OBC design and integration experiences

The story of the birth of the On Board Computer for SSETI Express

Karl Kaas Laursen

2004

Preface

This document is written as a reminder to myself and others of things to do and especially NOT do when designing and integrating a complex system like an on board computer for space applications. The first sections are memoires of more or less significant incidents and general documentation of the work done by my team and me from the beginning of Express until now where the space craft is close to completion. This part of the document is arranged into small, easy-to-digest "snacks" of the Story of The OBC. Later in the document are short problem descriptions leading to "lessons learned" which I regard as "worth taking a quick look at". All people mentioned in this document are REAL living people with thoughts and feelings of their own (well, at least most of them have thoughts) and if you are one of these people, don't feel offended if your name appears in the same sentence as an Italian – you might be alright, anyway.

Karl Kaas Laursen, October 2004, Jason's cleanroom at ESA/ESTEC

Part I

The Story of the OBC

From AAUSAT to SEx

It all started in December 2003 when Lars Alminde, AAU, came back from ESTEC with a proposal for my seventh semester group at Aalborg University: We could have the OBC that we were designing for AAUSAT-II flown on a joint European student satellite called SSETI Express (SEx). The proposal was gladly accepted without further a due and we continued our daily life doing other things as the seventh semester OBC project was already finished and we were caught up in our projects for the next semester.

By the word "finished" I *don't* mean "OBC-in-a-box-alive-and-kicking" but rather "semester-ended-and-we-got-bored-and-wanted-to-go-home". We spent the OBC design semester coming up with an on board communication protocol known as INSANE (Internal Satellite Area Network), and the main processor and operating system for it were chosen leading to a framework for OBC for AAUSAT-II. But no hardware design was accomplished as we regarded that to be somewhat trivial and quietly decided to postpone that until we felt the sour breath of the deadlines on the back of our necks (you feel that just moments before hearing the whooshing sound they make as they go by).

Two representatives from the group (Mike and Danny) were sent to ESTEC for the SSETI Express January workshop that initiated the design of the satellite. Along with them went Lars Alminde and Morten Bisgaard (from here on known as "the Grumpy Old Men" – no offence, of course) who also had subsystems to stick onto Express, namely the ACDS (attitude control and determination) and CAM (camera payload).

At the workshop it became clear that the AAUSAT-II computer would not exactly fit Express in its original form but there were ways around that.

Designing the On Board Computer for SSETI Express

We decided to design a computer that would be much like the one for AAUSAT-II and develop an interface card for it to comply with the interfaces in Express. It had been decided to use a CAN bus for all internal communication in AAUSAT-II because of its robustness and the simple electrical properties that enable compact design. Unfortunately, only one other subsystem on Express would use the CAN interface so the interface card was to be fitted with five RS232 connections for EPS, UHF, SBAND, CAM and ACDS. In addition, the interface card should have analogue inputs for sampling thermistors in TCS, the "thermal control system", and for reading values of sun sensors in the ACDS. Also, digital output lines were needed in order to control the electro-magnetic attitude control actuators, the magnetorquers.

The design of the computer and the interface card was progressing quite slowly. Some of us were more focused on developing attitude control algorithms for AAUSAT-II and testing them on the 7th Student Parabolic Flight Campaign in the summer 2004 and others were doing non-space related projects and did not pay attention to Express at all. As the design workshop in May rapidly approached we made the first draft for the overall architecture of the OBC which I presented at the workshop and people seemed to accept it.

The design of the main board included an Atmel ARM7 main processor (good plan) for the on board data handling software and a smaller (and less powerful) utility processor for interfacing the ARM to CAN bus (very bad idea). The utility processor was chosen instead of a dedicated CAN controller because it would provide us with a seemingly clever way of making software upload to a non-volatile rewritable FLASH memory. The story of the utility processor should have its own chapter... maybe later,

The RS232 connections on the interface was a really annoying feature to implement because it requires a lot of hardware beside the compulsory micro controller. I decided to multiplex the RS232 connections such that we only needed to have one UART (universal asynchronous receiver/transmitter) which any micro controller would have internally. This idea demanded the need for hardware flow control (good idea) on all subsystems communicating via these connections and at some point during the workshop the multiplexer-thing was dropped and replaced by five independent UARTs that all had to work in parallel – without flow control (very bad idea). At the time it looked like a doable solution and quick calculations done in my head convinced me that we could easily accommodate the data flow demands from

the five RS232 connections and the single CAN connection. I should come to regret this.

Interfaces

The design workshop in May was the place for interface discussions and in the perfect world it would have been the *last* place for interface changes. The world is not perfect. The story of the changing interfaces is too long to fit into this document so only the conclusion is included: Define all interfaces and when they have been agreed on don't change them without all parties involved being totally aware of the change. Otherwise, there will be misaligned interfaces that are not discovered until *after* someone's flight hardware is already finished.

The design

The final design of the OBC was carried out in a week or two in July 2004. Danny and I were at the parabolic flight while the rest of the group spent three days sketching a schematic of the main board. We returned from Bordeaux to a half-finished design and discovered quickly that someone had forgot that the utility processor (like any other processor in space) also had to use one-time programmable code memory. The chips we had acquired for utility and interface card processors were the Atmel AT89C51CC03 with internal FLASH memory for code and option for external code memory. That meant that we had to plug an external OTPROM (one-time programmable read-only memory) onto each of the processors as OTPROM is much more resistant to radiation than the FLASH memory. Having three of these 89C51 processors we would, of course, have to have three external PROMs – each with a data latch for address/data bus multiplexing.

Now, it suddenly occurred to us that we had four processors in our design with both FLASH and PROM, so why not design the hardware in such a way that it would be possible to upload new software to any of them at any given time (no no no!)? We added external latches to the processors enabling them to switch between FLASH and PROM code memory by sending it commands when executing from PROM (default code memory). The idea was that at some point we should programme a boot loader that could receive new software from “somewhere”, store it in FLASH and then execute the code. But the “some point” never came – I am still waiting for it.

After a few intensive days of computer design there were pages of schematics but no time to prototype anything – in just ten days we had to go to ESTEC to “integrate” the OBC flight hardware (and software) with all the other subsystems. There was only one sensible thing to do then: Lay out the flight PCBs in a jiffy, so Christian and I sat down for thirty hours and laid out the main board and interface board. Christian did

the main and I did the IF. Thirty hours passed and we had a completely finished hardware design of a main board with an ARM7 CPU, an 89C51, two RAM chips, three PROMs, a FLASH, some latches, an RS232 line driver for debugging interface, a CAN transceiver, two crystals and lots of accessory components like decoupling capacitors, pull-up/-down resistors and power-up reset circuits. And we had a ready design of the IF board with two 89C51 CPUs, two PROMs, three latches, two dual UARTS for RS232, three dual RS232 line drivers, two CAN transceivers, twenty eight protection diodes for analogue and digital IO, two crystals and all the little things plus a switching power converter, coil, capacitors and a diode for converting the 28V regulated bus voltage down to 3.3V that most electronics use; including the on board computer.

Coincidentally, we had got a very nice deal with the PCB manufacturer Elprint A/S who wanted to help us by producing the flight boards free of charge saving us thousands of Euros. The PCB Gerber files were sent to Elprint while we were busy programming all the software needed to operate no less than four processors in our distributed computer architecture. Thousands of line of code were written in a few days without any prototype hardware to test it on (don't do that) – we call this procedure “open-loop programming”.

ESTEC – here we come

Time caught up with us and the PCBs had still not arrived from Elprint but we had an appointment with Dr. Melville (you may call him Neil) to come to ESTEC with the OBC flight hardware so we left Aalborg. Danny, Mike and Jakob in a Lupo. Christian and I on a plane. Arriving at ESTEC we could tell Neil that our “stuff” was on the way and just had to be “put together” before we could start testing. Confidence was still high. While waiting for the PCBs we continued writing software in open-loop and Jakob started programming the on board data handling system to run on the main processor on top of the eCos operating system.

PCBs arrived and we were thrilled to see the quality of the boards from Elprint. It would be a pleasure to solder those boards so we immediately set up a soldering laboratory in Neil's and Marie's office (don't do that – it is not anti-static) and soldered the (first) engineering models of the main board and the interface card. Even though we used a nice old Weller soldering iron, it was still like trying to use a cow for soldering the .5 mm pitch pins on the processors, UARTs and the FLASH. But it seemed to work out alright, and after many hours we were ready to power up the boards.

Five PROMs with first-draft software were burned (using our illegally acquired ROM programmer) and placed in the five PLCC chip carriers (btw, for flight hardware,

don't use PLCC sockets with pin grid array package that is not compatible with the footprint of the chips that go in the sockets).

The main board

The main board was powered up and debugging the hardware could begin. The number of problems presenting themselves in the hours, days, weeks and months to come doesn't fit the format of this document. In the beginning, there were problems with dead RAM – possibly caused by static electricity in Neil's soldering lab. They were for sure dead and had to be replaced. After some time we succeeded in getting the main board to show signs of life on the debugging interface and Jakob could start testing the data handling software in its real environment instead of using the ARM7 evaluation board. But by accident, a self-destruct version of his software found its way onto the PROM and killed the RAMs again.

When more memory chips are sharing a data bus (RAM, ROM and FLASH), you must ensure that only one chip has a defined electrical potential on its data outputs at any given time, otherwise the chips will source current from each other's outputs until only one of the competing chips is left alive. This time, the FLASH chip won and both RAM died from a mistake in the memory layout that the ARM7 was instructed to follow after the so-called remap-instruction.

The error was corrected but the board was in a poor shape after the previous de-soldering of the RAMs and couldn't survive another re-fitting of RAM. Another engineering board (to be known as the pre-flight board) was made a few days later in the cleanroom.

Meanwhile, the interface card was being tested in parallel with the main board after the completion of soldering in Neil's lab. Power on. On? Off. On again. Nothing smelling funny (which is good) but not much fun happening at all. Off. On. And such begins the story of external code fetch on the AT89C51CC03 from Atmel Corp.

The Story of External Code Fetch On the AT89C51CC03

The story of external code fetch on the AT89C51CC03 is long but I'll make it short-ish. The 89C51 processor can execute code from either internal FLASH memory or external anything-memory, in our case it was PROM. It says so in the datasheet for the chip. But when we configured the chip for external code fetch using the “external addressing” switch the processor just pretended to execute the external code. Using our logic analyser we could observe the CPU addressing the PROM just as expected from address zero and counting upwards and the bytes of code returned from the PROM to the CPU looked just like the code we had written. At address zero was instruction 0x02 followed by two bytes parameter which is a “long jump” to the

beginning of the code. So after addressing the first three addresses the CPU should jump to the address pointed to by the long jump and start our program. But the addresses just cycled upwards in binary counting sequential order. That meant that no long jump was performed (this analysis took several days). Strange.

Back to the datasheet. Everything wired up correctly? Yes. Correct instruction format generated by the compiler? Yes. Timing of the addressing and the PROM output correct? Yes. Anything out the ordinary anywhere??? No. Days went by staring at the logic analyser and trying to hook it up to different processors. Nothing. We called Jens, our supervisor, on a Sunday night and he started studying the datasheet to see if he could find something weird that we would never think of. This lead to the discovery of the boot loader jump bit which might not be set correctly. The boot loader jump bit is one bit in a register in the CPU that tells it whether or not it should execute its pre-programmed internal boot loader instead of the user application regardless of the hardware condition (like external addressing switch). The datasheet seemed a bit fuzzy on this point so we had to find a way to go in and get that jump bit out.

There are two ways of altering the contents the a register in the CPU: Via an external parallel programming device or by interaction with a program running on CPU. The first option was not really possible as the chips were already soldered to the board and we didn't want to remove them to do that (very damaging to the board and totally destructive to the chip itself). So we had to find a way to communicate with the boot loader resident in internal FLASH. This particular boot loader uses CAN interface to communicate with the world, but being without any CAN interface card for any of our computers we would have to build one out of the components in our pockets.

Luckily, we are engineers (if by diploma yet, then by spirit, anyway) which means that we have lots of components in our pockets so we found a PIC18F458 micro controller which has a CAN interface and an RS232 interface and quickly wired up a circuit on a piece of cardboard. Danny wrote some software for the PIC, Christian wrote some low-level driver for it for a PC and I designed a piece of software to parse hex files on the computer and communicate via the protocol of the boot loader. After a few days (everything happens in sequences of “a few days”) we had made a CAN interface for a PC with an RS232 connections (standard serial line) and we could chat with the boot loader and flip the darn jump bit.

The CAN interface system comprising the PIC plus other hardware on a piece of cardboard and a program for Linux would be known as CanTerm and it is the best invention of the entire OBC project. It has simply proven to be invaluable for developing a system with CAN interface and it has been modified/expanded many times since its birth in a smelly office at ESTEC.

The days of the nasty jump bit were over and we powered up the system in external addressing mode and waited for “something” to happen. Waiting, I said. Nothing. Now, things really started to look scary because, according to the datasheet, everything was beautiful. Only, it wasn't. No instructions were executed from the external memory so we contacted the chip manufacturer, Atmel. After some days they answered us and the answer contained the words “we are sorry for ...”. It turned out that the processors could not execute code from external memory because of a factory setting of a hardware security byte that disabled external code fetch. Atmel apologised for not writing that fact loud and clear in the datasheet when external addressing was described as a feature of the CPU.

There was only one way of resetting the security level of the chips to enable external fetch and that was by inserting the chips into a parallel programmer and perform a full chip erase. Chips had to be de-soldered while new chips were being erased and reprogrammed with the boot loader which we had grown fond of because it makes it possible to upload and execute software on the processors without burning a new PROM each time there is a change. And CanTerm was the number one tool for doing so.

One chip was de-soldered and replaced by a freshly erased brother and it was the moment of truth. While waiting for answers from Atmel we had done software development and testing on the CPUs using CanTerm and had some working software. A working piece of test software was burned into a PROM that was placed in the socket for PROM 1 for CPU 1 on the interface card. Power up. RS232 cable connected. One byte transmitted to the CPU. And it answered. External code fetch was now working after three weeks at ESTEC. Without hesitation we erased a bunch of chips to be used on the flight boards and rushed to the cleanroom where Jason awaited us.

Em111b

This is the story of Jason Page's cleanroom, room Em111b. Like most other stories, this could potentially become a long story. Jason's cleanroom is the place where SSETI Express is being integrated right now (this document is being written in this room) and Jason's words are the legislation inhere. And this is where we brought our PCBs and hundreds of components to integrate the OBC flight hardware using “proper equipment”.

The cleanroom is equipped with microwave soldering equipment, microscopes for use when soldering, 3D microscope for inspection of solder joints, all kinds of cleaning equipment and all chemicals needed for that, component tinning baths, harness manufacturing tools and wires/connectors and an infinite number of infinitely clever

tools for every situation. The D-connector pin extraction tool is an example of a remarkably effective and stunningly simple tool that I have a special place for in my heart (yes, I have a heart somewhere).

Having received instructions by Jason we started soldering a board that we classified as a pre-flight board to replace the dead engineering board. This could have been a flight board but we considered it a good idea to practise soldering skills on something non-flight before making the flight hardware. Christian started the work on the pre-flight board and after one day of SSETI General Assembly, I took over. For a week I was the guy at the microscope, visible from the webcam, apparently not moving one centimetre during a whole day. After about five days like that the pre-flight and the flight boards were completed except for a few components that had replaced components that suffered a cruel death on the engineering boards.

Soldering using the equipment in the cleanroom is a totally different story than soldering in Neil's lab using a cow. Even .5 mm pitch pins can be soldered perfectly without any danger of coming into contact with adjacent pins or solder pads. A soldering course is outside the scope of this document but it is at least worth mentioning that using the right tools and knowing what you are doing is essential to a good result. Following the soldering guidelines, however over-precocious they might sound, is instantly rewarding in that the boards work the first time and continues to work forever if they are properly soldered.

The cleanroom is clean and therefore everything in it should also be clean at all times including PCBs, so when soldering on PCBs you always keep it clean using IPA every ten minutes or so. This not only makes it easier to work with but also has the advantage that every ten minutes you take away residual flux and other kinds of more of less opaque substances/particles that may conceal erroneous solder joints.

Autumn activities – no life

August passed at ESTEC and September lured around the corner with a brand new semester and lots of lectures to follow back home at the university. Three out of six team members decided that the lectures and their semester projects were far more important than a student satellite project in the Netherlands so Mike and I didn't see much of them for the months to come – and the OBC was still not ready. The open loop software development throughout the summer had resulted in thousands of lines of code with hundreds of seemingly unresolvable bugs that tended to drive us crazy.

Returning to Em111b

The deal was that I should go back to ESTEC after finishing the OBC software back home in little over a week. The flight was booked and departure day came. The

software looked pretty good and I flew to A'dam with Mike backing up from Aalborg sitting in the SatLab there with our engineering model adding the last coats of chrome to the software while I was working in the cleanroom soldering the last few components and soldering redundant wires onto the boards.

It turned out that the days of the hundred bugs in the software were not over yet and with me in ESTEC and Mike, the only remaining team member, in Aalborg it was almost impossible to do efficient debugging. So Lars made a quick decision from his recently acquired “management chair” (PhD student position). Mike left for ESTEC with twelve hours notice and he and I did some intensive debugging down here. But strange software bugs just seemed to replace themselves by even stranger bugs when the EM was moved 700 km south-west of Aalborg.

Two days later Mike flew back to Aalborg with the EM and more man power had to be brought in. This meant dragging the Grumpy Old Men away from their “research” and down to the cosy SatLab where we learned to live our lives in complete isolation from the other students in our department though most of them were based just next door (including the three deserters).

After ten more days at ESTEC the OBC flight hardware looked pretty much ready to fly except for the empty PLCC chip carriers that just waited for five PROMs with working software to be plugged into them. But the software was not at all ready yet when I flew back to Aalborg.

Aalborg again

Back in Aalborg I immediately returned to the SatLab to assist in the software debugging. Meanwhile, Other Jacob had been enrolled by Mike to help with the software. One of the major problems was getting the CAN driver on the 89C51 to work properly. The original programmer of the driver had left the project and the bug count was scary so Other Jacob decided to completely redo the driver from scratch in a more logical and thorough way which turned out to be a brilliant idea.

Jakob (not Other Jacob) had told me that the on board data handling system was finished and had “no known bugs”. I read through most of the code and found huge parts missing (you can't really see that a missing part is huge unless you have an idea of how it should be implemented – I did) and the parts that weren't missing were pretty much covered in bugs leaving close to no bug-less functionality left. With close to ten thousand lines of code to debug I was looking forward to many long days in the SatLab.

I started debugging the buggy parts and filling in the missing parts (like e.g. telemetry and picture download and communication with the modem) and the data handling

system was taking shape. One of my favourite games was “Finding the FLASH Bug of the Day” which I could continue doing for more than a week. And still, three weeks later, I find small bugs in the FLASH storage code that are mission critical rendering telemetry storage impossible (now it all seems to work, though). Isn't that fun?!?

After two weeks of living in SatLab the software was close to a release candidate. By the way, don't try to a use fixed baud rate for the CAN boot loader in the AT89C51CC03. By default, is uses automatic baud rate detection, which doesn't work that well when more than two CAN nodes are connected to the network, but setting the baud rate to a fixed value on CPU 2 on our interface card the CPU stayed forever silent. The baud rate is set by setting three internal registers via the boot loader by sending it certain CAN messages described in the datasheet. But I guess the datasheet had got the numbering of those registers backwards or something because – silence. So, we had to solder a CPU 2 onto another interface card and hook it up to the other board. The OBC engineering model now consisted of three, not two, boards. Furry muff...

Anyway, we had a design review for AAUSAT-II the day before my third trip to ESTEC with “the completed flight OBC” and we were to present our final design of the OBC for AAUSAT-II which we had not come around to think about yet. We just knew for sure that we didn't want more than one processor! The utility CPU had proven to be a real menace and an ARM7 with built-in CAN controller had reached the market just a few months ago so if that could be acquired for AAUSAT-II soon it would be much better. Also, we considered to use an ARM7 with internal RAM and FLASH in order to completely eliminate the risk of output-enabling more than one memory unit at a time causing cruel death of RAMs and FLASH chips. Then, we would only need to plug the external PROMs to the CPU and that would comprise a complete OBC.

On the morning of the design review we had Holger Eckardt (UHF, SSETI Express) visiting SatLab to help us out with the SSETI Express KISS-TNC modem engineering model that didn't seem to work when we attempted to integrate it with the on board computer. After a while we declared the modem dead and the integration of that piece of hardware was interrupted. After the review we burned the first five “flight” PROMs for the OBC that I was to bring to ESTEC the next morning.

ESTEC once again

Once again, I woke up early in the morning in an aeroplane on my way to A'dam via Copenhagen. This could only mean one thing: I was on my way to ESTEC with yet another “flight ready OBC” as usual. I arrived at ESTEC and rushed to the cleanroom (my second home). Here, I spent an hour or two soldering the ACDS flight board for

Lars and helped mommy and daddy scrubbing down honey comb panels before they had to go to the Dutch marines to be sprayed black for the ever important “thermal reasons”. Hairy chuff...

The next day went by with a little bit of soldering and wire crimping and ended with an “OBC-in-a-box-alive-and-kicking”. This time, the OBC was very close to flight status: Go! The only “but” was that the utility processor software on the PROMs in the PLCC socket was compiled without out ARM-RESET option which means that it didn't perform power-up reset of the main processor as it was intended to. Not a fatal mistake, yet very annoying, because it meant that I had to solder a reset wire directly onto the main board in order to be able to start up the OBC and begin testing it. And so I did.

The Self-Destruct Wire

This is the story of “the Self-Destruct Wire” of the SSETI Express On Bard Computer flight model 00. The reset wire, that I had so beautifully and carefully soldered onto the main board enabling me to start the ARM CPU up, turned against me in a moment of unawareness, I guess. Being connected directly to the reset pin of the CPU this wire was the perfect point-of-attack for any high-voltage terrorist wanting to permanently disable the OBC. As the wire was “firmly secured and thoroughly electrically isolated from harmful power sources”, an act of what's-his-face caused an unfriendly electric potential to find its way to the tip of the pin on the end of the Self-Destruct Wire causing instant self-destruction of some internal circuits of the main CPU. At first, the CPU pretended to continue executing code but nothing deterministic came out of it. No welcome greeting at power-up.

Half an hour of frantic hardware debugging went by. This simply could not be true! The computer was working perfectly a few minutes ago! Approximately thirty minutes after time of impact the FLASH resigned in a puff of blue smoke (not quite a puff, but one should try to keep technical documentation as animating as possible). I remembered that fault just a little bit too well – more chips had been output-enabled at the same time meaning that no chips on the data bus (CPU, RAM, FLASH and PROM) could be trusted any more. Solution: Make a new flight board. Lesson learned: Don't solder self-destruct wires onto flight PCBs.

Yet Another Flight Board

Some people call me “lucky bastard” after this incident. Normally, it would take days or weeks to order new components for a computer like this not to mention having a new flight PCB manufactured by Elprint. But, fortunately, we were also building an OBC for a Russian “student” satellite, the Baumanetz, and new improved boards and

components for three complete computers were kicking around in SatLab back home, and the Grumpy Old Men had already scheduled a visit to ESTEC the next evening. Fantastic, I thought, they could bring “new stuff” and I could solder a new flight board the following day. And so I did.

Forty eight hours after self-destruction a brand new OBC flight model was greeting the “not-so-unhappy” Aalborgians with the good old

+SSETI-ExpressPlatform: Atmel AT91/AAUOBC (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, Red Hat, Inc.”

RAM: 0x03000000-0x03200000, [0x0300bf68-0x03200000] available
RedBoot>

which is the welcome screen of the boot loader for our ARM7 which enables software to be downloaded into RAM at runtime without burning a new PROM. With the new board being as finished as the old one, the only remaining task was soldering a thermistor onto some wires and gluing it onto the surface of the ARM processor. This meant having wires routed on top of the board from the plated holes to the CPU, and there is only one correct way of doing this: The Jason way.

The Jason way of “modifying” a board (using space qualified teflon isolated wire, of course) is by routing the wires using only ninety degree or forty-five degree angles while keeping the wires parallel and straight. I can tell you this is a tedious task that could drive most people crazy (too late for me). The wires were routed on a board without components (easier to work with) and bent into shape – held in place with small pieces of capton tape and then moved to the flight board where it was soldered and then glued down in three places with epoxy. Finally, the idea was to glue the thermistor itself onto the processor using thermally conductive glue (thermalbound), but the Thermalbound had suffered some kind of cruel death and was no-better at gluing thermistors onto processors than a duck. (Which in return doesn't speak much German, like most Italians don't do, either. Therefore, the Thermalbound must be Italian.)

Before all this thermistor-fun, which I did on a quiet day in the cleanroom after the Grumpy old Men had left again, the actual electrical integration of subsystems had begun with the success of the new OBC flight board.

Integration

Camera

Now the fun could begin as Morten, camera guy and Grumpy Old Man, was present in the cleanroom with his, at that time, fully functioning camera for the satellite. We

managed to integrate the OBC and camera without major problems and a picture was taken via a telecommand, stored in the on board computer FLASH memory and downloaded onto a laptop without loss of data. And then the camera suffered a cruel death (read more about that incident in the Report of the Grumpy Old Men).

Attitude Control and Determination System

With the camera out of the way I had time to engage in integration activities with the magnetometer – the primary sensor of the ACDS. Naturally, that didn't result in a hole-in-one but after having modified the ACDS software on the OBC to comply with the magnetometer it was playing the tune that a certain Grumpy Old Man was expecting. Done. Time for rewiring the pins of the D-sub 9 connectors for ACDS as this rather confusing interface had continued to confuse me even after I had made a sketch of all the translations: ADC_sun_12 -> Ch 0 -> ADC 3 -> Pin 8 -> Sun1D2, and so on for all the different inputs and outputs connected to ACDS.

Electric Power System

The OBC has two interfaces to the EPS: Power and data. The power input to the OBC has not yet been fed by EPS as it is not yet capable of feeding power. The data interface to the power distribution unit has been successfully tested with the EPS flight PDU and all telecommands have been tried in different sequences. The OBC received telemetry from the PDU and processed the keep-alive pings every twenty seconds. For some strange reason, a few pings were received wrong on the OBC and therefore not processed. This resulted in keep-alive timeout in the PDU which in return issued a shut down command to the OBC that gracefully shout down all its thread and was power-cycled by EPS (the power-cycle was simulated with light emitting diodes). The source of this missing-ping-fluke is yet to be determined.

The MAGIC box

The MAGIC box that controls the propulsion system is the only subsystem using a CAN interface to communicate with OBC. The baud rate of the CAN bus is 1 Mbps offering great chances for nodes on the bus to overrun the utility processor with large bursts of data. This fact was the source of some worried thoughts before testing with the MAGIC box as it is capable of transmitting rather large amounts of internal telemetry on the bus.

After unifying the OBC CAN protocol and the MAGIC CAN protocol to CAN 2.A by uploading new software to MAGIC the two systems were hooked up and the start-up message, an ITM packet on measurement ID 0x7D, was received by OBC and stored in the telemetry queue. All telecommands to MAGIC box were tested with different parameters and after only a few hours of testing the OBC<->MAGIC interface could

be declared “integrated”. Not even the huge amounts of telemetry from MAGIC showed any signs of problems.

UHF communication system

Next to EPS, UHF must be considered the most vital subsystem on the satellite. Being constructed by a German radio amateur rather than a student, this subsystem was considered to be among the most reliable systems on Express despite its complexity. The interface between the OBC and UHF is an RS232 connection utilising the KISS-TNC protocol (I hope). Because of the special grounding scheme of the radio transceiver all interfaces to UHF but the antenna must be galvanically separated from the radio.

In May, it was decided that the galvanic separation on the RS232 line should be implemented with two optocouplers on the UHF modem and that the UHF Rx line would use standard 5 V non-inverted logic while Tx would be an open-collector output requiring a pull-up resistor on the OBC. Later, this decision was overruled by the decision that all RS232 connection should use standard RS232 levels enabling a standard PC to be used for testing the interfaces. This decision was not communicated in such a way that Holger, responsible for the UHF system, was aware of it which resulted in him following the old interface rendering communication between OBC and UHF impossible without heavy modifications to one of the systems. This problem revealed itself when the UHF engineering model arrived at ESTEC to be tested with OBC and it was decided to change the interface on UHF to standard RS232 levels for the flight model meaning that new PCBs were to be manufactured.

Having modified the engineering model to conform with standard RS232 I could test the modem which was supposed to use the KISS-TNC protocol at 19k2 baud. I hooked the OBC and modem together and send a KISS-TNC encoded AX.25 frame to the modem while measuring the output from the modem to the radio. The modem is supposed to send a 2V FSK modulated audio signal to the radio upon reception of a valid KISS data frame but nothing happened on the output. The problem could be with OBC so I took it out of the loop and used a laptop to manually send KISS packages to the modem. I tried to most simple data frame consisting of

<FEND> 0x00 DATA <FEND>

Where FEND is a frame delimiter equal to 0xC0. The 0x00 in the first byte of the payload of the KISS frame indicates that the frames is a data frame, not a command frame. The data was one byte, 0x41 which is simply a capital 'A'. I tried to send this package at the correct baud rate without seeing any reaction on the modem audio output. I tried the same package at all baud rates available on the laptop. Nothing. Had this modem suffered a cruel death, somehow? Or does it use a different protocol than

KISS-TNC? SMACK, maybe? Or FlexNet?

After talking to Herr Güttner, the man who designed the modem, I got hold of the documentation for the modem and read through it (all in German, of course). According to that document, the modem should be using KISS-TNC by default and run at 38k4 baud on the RS232 line, so I tested it with 38k4 once again. Nothing. Time to toss the towel and leave the fate of the modem in the hands of someone capable of talking to it.

No problem, I just wait for a communications expert to show up at ESTEC to sort out the problems and get the UHF integrated with OBC before I go back home. This meant the second postpone of my return flight.

Part II

Lessons Learned

Lesson 1: Design and implementation takes time

From the beginning of the OBC project we regarded it as rather trivial to do the actual hardware design and implementation of a computer. We had all tried that before. But when time came to do the design we found ourselves covered in problems that had to solved with quick decisions which seen in retrospect wasn't always the optimal decisions.

Lesson 2: Centralised architecture beats distribution

The primary bottleneck constantly causing us problems during the software design was the number of processors having so communicate together. The communication between the processors was meant to be very simply and fail-safe but “fail-safe” and “simple” are not necessarily compatible requirements in a distributed system. Therefore, it slowly became quite complicated to programme the software to ensure robust communication using the simple protocols we had specified.

Had the five RS232 channels been implemented via UARTs connectors directly to the main processor and memory-mapped to its address space the incoming data could be expedited much faster with less chance of package loss. This leads to lesson three.

Lesson 3: Flow control on data lines is necessary

Loosing packages on data channels is not acceptable for an on board computer. But without flow control on the data lines going from multiple subsystems into a computer via a common serial communications channel, in this case the internal CAN

bus of the OBC, it cannot be guaranteed that every byte sent to the computer actually gets there before it is overrun by the next byte.

The utility processor is a classic example of a bottleneck in a network because it theoretically cannot processes all incoming data from both sides (CAN and ARM7) if the data flow of these communication channels reach a certain limit below the physical limits. This limit can be reached just by letting all subsystems transmit at the same time for a while until the local buffers in the CPUs on the interface card and in the utility processor are full.

At this point the best thing to do is to stop the data flow from the peripheral data source which can be done by means of hardware flow control on RS232 using DTR (Data Terminal Ready). This is implemented for the Russian Baumanetz satellite that features a modified version of the SEx OBC.

Lesson 4: Listen to Jason

This lesson did *not* come from bitter experience rather than from the joy of seeing the quality of the hardware when it has been implemented the Jason way. It doesn't take long to learn the Jason way if you have an open mind and accept that flight hardware has to be perfect.

Lesson 5: Follow lesson 4

Lesson 6: Make a schedule and multiply by л

This lesson is very difficult to learn for most people (including me). Stuff simply takes more time than you would expect. When getting an engineering assignment like developing a system for a satellite, make a schedule, use worst case duration for each task in the schedule, add a bit here and there, then multiply the whole lot by 3.14 and that is the amount of time you will actually spend on the assignment. On to the next Lesson.

Lesson 7: Building satellites makes you a Grumpy Old Man

If you are not yet a Grumpy Old Man then it means that you did not build a satellite. When venturing into the challenging world of space engineering you probably want to build a completely autonomous intelligent spacecraft with more fancy features than an electronic toaster. But after a while you discover that often it is not necessary for a toaster to be intelligent nor autonomous. In fact, you might even find it challenging enough to build the heating element of the toaster such that it complies with the Jason way.

Having had this experience you find yourself struggling to convince the less-

experienced ones that a spacecraft is complicated enough in it self – you don't need to add extra complexity just for the fun of it. It takes just about four months to become a GOM (Grumpy Old Man).

Lesson 8: Don't implement self-destruct wire

If you think it is a very clever idea to solder a reset wire onto you flight PCB with connection directly to you main processor then don't. It's *not* a clever idea!

Lesson 9: FLASH is bad. You shouldn't do FLASH. Or?

According to Danish space hardware experts at the Danish Space Research Institute it is forbidden to rely on FLASH memory for storing program code that a system on a satellite depends on. That is, FLASH can be used to store on board software as long as there is a boot loader residing in on-time programmable non-volatile memory, OTPROM, which loads the program code from FLASH and corrects any bit flips that have occurred in the code by means of error-correcting code like e.g. Hamming code or other block code or cyclic codes.

The OBC on SSETI Express used only OTPROM for all software on all processors in it giving no option for changes in the flight software. This choice was made in order to guarantee that the software running on the space segment is 100% equal to a copy running on the EM on ground.

But not all experts agree that FLASH is bad. Someone ought to perform a quantitative test on the probability of bit-flips in FLASH memory – perhaps a SSETI team?

Lesson 10: Don't buy return plane tickets

If you go to ESTEC to integrate a subsystem with lots of non-existent subsystems, don't buy a return ticket – you will not be on the return flight, anyway.

Lesson 11: Exchange protocol definitions

This lesson is quite important. When you agree on a certain interface like e.g. a data interface between two systems make sure that both parties have the same definition of the interface at all levels: Physical, electrical and protocol wise. All of these three levels may be difficult to change in the last minute when integrating systems, and one might have another definition of a protocol than the other. Therefore, make sure that all parties use the same protocol reference documents. If one person decides on a particular protocol then that person should make a complete protocol description available to anybody sharing that particular interface.

Lesson 12: If you are a poor student, don't build satellites

Speaks for itself. Building satellites means travelling. Travel reimbursements do not include local buses, food and wine which means that you will spend approximately 150 Euro a week instead of 50 Euro. As a student you don't have money for that.

Alphabetical Index