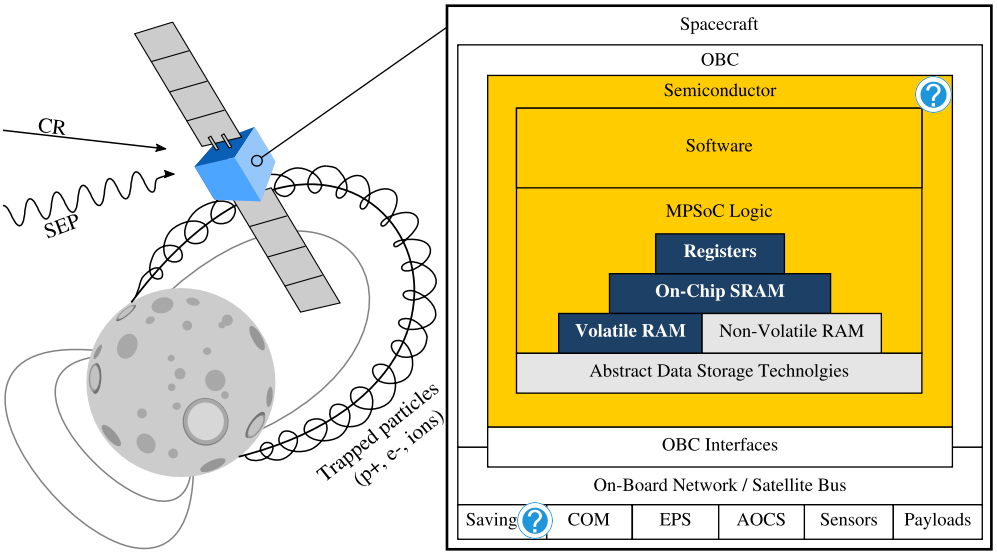
Fuchs, C.M.

Chapter 5

MPSoC Management and Reconfiguration

Stage 2

In this chapter, we describe the functionality of the second fault-mitigation stage of our architecture and address RQ2. Stage 2's functionality was originally developed as a saving-subsystem for the MOVE-II CubeSat and was meant to perform autonomous chip-level debugging only. Within the system architecture described in this thesis, it now fulfills the role of the FPGA's supervisor*, but the concept itself predates the architecture described in the previous chapter. In the context of this thesis, remote debugging is one among several tasks this component performs: it controls the coarse-grain lockstep of Stage 1, conducts FPGA configuration management, and handles thread-allocation within the system for Stage 3. It safeguards the integrity of the FPGA-fabric, may repair defective processor cores through partial reconfiguration, and can offload tasks to the configuration controller implemented within the FPGA. Thereby, it can increase the long-term fault coverage of the system as a whole.*



5.1 Introduction

Nano- and microsatellites have evolved from purely educational projects to fit a diverse range of commercial and scientific use-cases. This class of satellites can do so by combining rapid development, reduced design complexity, low manpower requirements, and minimal cost through a reliance on commercial off-the-shelf components (COTS). Modern embedded technology enables a high level of compute performance at the cost of little energy. Miniaturized satellite development has begun to rely upon conventional application processor architectures as well as FPGAs. Hence these satellites can nowadays offer an abundance of storage capacity and compute performance [220].

CubeSats have proven to be both versatile and efficient for various use cases. They have also become platforms for an increasing variety of scientific payloads and commercial applications [32]. However, such missions require an increased level of dependability in all subsystems compared to educational vessels, especially to enable their use within critical missions and for such with prolonged lifetime requirements. Currently, miniaturized satellites are plagued by low dependability, and will be requiring failure tolerance and reliability enhancing measures in the future. Due to the limited budget, mass and volume restrictions within miniaturized satellite projects, such measures usually must be achieved using means beyond replication and redundancy.

Data storage and processing applications can be protected using architectural and software side approaches, combining them into hybrid solutions. However, even utilizing such hybrid concepts, component level failure tolerance remains limited using only COTS hardware. Acceptance of eventual failure of an on-board computer (OBC) due to issues beyond the control of the deployed flight software without a viable recovery strategy in place is a tolerable approach for educational satellites. However, especially when deployed in larger quantities (e.g., constellations), failure diagnostics and recovery measures that do not require the active cooperation of an OBC or its operating system should be available.

In contrast to larger vessels, the use of chip-level debug functionality aboard miniaturized satellites has up until now largely been restricted to the development and testing phases. During system development and testing on the ground, low-level debug interfaces are usually used for diagnostics, debugging and failure analysis, providing chip-level access to satellite hardware. However, such functionality often lays dormant once the satellite has been deployed or is not even activated in a satellite OBC's flight model. Thus, debugging functionality has rarely been utilized in-orbit aboard CubeSats, as the necessary protocols could not be implemented over the unreliable low-bandwidth links without major effort.

Few nanosatellite projects possess the manpower and time to implement sophisticated failover functionality and testing effort until a very late phase during development when facing non-trivial bugs. Many CubeSat developers also are unaware of the challenges of hardware development, and therefore ignore low-level debug functionality in satellite design altogether. In contrast to debugging capabilities, flight software reprogramming functionality is usually desired aboard nanosatellites. Hence, several CubeSats were equipped with simple proprietary update solutions [221–223]. Even though the capabilities of these concepts were limited with little re-use potential, they underlined the importance of software-independent chip-level debug functionality such as JTAG [6].

Hence, began exploring how a miniaturized satellite's saving subsystem could be

outfitted with chip-level debugging capabilities in late 2014, and developed a concise concept in early 2015 and implemented the prototype described in this chapter in late 2015. We designed this subsystem to enable extensive debugging and analysis support for the MOVE-II CubeSat [Fuchs13], as prior experiences in the field and especially in the FirstMOVE predecessor CubeSat showed that this functionality is critical [Fuchs17]. It is designed to support testing, verification, and debugging on the ground as well during a space mission. It offers scripting support through the use of STAPL [224] bytecode which is then translated into JTAG operations using a STAPL virtual machine, thereby offering near universal test-target support. Hence, the subsystem's software can remain static at run-time and does not need to be changed throughout a space mission. The multi-stage fault tolerance architecture described in Chapter 4 is a direct evolution of the concept described in this chapter. In the remainder of this thesis, this saving subsystem also takes on the role of the MPSoC's supervisor, integrating most of the usage concepts described in Section 5.4.

In the next section, we will analyze how and why debugging at chip level can help improve dependability. We outline why this functionality up until now is largely unavailable aboard miniaturized satellites, and what functionality is required to implement such a saving subsystem. Section 5.3 then contains a description of our work and offers insight into several key aspects of the developed concept. Afterwards, use cases beyond mid-mission debugging are presented in Section 5.4. We discuss plans for future work and present our conclusions in the final two sections.

5.2 Debugging and Reliability

Testing and error diagnostics are critical tasks during hardware development, and thus also when developing nanosatellites. While larger spacecrafts' OBCs have extensive debugging support, CubeSats usually offer no equivalent functionality and, if at all, resort to creative ad-hoc testing solutions. Most such solutions can not deliver equivalent functionality to the comprehensive set of testing and debugging features often encountered within COTS hardware or aboard larger spacecrafts. Besides functionality, the reliability and universal usability of these solutions is often insufficient, resulting in few CubeSats fielding any form of software-independent mid-mission capable fault analysis functionality. In consequence, few CubeSats nowadays offer sufficient fault detection, isolation and recovery functionality (FDIR) to reliably detect and recover from hard- or software malfunctions.

Most system-on-chip architectures, FPGAs, and many other ICs provide JTAG test access ports (TAPs) [6]. Originally developed for circuit testing, JTAG nowadays is the de-facto standard chip-level debugging interface and is widely used in electronics for larger satellites. Hence, JTAG is an ideal interface for sophisticated fault detection, isolation and recovery in case of component failure. In addition, it can be utilized to update an OBC's software, firmware, as well as to control and reconfigure the programmable logic of an FPGA. We argue that chip-level debugging is currently not widely used because there are no readily available CubeSat-compatible solutions that can be adapted to a wide variety of different designs.

The properties of the communication bands utilized for commandeering aboard contemporary CubeSats (usually UHF and VHF, see Chapter 3), the constrained up- and downlink availability, and the low bandwidth make mid-mission debugging challenging. As discussed in Chapter 3, these restrictions result in constrained data rates

around tens of kbps, even if strong error correction is utilized. As ground station networks and satellite relay systems at the time of writing are not accessible to ordinary nanosatellites, debugging and error diagnostics must be conducted fully remotely. JTAG requires bi-directional real-time communication and is sensitive to timing issues, aspects which are not suitable for satellite links in general and especially the links available aboard miniaturized satellites. Hence, the chip-level debugging must be decoupled from the satellite link, so that live-interaction during debug sessions only happens locally within the spacecraft.

STAPL scripts can be executed autonomously and perform all timing-critical operations locally within the space segment. Thereby, we can terminate the timing-critical aspects of chip-level debugging while minimizing link congestion. The saving subsystem described in this chapter can, thus, efficiently operate even via a lossy, unreliable very-low-bandwidth communication channel. It can operate even in environments with elevated radiation levels, requires little PCB space, low power and entails minimal cost.

5.3 Implementation Details

The main objective of the research described in this chapter is to improve overall reliability and survivability of a spacecraft. Hardware complexity has been a major issue in CubeSat projects, often resulting in oversimplified systems due to lack of experience and sometimes even in overly complex systems due to uncontrolled feature creep. Due to the absence of sophisticated FDIR functionality, even minor hardware and software may cause a CubeSat to become unrecoverable.

In the remainder of this section, we will discuss the MOVE-II CubeSat specific implementing of our saving subsystem using an Microchip/Atmel SAM7SE MCU. However, it should be noted that besides the hardware choices outlined in this chapter, there are numerous other MCUs which could be utilize instead. Originally, the this saving subsystem was intended to integrate into an existing Spartan 6 LX45 FPGA on MOVE-II's transceiver module. However, due to the densely populated transceiver board and insufficient FPGA resources on the LX45, a microcontroller (MCU) based implementation was developed instead.

In the context of this thesis, we instead chose to utilize a radiation-robust TI MSP430FR MCU, as we describe further in Chapters 9 and 10. A SAM7SE offers considerably more performance than an MSP430FR MCU. However, the tasks this saving subsystem is meant to perform within the architecture described in Chapter 4 require little performance, and MSP430FR MCUs have been shown to perform exceptionally well under radiation [225].

5.3.1 Hardware Requirements

The saving subsystem can be implemented with comparably basic hardware, however, we must also consider assuring integrity of the subsystem itself. MRAM [150] and phase-change memory (PCM) [226] both are ideal technologies for holding saving subsystem's code and stack segments, as their storage cells are radiation immune. At the time of this writing, no affordable highly-reliable nanosatellite-compatible hardware that could be used to implement the presented saving subsystem is available. Thus, we have to resort to utilizing COTS MCUs and minimize fault potential. This MCU must provide the following functionality:

- an external memory interface to attach a parallel magnetoresistive RAM (MRAM [150]) to contain the saving subsystem's code, or an MCU with internal MRAM. However, we are unaware of the existence of COTS MCUs equipped with sufficient MRAM.

- A second memory interface will be needed to access flash memory to store larger chunks of data such as FPGA configurations operating system updates. Once PCM or STT-MRAM with larger capacities [227] becomes widely available, the saving subsystem could also be implemented using just one large memory IC.

- The saving subsystem does not require a real-time clock, as we intended the saving subsystem to be as static and stateless as possible. However, we still must assure precise timing for certain operations requiring at least a counter/timer.

- We also must be able to interface with at least one JTAG chain which we can best achieve using a set of general-purpose I/O pins. The capability to access additional JTAG chains enables more advanced usage scenarios.

The program code of the saving subsystem resides in a write-protected MRAM region, whereas the stack segment will be kept within a separate writable region. Thus, faults in the running system's state can be resolved through a reboot in many cases. In consequence, it can then resolve or remove leftover information from the (corrupted) previous system state and thereby recover to a consistent system state. The saving subsystem's (runtime-static) firmware, in turn, can be protected from corruption through erasure coding as described in Chapter 7. Redundancies for MCU and memories can be added as necessary, and are omitted from this chapter for the sake of briefness.

5.3.2 STAPL Scripts and Commandeering Interface

The subsystem offers extensive scripting support through the use of the STAPL scripting language, which is then translated into JTAG operations using a STAPL virtual machine [6, 224]. Hence, the saving subsystem's program code can remain static at run-time requiring no modification to the virtual machine's code. As the STAPL scripting language is Turing-complete[1], it can be utilized to implement arbitrary sequences of JTAG operations in the form of STAPL scripts, achieving code separation and time triggered execution. By using STAPL scripts, we can thus avoid timing critical aspects of chip-level debugging aboard the satellite while minimizing link congestion. Thereby, the saving subsystem can be efficiently operated even over a unreliable very-low-bandwidth communication channel, which would otherwise make chip-level debugging infeasible.

We chose to utilize the STAPL bytecode format [224] to minimize script- and code-size while retaining flexibility. These scripts as well as all relevant program code and state information must reside within radiation tolerant MRAM. Even though STAPL bytecode is more compact than the text based equivalent, experiments have shown that more complex scripts can still become as large as 50kB.

Due to the limited memory capacity in MRAM, only few scripts can be uploaded to and stored permanently within the STAPL machine. For the sake of simplicity, we

[1] in our context it most importantly supports recursion and jumps

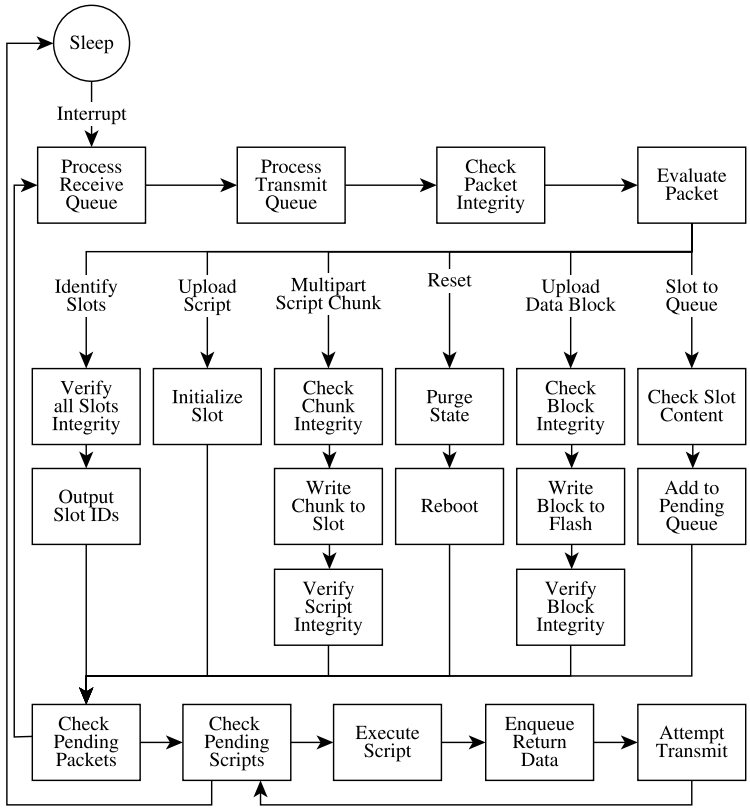


Figure 33: A visualization of the saving subsystem's program flow and commandeering protocol we developed around the Altera JAM player.

utilize a compile-time space distribution, creating a fixed number of identically sized script slots. Each slot can only hold one script, even if the script does not utilize entire entire capacity of a slot. The original implementation of this saving subsystem utilized 2MB of MRAM, and we implemented 10 x 50kB sized slots leaving 1.5MB of MRAM for the stack and code segments.

In the current implementation, slot allocation is managed at the ground segment by the satellite operator and we currently support only equally sized slots. A potential future optimization would be to utilize differently sized slots (e.g., 5×10kB slots, 5×50kB slots, 2×100kB slots), to achieve better resource utilization. We implemented static slot management to minimize code-complexity and failure potential.

Slots are identified by a CRC16 checksum used as reference for commandeering, and also for integrity checking of an individual script. This checksum is uploaded with each new script, and verified once the transfer of all script-parts has been concluded.

An additional identifier beyond this checksum is unnecessary. The low number of scripts minimizes the chance of checksum-collisions due to the birthday paradox [228], Operators can avoid collisions altogether through padding scripts on the ground.

Scripts are directly committed to a slot and then checked for integrity to minimize data duplication and resource usage. Hence, we can assure that only uniquely identified, correctly and completely uploaded scripts will be executed.

5.3.3 Transfer of Large Scripts and Data Housekeeping

The maximum frame size supported by the communication modules of most nanosatellites is considerably smaller than the script size, hence the saving subsystem supports multipart transfers for scripts and other data. A multipart script transfer initialization packet contains the intended slot ID to be overwritten, the expected script checksum and size, as well as the chunk size. The initialization packet also provides a null terminated array of checksums for each to be expected chunk.

For each active multipart transfer, the saving subsystem retains a list of missing frames. It notifies the ground station in case the final missing chunk has been received, or upon command. For slots, this information is stored within the slot header. Later packets indicate the chunk-offset, to facilitate simple retransmission.

FPGA configuration variants and software updates for the OBC can be as large as several megabytes. Hence, they must be stored in dedicated heap memory and multipart transfers of such data is conducted akin to multi-part scripts. We decided to perform allocation and data management on the ground, instead of implementing dynamic heap memory management. Again, this implementation decision was made to minimize software complexity and failure potential. As all operations executed by the saving subsystem must be pre-planned by the operator, more advanced allocation mechanisms do not result in operational advantages.

We utilize flash memory to store larger data volumes outside of the script-slots as neither PCM nor larger MRAM chips are currently widely available. As this data is not executed, we can utilize flash memory and store the data using erasure coding in software. However, in STAPL scripts all payload-data is usually encoded inline and cannot be omitted without modifications to the scripting language syntax.

For this purpose, we extended the STAPL syntax to also support references to external data. We replace inline data with a reference to data in flash, which can then be uploaded independently. Therefore, the STAPL Bytecode player was modified to make it capable of side-loading auxiliary data.

The results of scripts, e.g., kernel dumps, system state information and other diagnostics data, are thus also held in flash memory until they can be transmitted to the ground station. Script execution can be triggered in bulk, hence outgoing packets are being stored in a FIFO queue for transmission. A more detailed representation of the saving subsystem's program flow is provided in Figure 33.

To safeguard against data corruption due to space radiation effects (single- and multi-event upsets), coarse symbol level Reed-Solomon erasure coding [229] will be applied when writing to flash memory [230]. As flash memory with comparably low density is utilized, no additional layers of erasure coding are necessary but could be implemented, see Chapter 7. Reasons for utilizing higher-density flash memory may be the requirement for storing more partial reconfiguration partition variants to cover the increased number of permanent faults that can be expected in space missions with longer duration, or to provide feature-diversity as described in Section 5.4.3.

5.3.4 Integration into an On-Board Computer

Our current saving subsystem implementation consists of an ARM7TDMI MCU with an OBC-independent communication channel toward the CubeSats transceiver or saving subsystem as depicted in Figure 34. We chose to utilize an interrupt-driven bidirectional SPI-based interface to implement this channel due to its flexibility and

simplicity. Also, this interface is less prone to implementation issues than I^2C, however there are many other alternatives and the saving subsystem's concept does not foresee a specific interface. The saving subsystem is attached to a single four pinned JTAG chain, containing all to be debugged JTAG enabled devices. Due to abundantly available GPIO pins, additional JTAG chains could be attached with ease once the software has been adapted.

The Microchip/Atmel SAM7SE MCU is able to boot from memory attached to its external interface, has excellent toolchain support, documentation and minimal energy consumption. Attached to the external memory interface are an Everspin 2MB MRAM memory chip as well as 16MB of NAND Flash. The MRAM chip is connected to the 16-bit memory interface and used to store the program code, scripts, and also serves as main memory. The use of the SAM7SE's internal memories is avoided whenever possible since radiation hardness cannot be achieved here. Only the MRAM address ranges used as main memory and for STAPL scripts and the stack segment are writable by software, all the rest of the memory is set read-only through the ARM7TDMI's MPU.

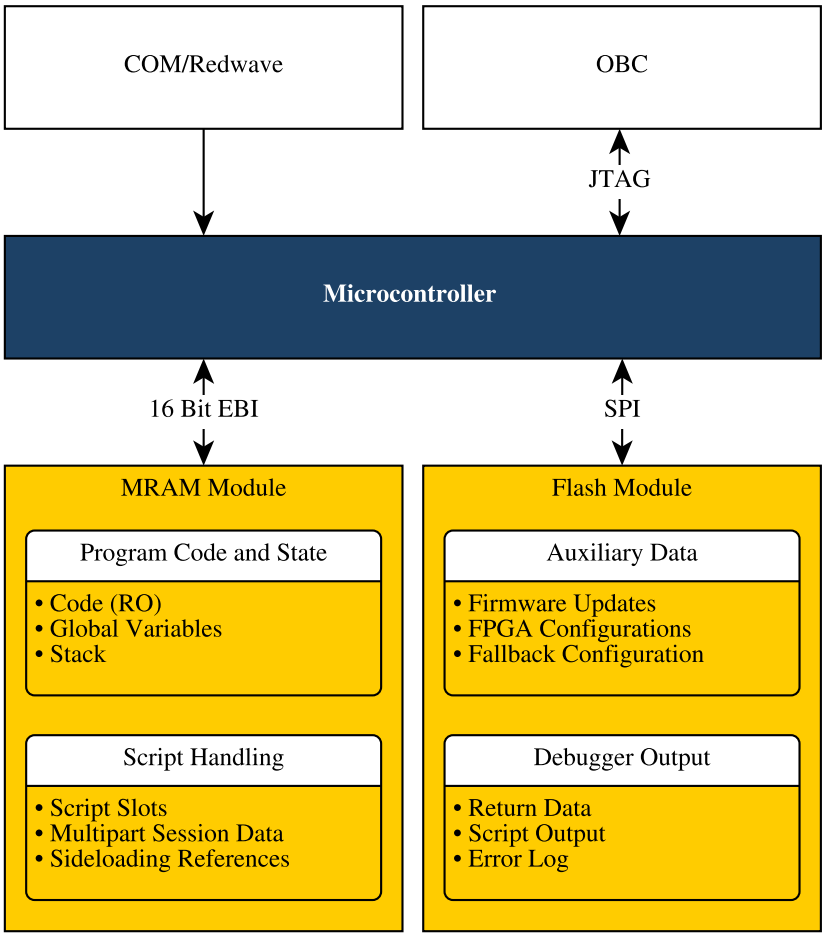


Figure 34: A component-level view of the saving subsystem.

Since the MCU reads its output data from different external memories at different clock rates (MRAM and flash), JTAG clock frequency is dynamically adapted. However, the JTAG clock speed is capped by the maximum support frequency of the debugging target. Dynamic clocking and the use of lower JTAG frequencies are common. Therefore, the duration of one clock cycle is variable, resulting in drastically varying clock speeds especially if access to flash memory is necessary.

Figure 35 shows the hardware setup used to port the saving subsystem from the original proof-of-concept implementation to embedded hardware. It includes a SAM7SE512 MCU and two external memories:

- 16MB SDRAM to simulate the MRAM, and
- 256MB flash memory.

All components and interfaces besides the SDRAM correspond to the originally intended design of the saving subsystem. The commandeering interface has been successfully tested with several self-contained scripts as well as such referencing external data. The saving subsystem currently implements the commandeering API depicted in Figure 33 directly. In a future version of this implementation, we plan to replace the SAM7SE512 MCU with a radiation-robust MSP430FR microcontroller, to reduce failure potential, and as this saving subsystem has very low performance requirements.

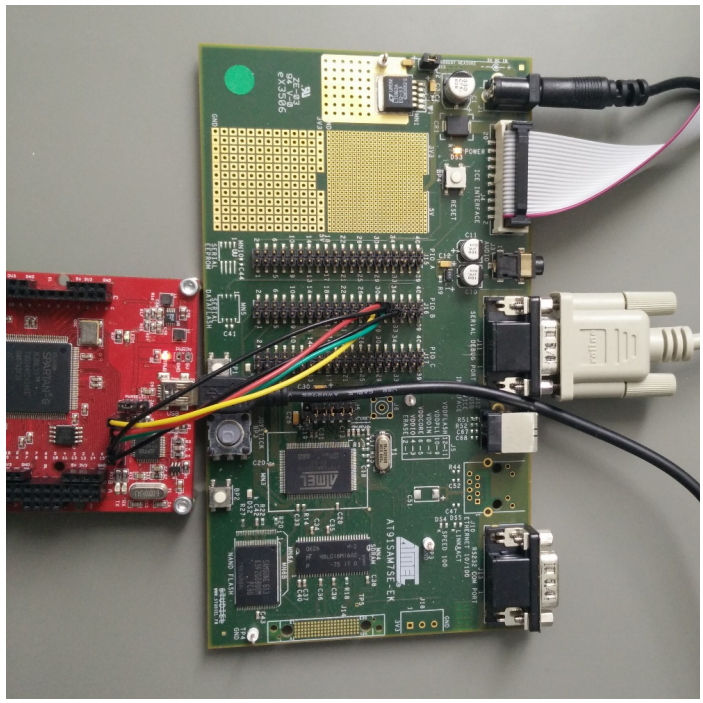


Figure 35: A saving subsystem demonstration setup utilizing the SAM7SE (green PCB) and external NAND-flash and external SDRAM. In this picture, the system was interfaced with a Xilinx Spartan 6 FPGA (red PCB to the left) and validated the FPGA configuration. Due to the concepts simplicity and flexibility, the saving subsystem can be implemented in full just a micro-controller development board.

5.4 Use Cases beyond Debugging

While the presented subsystem was developed primarily for FDIR reasons, there are several additional use-cases that were considered during design. The saving subsystem could be extended with additional functionality or may even be used outside of its originally intended usage scenario aboard a spacecraft. Hence, we dedicate this section to discuss other use cases for this saving subsystem beyond traditional LEO CubeSat applications.

The main limitation of the saving subsystem within a CubeSat application scenario is storage capacity and buffer size to return data via a satellite link. However, these limitations mainly affect the following capabilities:

- size and number of slots available within the saving subsystem,
- storage space for referenced data such as FPGA configurations and
- to-be-returned information and logs, and finally the
- total size of FPGA configurations.

For ground applications and even aboard vessels only slightly larger than 1U CubeSats, these restrictions can easily be lifted.

5.4.1 Watchdog Integration

The saving subsystem can be interfaced with a watchdog to achieve extended functionality. This watchdog could notify the saving subsystem about malfunctions within other components of the OBC. The saving subsystem could then begin recovery measures, enabling considerably better fault-recovery and logging possibilities than the usual reset triggered by CubeSat watchdogs. Instead of directly rebooting the OBC into a (presumably) safe mode, the saving subsystem can first collect relevant log information (i.e. retrieve register contents and a stack-trace). Once this information has been stored, it can then be directly reported to the ground station. Also, this functionality could be adapted, e.g., to take into account known permanent faults that may have occurred in a previous mission phase.

We have not yet implemented this functionality, as the described logic first would have to be written as STAPL script and is highly hardware and software dependent. To avoid the saving subsystem's return-buffer from being flooded with crash-logs in case of frequent or repeated crashes, additional logic must be implemented. A simple mitigation method would be a message queue implemented as a ring buffer. Then only a fixed number of diagnostics messages would be retained at any given time, assuring that only the most recent logs are retained and transmitted to the ground.

As watchdog functionality is usually rather simple, it could also be provided by the saving subsystem itself. Integrated watchdog functionality would only require minimal additional code and could be combined more efficiently with the script-driven state machine. However, such functionality is usually considered critical and malfunctions of the watchdog code within the saving subsystem could cripple the rest of the OBC. Hence, watchdog functionality should only be integrated if a suitable interface setup can be achieved, as described see Chapter 10).

5.4.2 FPGA Management and Validation

Radiation tests of several COTS FPGAs such as Xilinx' Spartan 6 FPGA family have yielded promising results. Recent radiation testing activities have shown the Spartan 6 devices to be outright latch-up immune and largely unaffected by event effects [231–233]. While these devices are not truly radiation-hard, they can offer a sufficient level of reliability to be used aboard spacecraft if scrubbing and other software-driven dependability measures are applied.

In contrast to using a discrete processor or a classical SOC design, an FPGA based Soft-SOC could provide drastically improved OBC flexibility. As an FPGA can be programmed with largely arbitrary logic, a broad variety of interfaces and processor architectures can be utilized. Such interface logic thus no longer has to be implemented in hardware using separate controller ICs, and can directly be attached to the FPGA. System parameters can be modified and interface assignment can be changed even mid-mission. Also, permanent faults in reconfigurable logic could be worked around by deploying a similar configuration avoid the use of a certain FPGA area. Hence, one of the main drivers for the saving subsystem's design was to control an FPGA and take full advantage of programmable logic devices.

As depicted in Figure 36, the saving subsystem can not only control and reprogram an FPGA, it can also be used to implement more advanced usage scenarios: A continuous read-verify-repair cycle could be scripted and executed in a timed manner to enable scrubbing and reduce the impact of transient errors [234]. As most radiation effects within FPGAs are transients, thus temporary errors, their impact on the system can be reduced even if radiation-soft SRAM FPGAs were used.

While access to the running configuration of an FPGA is comparably well documented, access to attached configuration memory requires slightly more effort:

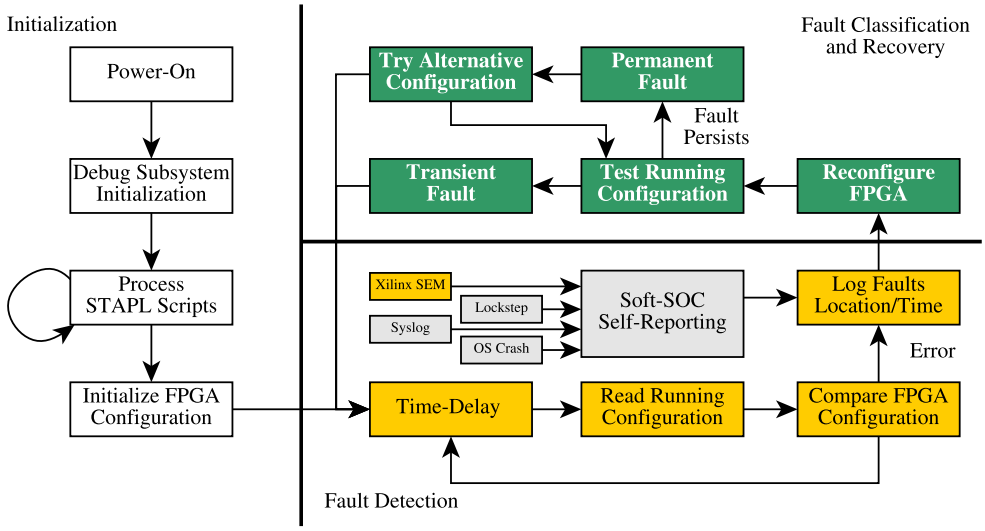


Figure 36: The saving subsystem can also be adapted for radiation testing and FPGA integrity assurance in space. In this case, the saving subsystem can implement all functionality necessary for MPSoC supervision as described in Chapters 4 and 6.

- In case the running configuration is still functional, the saving subsystem can access such memory via the system bus through a separate JTAG bridge.our Such bridges are standard IP-cores and readily available for many platforms (e.g., AMBA/AHB, AXI, ...) and often are even foreseen in the platform specification for system debugging (i.e. GRLIB). In Chapter 10 we realize this functionality through an SPI2AXI bridge.
- For simple interfaces such as SPI, a multi-master setup with both the FPGA and the saving subsystem driving configuration memory can be realized. Again, we utilize such a setup in Chapter 10.
- Otherwise, a separate FPGA configuration must be uploaded to function as a JTAG bridge.

On some FPGA platforms, the second approach is being performed using nested configurations (nested bit-files). An FPGA configuration implementing a JTAG to SPI interface is used to transfer the actual configuration bit file into the configuration memory. Even though this interface requires minimal logic and usually covers only few slices on an FPGA, the total size of an FPGA configuration is still determined by the size of the FPGA. Compression can be used to reduce this dead-space, thus the JAM player foresees ACA [224] compression. However, the saving subsystem then still has to store multiple bit-files.

5.4.3 Flexible OBC Provisioning for Advanced Missions

The saving subsystem can also reconfigure an OBC with several different FPGA configurations for reasons beyond FDIR. More complex space missions consist of several different phases with varying duration and requirements towards the OBC as depicted in color in Figure 37. Using traditional discrete processing components or write-once anti-fuse FPGAs, the properties of a system are static and can not be modified later on. An $n+1$-voting circuit can deliver a fixed amount of compute performance and a certain level of dependability. Thus, if the OBC must be able to handle an increased compute burden or provide stronger integrity assurance guarantees for a certain mission phase, the system design as a whole has to be adapted.

To fulfill varying requirements, systems engineers usually resort to over-provisioning to assure system performance and failover capabilities. Thus, if additional compute performance was required for a voted SOC setup, system properties such as clock frequency and the number of processing cores being part of the voter could be increased. If this is insufficient, then a second, identical setup would have to be added to allow the system to scale with these requirements. Of course, the resulting system's efficient will thereby be reduced.

Additional compute resources or redundancy thus remain unused throughout most of a mission, increasing overall power consumption and system complexity. Dynamic FPGA configuration management based on mission phase requirements could drastically improve overall performance and reliability of an OBC design. As shown in Figure 37, the saving subsystem could provision different SOC variants with a varying number of processing cores and TMR strength depending. Provisioning could be conducted automatically based on the requirements of different mission phases. Thereby, instead of over-provisioning, an OBC design could be adapted to deliver a near-optimal level of performance, reliability, latency and power saving for each mission phase.

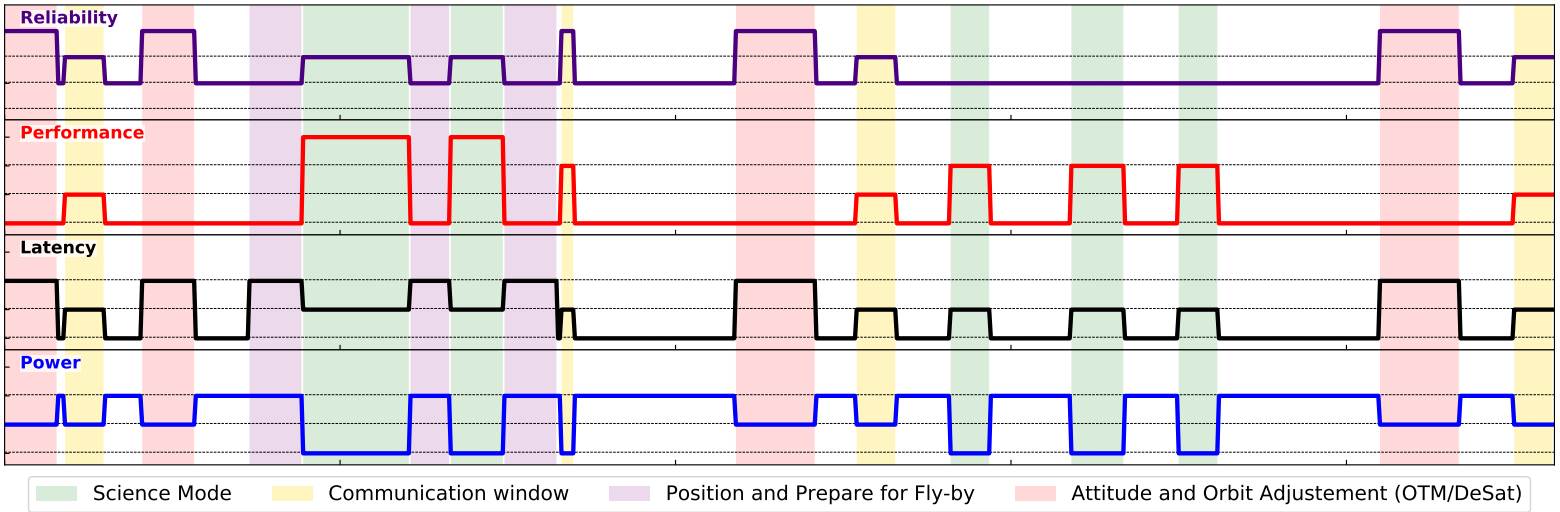


Figure 37: The different operational phases of an exemplary solar system exploration mission modeled after NASA's Enceladus Life Finder (ELF) mission. Each phase implies different requirements towards an OBC's reliability and robustness to faults (purple line), raw compute performance (red), power saving capabilities (blue) and latency to achieve a desired level real-time capability (black). The dashed lines in each plot indicate qualitative "high", "normal", and "reduced" levels of each properties that must be satisfied delivered by an OBC. E.g. in science mode, it will be beneficial to maximize system reliability and performance, and in turn accept an increased level of energy consumption for a this period of time. In communication or maneuvering modes, raw compute performance may be secondary to increased reliability (communications), or better real-time capabilities (maneuvering and orbit adjustment). Operational phases corresponding to those of Figure 20.

As an example, a regular TMRed system consisting of three active cores and one spare could be slit into two independent DMRed SOC pairs using a different SoftSoC configuration. As shown in the figure as well, during some phases of the mission, not all interfaces to other subsystems of the spacecraft are necessary. A separate FPGA configuration could be deployed which does not drive these interfaces to help conserve energy. Hence, the same chip on an unaltered OBC board could fulfill its role in a considerably more efficient way, resulting in efficiency improvements in all regards.

5.4.4 Radiation Testing and Profiling

There are also use cases for this concept on the ground, e.g., to substitute for equipment usually used for radiation testing and profiling of programmable logic or processor designs. To improve the quality of results on a device's behavior undergoing radiation testing, the subject device or FPGA should be continuously probed to log the type of radiation-induced errors when they occur. A post-mortem analysis hereby would only reduce the quality of information obtained and may even mask errors.

As outlined in Section 5.4.2, the saving subsystem can maintain a configuration scrubbing and reprogramming cycle. While the necessary hardware to do so has been developed in the past already, the saving subsystem allows improved flexibility while reducing the need for support equipment. To do so, the saving subsystem must be implemented using radiation hardened components, and the simple design and low performance requirements allow the use of primitive electrical components.

Instead of counteracting the effects of radiation events, the saving subsystem can log upsets within the running configuration of the subject device. Later on, this information can be forwarded to perform forensic analysis and look up which region of the configuration was affected and in what way. If combined with watchdog functionality as outlined in Section 5.4.1, the setup can also help assess the severity and impact of event upsets and can help to map critical logic. The saving subsystem can automatically determine information about which of the most recent upsets could trigger system failure within, e.g., Soft-SOC configurations. Of course, the saving subsystem can also make use of more advanced integrity control functionality and can therefore improve logging. It can directly utilize other information sources such as crash logs, information about software-handled errors, and faults detected by specialized IP (e.g., Xilinx Soft Error Mitigation [235]).

The saving subsystem can also perform scrubbing on an FPGA configuration, which allows further classification into transient and permanent errors, refining testing results. Hence, fault analysis can then be conducted using high-quality information and the results obtained can also be fed-back into the testing cycle, see Figure 36. This information could ultimately also be introduced into an FPGA design's testbench and can help simulate the impact of changes to design based on realistic information without performing additional radiation tests. Analysis suites such as SETA [236] could further help automate this process and may be used to obtain additional information from saving subsystem traces. The saving subsystem can thus drastically improve the quality of radiation testing results when working with FPGAs and can substitute a major part of the otherwise required testing infrastructure.

5.5 Discussions

Development of the saving subsystem currently is in the prototype stage and a successful proof-of-concept has been implemented. Therefore the next step is to integrate it with other components of a CubeSat on-board Computer. The protocol to interface with the communication module via SPI has to be implemented and tested thoroughly. Once the API has been adapted to this protocol, a custom hardware prototype with the respective memories can be implemented.

Also, the saving subsystem is currently based upon a set of development boards meant for rapid prototyping. It therefore must be condensed to a CubeSat compatible form factor. Testing in this case also requires a broad variety of STAPL scripts to be developed to assure code coverage during testing. These additional scripts will then also be utilized to support development of other subsystems and testing of the attached OBC. Performance measurements, including power consumption under load, execution speed of different debugging operations must be performed as well.

There are also several extensions to the current saving subsystem implementation that should be added, such as support for multiple JTAG chains. The current implementation relies on using only one JTAG chain for all devices connected to the debugger, subjecting it to the risk of failure. In case one of the JTAG chain members malfunctions and can not transport the test data signal, the chain is rendered useless and debug operations can not be performed. Support for more than one JTAG chain would allow access to, e.g., a SoftSOC to be implemented in parallel to controlling the FPGA itself. The to-be-executed script could then also select the correct JTAG chain, requiring only minimal modifications to the STAPL logic. This also opens up additional usage scenarios especially when combined with FPGA/SOC hybrids such as Xilinx's Zynq family and the more powerful FPGAs utilized to realize the proof-of-concept MPSoC described in Chapters 9 and 10.

5.6 Conclusions

In this chapter we presented a subsystem enabling autonomous chip-level debugging for nanosatellite OBCs. Until now, chip-level debug functionality had not been readily available aboard miniaturized satellites. If at all present aboard CubeSats, such functionality had largely been restricted to the development and testing phases. We are convinced that the low survivability of many earlier CubeSats can be attributed, among other causes, to low per system dependability and a lack of FDIR functionality. Hence, we developed this concept to provide a readily usable CubeSat compatible mid-mission FDIR solution for the nanosatellite audience.

We developed two prototype implementations up until now:

1. an initial proof-of-concept based upon a Raspberry-Pi to demonstrate the general feasibility of the saving subsystem and to determine requirements for further development.

2. An embedded implementation for an ARM7TDMI MCU in preparation to migrating the design to CubeSat compatible form factor.

The saving subsystem can be integrated into most CubeSat architectures requiring only a JTAG interface towards to-be-controlled devices. It is based upon a minimal set

of components to retain simplicity, utilizing smart technological choices and erasure coding where necessary to achieve dependability using affordable COTS hardware. The presented design utilizes the STAPL scripting language and therefore can support a wide variety of devices. Due to its flexibility, several other use cases beyond debugging are imaginable, both in space and on the ground. The setup has been implemented successfully and thoroughly tested by controlling several ARM SoCs as well as FPGAs.