Research Article

Development of Embedded Boot Software for a Satellite Instrument Control Unit: Lessons Learned

Jesús Fernández-Conde, Jaime Gómez-Saez-de-Tejada, David Pérez-Lizán, and Rafael Toledo-Moreo

1GSyC Department, ETSIT, Universidad Rey Juan Carlos, 28943 Madrid, Spain
2Space Science and Engineering Laboratory, Univ. Politécnica de Cartagena, 30202 Cartagena, Spain

Correspondence should be addressed to Jesús Fernández-Conde; jesus.fernandez@urjc.es

Received 24 May 2019; Revised 23 November 2019; Accepted 28 November 2019; Published 11 December 2019

1. Introduction

In a satellite spacecraft, an Instrument Control Unit (ICU) is a box in charge of control and monitoring one of the satellite instruments. The ICU will generally communicate with other elements of the instrument, as well as the spacecraft main Control and Management Unit (CDMU), by means of different buses [1] (e.g., Milbus 1553, SpaceWire, high-speed serial lines).

A typical ICU is in charge of receiving telecommands (TCs) from the CDMU and carrying out several housekeeping duties, performing the mechanical and thermal control of the instrument itself. Normally, these tasks are accomplished by means of radiation-hardened CPUs and FPGAs, including the necessary interface controllers.

The embedded boot software (BSW) is a small computer program responsible for ICU HW initialization and self-testing, maintenance of ICU on-board memories (RAM, EEPROM, and PROM), and ICU application software (ASW) launching. ASW [2] is devoted to manage all functionalities related to instrument command and control.

Although the BSW is a small-sized software package, its mission-critical role, hard real-time requirements, memory size constraints, and the inability to be patched in-flight make its development and integration with ICU HW rather challenging tasks.

In this paper, we introduce some major problems occurring during the BSW development and HW integration of a real spacecraft ICU [3], to be utilized in a European Space Agency mission related to the study of the dark matter and dark energy in the Universe [4–9]. Lessons learned from these problems are considered to be of valuable interest for the aerospace and embedded software engineering community.

The remainder of this paper is organized as follows: In Section 2, we discuss in more detail the ICU BSW characteristics. The third section describes the problems originating in
the BSW development and integration stages. The lessons learned from these problems are summarized in Section 4. Finally, Section 5 draws the main conclusions of this work.

2. Materials and Methods

The ICU Central Data Processing Unit (CDPU) is composed of the following HW elements [10]:

(i) System-on-a-Chip MDPA ASIC [11, 12]: containing a radiation-tolerant Leon2FT CPU [13, 14] (SPARC v8 architecture [15]) running at 80 MHz, including two 1553 Milbus [16] interfaces and several Space-Wire interfaces. Main register file is protected by EDAC (Error Detection and Correction).

(ii) Memories:
(a) 64 kBytes of PROM for storing the BSW code
(b) 4 MBytes of in-flight reWritable nonvolatile EEPROM for storing the ASW images and configuration/data to be recovered between resets/power cycles
(c) 8 MBytes of EDAC-protected [17] SRAM

(iii) RTAX2000-S1 FPGA [18]: containing functions mainly related to the internal and instrument interfaces

BSW will be executed in the Leon2FT CPU, being the very first piece of software executed when the ICU is powered up or reset. It is a stand-alone executable which will be stored in a small, reliable, dedicated read-only memory (the CDPU PROM).

It is in charge of initializing and testing the ICU processor module, as well as launching the execution of the ASW after copying it from a nonvolatile memory area (CDPU EEPROM) into the processing SRAM. The BSW is also capable of performing in-flight maintenance of the ASW, by means of memory patch/dump/check operations commanded from CDMU/Ground.

As the BSW does not have a direct code interface with other software items, its static and dynamic architectures are fairly straightforward. Details of both architectures can be found in [19].

Following the applicable software engineering standard ECSS-E-ST-40C [20], BSW has been implemented in C language and assembler. The use of assembler language has been minimized (basically processor initialization and the HW Error Trap handling).

Table 1: Software quality metrics.

<table>
<thead>
<tr>
<th>Clauses</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length of procedure</td>
<td>1</td>
<td>50</td>
<td>Number of executable statements in a function’s body</td>
</tr>
<tr>
<td>Commentary</td>
<td>20%</td>
<td>NA</td>
<td>Number of comment lines per executable statement in a function’s body</td>
</tr>
<tr>
<td>Nesting complexity</td>
<td>0</td>
<td>4</td>
<td>Maximum number of control structure nesting in a function’s body</td>
</tr>
<tr>
<td>Cyclomatic complexity</td>
<td>1</td>
<td>8</td>
<td>McCabe cyclomatic complexity number of a function’s body</td>
</tr>
</tbody>
</table>

The Gcov utility (version 4.6.3), a source code analysis tool that is part of the GNU’s GCC tool suite, has been used to check the compliance of the BSW code with the MISRA-C 2004 rules.

2.1. BSW Memory Constraints.

Codification rules have been adapted from [21]. The process of codification rule checking has been performed by means of an Automatic Review Tool, namely, PC-LINT version 9.0 [22]. This tool has been tailored to check those MISRA-C 2004 rules applicable in the scope of BSW development.

The process of metrics checking has been performed by means of an Automatic Review Tool, namely, RSM version 7.75 [23]. Software metrics imposed to ensure software quality are summarized in Table 1.

In order to ensure that software product BSW has been coded according to design specifications and behaviors according to BSW architectural design, unit testing has been performed on it. Using a “white box” philosophy, for each BSW function developed, one or more unit tests are defined to check the correctness of the function under all possible conditions (different values of input parameters and different simulated HW conditions), obtaining the required statement and condition coverage.

Classified as B-class software, the coverage required for BSW code is 100% (both statement and condition coverage). The Gcov utility (version 4.6.3), a source code analysis tool that is part of the GNU’s GCC tool suite, has been used to check BSW code coverage.

2.1. BSW Memory Constraints. In Figure 1, we can observe the memory allocated to the different parts of the BSW. The BSW TEXT segment, which contains the executable code and constants, has the highest memory size constraint. The BSW has to be implemented with a maximum
size of 64 kBytes, due to the small capacity of the ICU PROM. As a consequence, the BSW will be executed in stand-alone mode (neither real-time operating system nor utility library is present).

2.2. BSW Real-Time Constraints. BSW is comprised of three tasks (understood as major blocks of functionality executed sequentially upon an external trigger):

1. Main Task: triggered upon power-up/reset; it performs system initialization, 1553 core execution start, and the Command Loop, which executes incoming TCs

2. Sync Task: nominally triggered upon a periodic (16666-millisecond period) 1553 Milbus RT interruption (named SYNC interrupt). It performs 1553 messages/error handling, telecommands/telemetry transfer protocols, and cyclic activities with associated real-time deadlines

3. HW Error Trap: triggered upon one of the 12 hardware Error Traps. It updates a simple error management structure in SRAM and returns to the instruction following the failing instruction

The most important real-time constraint is placed in the Sync Task, as it has to finish processing before the next periodic interrupt occurs (a 16666-millisecond deadline). A detailed analysis of WCET for this task can be found in [19]. The Command Loop in the Main Task does not have a specific period. Each step in the cycle will take any amount of time it needs to perform its functionality, and when all steps are finished, a new cycle immediately starts. Each of those steps, however, has a rate limit: TC processing can execute commands at the same rate as they are received from CDMU/Ground (maximum rate is once per second by system design).

2.3. Validation Environment Description. According to the European Cooperation for Space Standardization (ECSS) standard, ECSS-Q-ST-80C [24], BSW is identified as class-B software, meaning that a failure or anomaly in the software execution could cause or contribute to a system failure resulting in critical consequences such as loss of mission. The same SW Quality Assurance standard declares that class-B software shall undergo Independent Software Verification and Validation (ISVV) activities.

In this way, the verification and validation campaign were based on the following guidelines:

(i) Elegant Bread Board (EBB) unit: an ICU-representative hardware unit

(ii) A set of test cases that covered the requirement baseline with different inputs, use cases, nominal and nonnominal interaction, dynamic behaviour, boundary conditions, and error cases

(iii) A set-up composed of the following main tools (see Figure 2):

(a) PC computer with a non-real-time operative system, USB to MIL-STD-1553B card, and UART connection for GRMON debugging and memory analysis

(b) Harness to interconnect: power lines, debug UART, and 1553 buses (2 channels, buses A and B) to the EBB

(c) A real-time FPGA-based module used to identify and synchronize the reset events produced by either software or hardware

(d) GUI tool that controls the set-up including communication with 1553 USB card, power supply, and FPGA module. It performs the communication with BSW by using the 1553 protocol sending and storing TC/TM PUS packets according to the test under execution. The tool is easily managed by the user who can use the GUI to select the test to execute as well as monitor the test status. Additionally, the tool can decide the next TC to send according to the received TM

(e) After finishing any test execution, several logs in a unique folder are stored. The contents of such files are 1553 exchanges, TM/TC packets, internal and test events, memory analysis, etc.

(f) Off-line tool used to evaluate the large amounts of data provided by the previous tool as well as to perform thorough checks in the PUS level

3. Results

In this section, we describe some of the problems arising in the process of ICU BSW development and HW/SW integration phases. Problems are characterized by their symptoms and analysed in further detail, and the solution taken in each case is presented.

3.1. Problem One: BSW Crashes Occasionally on Power-Up

3.1.1. Problem Symptoms. BSW was observed to fail occasionally after a power-off/on cycle, with a rather low probability. The probability of crashing was low (around 1%), but it increased up to 10% after the ICU HW had been working in nominal operation for 2-3 hours. The failure did not occur when a digital reset was done (i.e., no power cycle on the ICU processor). The consequence could be seen in two ways:

(i) A crash on an address position, usually arbitrary but not completely random (i.e., there was a set of usual suspects)

(ii) Sometimes, event telemetry packets were generated by BSW, reporting some failed tests and other trap failures

Common failure occurred upon an unimplemented instruction trap, but other traps were also possible (e.g.,
unaligned memory access and data access exception). The increase in the probability of crashing may be due to the fact that the circuits get “imprinted” for a longer time or because higher temperature slightly modifies the bit effect.

3.1.2. Problem Analysis. Evidences suggest that the failure occurs when an instruction is executed but its corresponding fetch cycle is not performed. The only justification would be a cache hit, even though it is the first time that instruction is executed.

After reset, the ICU processor leaves the cache disabled and its contents uninitialized. The BSW enables the cache but does not flush its contents. Thus, an invalid entry might be incorrectly interpreted as valid and its content used.

Looking at the contents of the instruction cache with the debugger and comparing to the disassembly, it can be observed that the contents of the cache are correct until the instruction that fails with the unimplemented instruction trap is reached. The contents of that cache line look like the actual instructions, but their code is not exactly the same; they are somehow corrupted. The reason is that a previous execution left some content, and during a power-off/on cycle, the “analog memory effect” makes the cache remain with similar content and sometimes the validity bits remain set.

The consequence has been observed, and the link between failing address and bad cache content has been established. An improvement to the BSW is needed, flushing the cache before enabling it at the initialization stage.

3.1.3. Problem Solution. This problem has been solved by modification of BSW. Now, the cache is flushed before being activated. Cache activation occurs both at the initialization stage and after boot ram tests are performed upon power-up. Furthermore, between cache flush and activation, the status of the cache register is polled to ensure that the flush process is over before activation.

3.2. Problem Two: BSW Crashes in Some ICU HW Units

3.2.1. Problem Symptoms. BSW crashed in a deterministic fashion in some ICU breadboard HW units, whereas it worked perfectly in others. Specifically, this problem
occurred in all HW units: initially (although they turned out to be related), as they were other symptoms not associated to this problem initially (and this fact does not affect the execution of BSW, unless some registered invalid value is used before its initialization.

In order to add robustness, a review of the potential registers at risk was performed, finding the following:

(i) **Memory Configuration Registers.** It was found that in some ICU HW units, these registers had the memory bit width initially set to 32, whereas in others, it was set to 16.

(ii) **SPARC Window Registers.** Even if they are not explicitly used by the BSW logic, the overflow trap handler actually reads them before downloading them to stack. Thus, when the first window was flushed, a Register File EDAC trap occurred, inside a trap handler, causing BSW to crash.

(iii) **Force Interrupt Register.** Invalid contents after reset can cause potential spurious interrupts.

(iv) **Floating Point Registers.** The FPU registers (f1-f32) contain uninitialized data (and therefore invalid EDAC codes). Even if these registers are not used by BSW, they may be read during a context switch on the ASW RTOS (BSW does not have such switches), causing a Register File EDAC trap.

### 3.2.3. Problem Solution

This problem has been solved by adding the following modifications to BSW:

1. Memory configuration registers are set to valid values at the very beginning of the BSW execution, before any access to memory may take place.
2. All SPARC input (%i) and local (%l) registers in all windows, and therefore output (%o) registers along the way, are cleared at the very beginning of the BSW execution.
3. Clear all possible interrupt sources in the primary interrupt controller clear register, before any interrupt is enabled.
4. Set to zero all floating point registers during initialization, in order to set their EDAC codes to valid values.

### 3.3. Problem Three: Memory Access Error in 1553 Milbus Communication

The internal MDPA architecture is built around the AMBA [25] bus, with the modules connected either to the AHB (ARM’s Advanced High-Performance Bus) or the APB (ARM’s Advanced Peripheral Bus). The LEON2FT CPU, memory controller (SRAM, PROM, I/O), and 1553 Milbus interfaces are connected to the AHB bus.

#### 3.3.1. Problem Symptoms

Memory access error (“busy”) was reported by 1553 Milbus, and Terminal Flag bit was activated in the 1553 Milbus status word, in the first message after 1553 SYNC mode code interrupt (this interrupt marks the start of a 1553 Milbus communication frame), when time between this interrupt and the first message was below a certain value, approximately 900 microseconds.

#### 3.3.2. Problem Analysis

First, we need to analyse the PROM access time. It is found to be relatively slow, because each access has a large number of associated wait-states (24, as required by the EEPROM, as both PROM and EEPROM are in the same memory bus area, named “PROM area”), and each instruction needs 4 consecutive accesses (PROM is a 1-byte width memory). Therefore, a single 4-byte instruction needs around 104 cycles, i.e., around 1.3 microseconds at 80 MHz clock speed.

Furthermore, the cache line is 32 bytes long, and therefore, if instruction burst accesses are made, the AHB bus is blocked for more than 10 microseconds (a burst access will fill a complete cache line and cannot be interrupted), because burst instruction fetch to load the cache line takes 8 32-bit words in an atomic process for the AHB bus arbiter. Any other core requesting access to the AHB bus has to wait for the instruction burst process to be finished.

The 1553 Milbus core has not been designed to wait for such a long time and therefore provokes a memory access error (“busy”) and a Terminal Flag bit activation in the status word.

#### 3.3.3. Problem Solution

This problem has been solved by BSW modification. The burst instruction cache has been disabled, avoiding blocking of the AHB bus for long periods of time. Now, each instruction is loaded into the instruction cache as an atomic process.

### 3.4. Problem Four: Inaccurate CPU Load Calculation

Estimation of the CPU load is performed in the Sync Task, by calculating the difference between the On-Board Time (OBT) read at two different instants in time:

1. **OBT1:** just after Sync Task starts its execution
2. **OBT2:** just before Sync Task ends its execution

The OBT is provided by ICU HW as two separated counters:

(i) **OBT large counter (32-bit):** 1 LSB = 1 second
(ii) **OBT fine counter (24-bit):** 1 LSB = 1/(2^{24}) seconds

Without loss of generality, we will assume that both OBT readings are performed in the same second, and therefore,
only the OBT fine counter is used in the CPU load calculation. As the Sync Task has a period of 16666 microseconds, the estimated CPU load in microseconds is computed as follows:

\[
\text{CPU load} = \frac{\text{elapsed time}_\mu \text{sec}}{16666}, \tag{1}
\]

where

\[
\text{elapsed time}_\mu \text{sec} = (\text{OBT2} - \text{OBT1}) \times \text{conv factor}_{\text{OBT to microsecs}}. \tag{2}
\]

Being

\[
\text{conv factor}_{\text{OBT to microsecs}} = \frac{100000}{0x1000000}. \tag{3}
\]

Taking into account that BSW does only use fixed-point arithmetic (that is, no floating point numbers) and the fact that the ICU CPU is a 32-bit processor, the elapsed time in microseconds is estimated in this fashion in the BSW code:

\[
\begin{align*}
\text{low byte} &= (((\text{oobt2} - \text{oobt1}) >> 8) \& 0xFF) \times 0x3D09 / 0x40000; \\
\text{mid byte} &= (((\text{oobt2} - \text{oobt1}) >> 16) \& 0xFF) \times 0x3D09 / 0x40000; \\
\text{high byte} &= (((\text{oobt2} - \text{oobt1}) >> 24) \& 0xFF) \times 0x3D09 / 0x40000; \\
\text{elapsed time\_usec} &= (\text{high byte} \ll 16) | (\text{mid byte} \ll 8) | \text{low byte};
\end{align*}
\]

3.4.1. Problem Symptoms. Calculated elapsed times in microseconds had an extraordinary variability. CPU load results obtained during BSW execution did not match the theoretical expected ones.

3.4.2. Problem Analysis. Calculation of elapsed time in microseconds is not accurate. On one side, the distributive operation would advise to use operation “+” instead of “|” when putting the bytes back together because otherwise carries are lost. However, the outcome would be the same because each byte operation guarantees that the result is always lower than one byte because the ratio is lower than one.

Nevertheless, the essential source of inaccuracy comes from the fact that, on each operation, a large part of the OBT information may be lost due to local rounding. For example, a value of 0x01000000 would correspond to approximately 4 milliseconds, but when the high byte is first placed alone, as 0x01, multiplied by 1000000 and divided by 0x01000000, the result of that byte piece is 0x00, so the resulting elapsed time would be zero. A similar rounding occurs on the second and third bytes.

3.4.3. Problem Solution. BSW has been modified in order to prevent this fault. First, the ratio (1000000/0x10000000) can be simplified with a common denominator, so the ratio becomes (0x3D09/0x40000). Now, we can multiply a 16-bit value by 0x3D09 in a 32-bit variable without overflowing and with a very small loss of accuracy. Therefore, the precise calculation is

\[
\begin{align*}
\text{low16} &= (((\text{oobt2} - \text{oobt1}) >> 8) \& 0xFFFF) \times 0x3D09 / 0x40000; \\
\text{high8} &= (((\text{oobt2} - \text{oobt1}) >> 16) \& 0xFF00) \times 0x3D09 / 0x40000; \\
\text{elapsed\_time\_usec} &= (\text{high8} \ll 8) + \text{low16};
\end{align*}
\]

Note that the “+” operation is now strictly needed (it should not be replaced by the OR operation), because both sides have one byte overlapping.

In Figure 3, we can notice the error due to the first (erroneous) calculation, as opposed to the proper estimation corresponding to the modified version of BSW.

4. Discussion of Results

Most errors regarding a HW-related failure are due to either documentation not matching the HW or engineer not following the documentation. This last case is due to assumptions made by the engineer after reading a (probably long) set of manuals. Of course, registers must be initialized and user manual guidelines must be followed, but the problems exposed in this paper show that even the most experienced engineers are not free from overseeing some small details that may cause large problems.

Flight SW systems have many design constraints. Generated flight SW code has to

(i) be small enough to fit in PROM, EEPROM, or RAM
(ii) be efficient to run in relatively low-speed CPUs
(iii) meet real-time requirements to approach determinism and make testing easier
(iv) be robust to avoid having minor issues that ruin the mission
(v) be efficiently testable because the time and budget is limited

It is not always possible to make all those constraints true, but in every possible scenario, the development process should ensure that a code error becomes noticed during Ground testing as soon as possible [26]. The issues here defined were solved because each assumption made during design had an explicit test method during validation [27], including internal logs and reporting that are key to understand SW behaviour when debugging is not an option.

All the software was developed under configuration control, in order to keep track of all changes between different software versions. In addition, for any problem found during BSW software development, an issue was opened using the issue-tracking system integrated in the configuration control tool [28], changes were performed in the BSW bound to that issue until the problem was finally solved, and the issue was closed in accordance.

The key lessons learned from the problems detected and corrected during the ICU BSW development and HW/SW integration phases are summarized below.

(1) First Lesson. It should be checked in the HW processor documentation if the cache needs to be flushed manually before cache activation
(2) Second Lesson. It should be checked in the HW processor documentation if all internal registers contain a valid value after power-up or reset. If this is not the case, the software should initialize all registers to valid values, as early as possible in code execution
(3) Third Lesson. Shared buses blocking times due to burst operations should be checked in order to prevent subsystem timeouts
(4) Fourth Lesson. Rounding errors should be taken into account in calculations that need to be performed in several, byte-wise steps

5. Conclusions

In this paper, we have presented some useful lessons learned during the development of the embedded boot software destined to be executed on-board an Instrument Control Unit of a satellite spacecraft designed for a European Space Agency mission.

Its critical nature, real-time response requirements, and memory constraints do add significant difficulty to the implementation of this peculiar sort of embedded software.

The software development team considers that the lessons learned during the analysis and resolution of the problems occurring surely can help future projects in the aerospace and embedded software engineering community.

Data Availability

No data were used to support this study.

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

Acknowledgments

This work has been supported by the Spanish Ministry of Economy under the project ESP2017-84272-C2-2-R, as well as by ERDF funds from the European Commission.

References

Submit your manuscripts at
www.hindawi.com