IMPLEMENTATION OF A RELIABLE DATA BUS FOR THE DELFI NANOSATELLITE PROGRAMME

N.E. Cornejo[1], J. Bouwmeester[2], G.N. Gaydadjiev[1]
ncornejo@gmail.com

1 Faculty of Electrical Engineering, Mathematics and Computer Sciences, Delft University of Technology, Mekelweg 4, 2628CD, Delft, The Netherlands
2 Faculty of Aerospace Engineering, Delft University of Technology, Kluyverweg 1, 2629HS, Delft, The Netherlands

The Delfi-n3Xt nano-satellite is the second Dutch university satellite currently being developed at the Delft University of Technology (TUD) as successor of the Delfi-C^3 that has been successfully launched in April 2008. Compared to Delfi-C^3, the Delfi-n3Xt platform provides significant advancements to the platform: a high-speed downlink, three-axis attitude control and a single-point of failure free battery. In total five payloads will be flown that generate a considerable larger amount of data compared to Delfi-C^3 that implies, as well, a robust and adequate design for the data handling system that interlinks the various embedded systems on board.

This paper examines the design and implementation of a fault tolerant data bus architecture as part of the satellite Command and Data Handling Subsystem (CDHS). Delfi-C^3 carries an I^{2}C protocol based implementation that currently experiences problems with data corruption and timeouts and is therefore subject of scrutiny and analysis in this paper. In particular, the relationship between error rates, master-slave speeds and processing overheads is evaluated in detail. After a tradeoff study between several bus standards for Delfi-n3Xt, the choice is once again an I^{2}C implementation, but with significant hardware and software improvements over the previous design. In terms of hardware, shielding and bus protection considerations are included in the very early stages of design. With respect to software, special care is taken in dealing with the varying clock speeds between slaves and masters, correct data handling and the feasibility of error detection and correction codes, as the amount of data generated by the payloads of the Delfi-n3Xt is significantly higher. The final result of this research is the selection of the most adequate reliability techniques and their implementation. This I^{2}C bus targeted middleware is intended for usage in the complete Delfi nanosatellite programme at TUD and for several other space applications in general.

1. Introduction

The Delfi-n3Xt nano-satellite is the second Dutch university satellite currently being developed at the Delft University of Technology (TUD) as successor of the Delfi-C^3 that has been successfully launched in April 2008. Compared to Delfi-C^3, the Delfi-n3Xt platform provides significant advancements to the platform: a high-speed downlink, three-axis attitude control and a single-point of failure free battery. In total five payloads will be flown that generate a considerable larger amount of data compared to Delfi-C^3 that implies, as well, a robust and adequate design for the data handling system that interlinks the various embedded systems on board.

2. The Delfi-C^3 Command & Data Handling Subsystem

2.1 Delfi-C^3 CDHS Overview

The Command & Data Handling Subsystem (CDHS) of the Delfi-C^3 was designed to process, store and manage all data and telecommands received by the satellite. It was also in charge of collecting all data from payloads and controlling the satellite itself based on relevant data. Therefore, the CDHS components included the on-board computer (OBC), the data bus, and software. The OBC used is the MSP430F169 from Texas Instruments, which comes bundled in the FM430 board of the Cubesat Kit from Pumpkin Inc. This microcontroller runs at 1 MHz and coordinates activities with the other subsystems which are managed with PIC microcontrollers from Microchip Technology Inc. These PIC microcontrollers handle all interfacing with the peripherals at their specific subsystem and communicate data back to the OBC. Moreover, they are also programmed to act autonomously in the case of an OBC failure or timeout.

The system data bus is an implementation of the I^2C by Royal Phillips from The Netherlands. Whereas many of the data buses currently used in space contain high resiliency and error correction, the I^2C standard does not define any type of EDAC methods, and therefore a parity scheme was implemented on top of the base standard. Physical cabling of the bus was realized via Nomex cabling and soldered connectors in each board. The bus speed was set to approximately 15 KHz to compensate for slower slaves running at 31 KHz in each of the distributed EPS nodes, which had to run at slow speed to diminish power consumption.

Finally, this design of a CDHS has to cope with unpredictable resets due mainly to the fact that there is no battery for the Electrical Power Subsystem (EPS) and therefore systems will be unpowered every orbital period during eclipse. The exact time of this reset is not predicted and the failure of any of the solar panels would also add to the uncertainty of this event, making it a pivotal requirement to be prepared at all times for power off. Due to the fact that the CDHS does not carry significant on-board storage, it is critical to be able to sustain power during communication windows as any data in volatile memory will be lost when the EPS shuts down.

2.2 Delfi-C^3 Experience and Issues with Data Bus

Although the software can handle most of the tasks required by the CDHS, it does experience errors and lost data entries from time to time. Determining the exact cause of the problem has proved to be a daunting task due to the large amount of activity, variables and factors that have to be taken into account. The time at which these errors occur also appears to be random. For these reasons, the approach taken here to shed a light on the problem is based on Bit Error Rate Analysis from testing a similar setup with the Delfi-C^3 service layer code. The preliminary hypothesis is that varying the bus speed has a noticeable impact in the amount of communication errors.

The results obtained from the BER experiments showed that indeed the EPS slaves running at 31 KHz experienced high amount of timeouts and error rate as the bus speed varied. The result of these

experiments is shown in figure 2. A marked rise is observed after the bus goes beyond 8Khz for the slowest slave, which is expected, since I²C guidelines dictate that a reliable implementation requires slaves running at least 10 times faster than the bus. Additional to this, parity calculation consumes a considerable amount of time, which of course, causes more impact on the slower running microcontrollers.

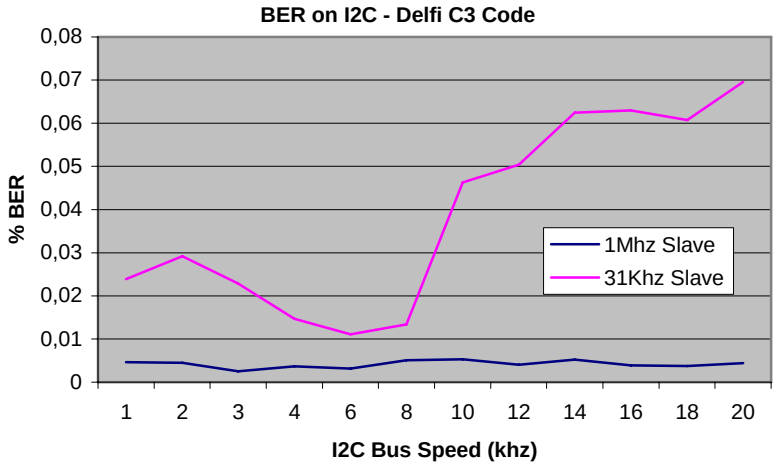

Fig. 2: BER Analysis of Delfi-C³ I²C Data Bus Setup

Finally, during integration of Delfi-C³, crosstalk was observed between both lines of the data bus, introducing a source of errors. The issue was resolved by applying a ground plane over the lines. The adhesive copper managed to suppress the inductive coupling to an acceptable margin. However, the watchdog implementation that would take action in case of hangs was not properly implemented.

3 The Delfi-n3Xt Data Bus

3.1 High Level Description of the Delfi-n3Xt CDHS

As in Delfi-C³, the Command and Data Handling Subsystem of Delfi-n3Xt comprises all components, hardware and software, required to interconnect, communicate and operate the subsystems within the satellite. In essence, the CDHS for Delfi-n3Xt will perform the same functions as it did for Delfi-C³, but will include numerous enhancements both in software and hardware.
The experience from Delfi-C³ has suggested numerous changes that should provide a higher degree of reliability. The OBC will now be custom designed as the FM430 requires more space than available and the new payloads and subsystems (i.e. ADCS) will require more capabilities. More importantly, and the focus of this paper, a correct implementation of the data bus has been deemed critical for this mission, thus, components have been chosen appropriately to guarantee proper functionality. Microcontrollers for all payloads and subsystems will also be chosen in a manner that complies with the data bus specifications, along with line shielding and proper cabling, all of which are explained further below.

3.2 Data Bus Selection

The selection of a proper data bus technology for Delfi-n3Xt has to comply with the requirements of the CDHS as described in the previous section. Traditional data buses used in space applications, such as those considered in ESA and NASA literature are high-end buses for long-haul communications between completely different and independent systems. Examples of these architectures are SpaceWire, FlexRay, SAFEBus, Ethernet and MIL-STD-1533. A nanosatellite such as

Delfi-n3Xt is in essence a collection of embedded systems, which require a compact and embedded data bus for inter-communication. Examples of serial embedded buses of this nature are SPI, I2C and CANBus. Additional to these, wireless standards, such as Bluetooth and Zigbee, are also used in embedded applications and are readily available in COTS components.

Wireless communications has been used previously in Delfi-C^3 for the Autonomous Wireless Sun Sensor (AWSS) with mixed results. Moreover, a wireless standard will not reduce PCB size, would probably create additional complexity and may interfere with the frequency bands of other systems onboard, such as the STX, since most COTS components work in the 2.4Ghz band. Therefore, a wireless standard for intercommunication would not be appropriate. The buses considered and compared for Delfi-n3Xt are the CANBus from BOSCH GmbH and the I^2C Bus developed by Royal Philips in the Netherlands.

The CANBus is mostly used in the automotive industry and can come in three variations: High Speed, Fault Tolerant and Single Wire. Some of the advantages of this bus are greater protection for EMC issues, hardware based error detection and tolerance to failures that avoid global bus hanging. However, the two most notorious disadvantages are its relatively high power consumption and added hardware/software complexity. Very preliminary estimates showed that power consumption for decentralized EPS nodes alone would consume around 600mW with a PIC18F2480 and around 1000mW in total if implemented with Microchip's MCP25025 I/O port. In terms of software, the PIC18F2480 needs 60 registers for the CAN module, while only 6 for the I^2C port. Moreover, the number of COTS components with I^2C support is higher than CAN and, although not as resilient, can be complemented with software and hardware features that will be detailed in the following sections. Power consumption is far lower and the required PCB area is lower as well. Finally, the heritage and experience gained from Delfi-C^3, along with the available tools, provide a higher degree of confidence on the suitability of the I^2C data bus for Delfi-n3Xt.

3.3 I^2C Serial Bus Overview

The I^2C bus is a multi master serial bus based on two open drain wires, one for data (SDA) and one for clock (SCL), both pulled up via resistors. The basic architecture is shown in Figure 3.

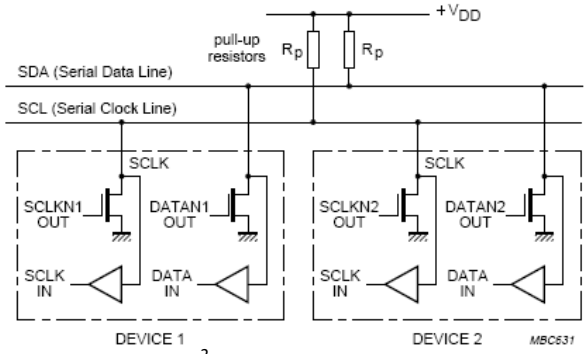

Fig. 3: I^2C Bus Basic Architecture

Data on the SDA line is only valid when SCL is low and its frequency is controlled by the master. It is common to use a speeds of 10 Kbit/s (low-speed mode) or 100 Kbit/s (standard mode), but recent revisions allow for speeds of 400 Kbit/s, 1 Mbit/s and 3.4 Mbit/s. Each device is identified by its unique 7-bit address (16 addresses are reserved) although recent revisions allow for extensions of 10-bit addresses that need to follow a slightly different protocol sequence. This basic communication sequence will always be initiated and finished by the master via START and STOP conditions and will also indicate which party will be transmitting data via an R/W bit. The START condition happens when the SDA line is pulled low while the SCL line is high and the STOP condition occurs when the SCL line is pulled high right before the SDA. The timing diagram for an I^2C transmission is illustrated in Fig. 4.

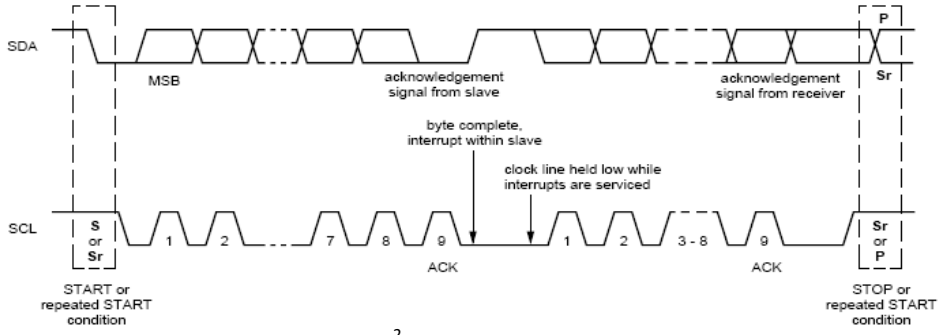

Fig 4: I^{2}C Data Transmission

The arbitration procedure is eased by the fact that both SDA and SCL are pulled up, and will therefore stay low when pulled down by a device. Every time a master pulls down a line it must check that the line actually goes to low. If this is not the case, that master has to back off because there must be another device using the bus at the moment. This mechanism also avoids data corruption because before the difference was detected and the device steps down, all previous bits were equal.

3.4 Data Bus Hardware Implementation

Although the I^{2}C protocol provides mechanisms for transfer, synchronization and acknowledgment as described in the previous section, it may be complemented with additional hardware mechanisms that ensure reliability for space flight. Moreover, the architecture has to consider a single-point-failure-free design as this is one of the main design goals of the project.

The actual implementation of the system data bus has many architecture possibilities, ranging from having a single bus through all subsystems to separate buses for different purposes. Overall, a single bus provides a simple implementation, while bus separation can provide more reliability and failure resistance. With all tradeoffs considered, the Delfi-n3Xt will be designed with a single bus, similar to the C^3 implementation but complemented with several enhancements. With all payloads and subsystems taken into account, the bus is expected to have 19 nodes, including the OBC as the bus master. In the C^3, the communication hiccups are heavily influenced by slow PIC microcontrollers attached to the local EPS switch on each node. In this next version of the satellite, the implementation will substitute these PIC microcontrollers with I/O ports (PCF8574) controlled directly via I^{2}C commands, which consume around 13 times less power (60uW -600uW to 8mW for the PIC) and can handle speeds of 100Kbit/s, unlike the PIC's which handled 15Kbit/s. An illustration of the architecture of a bus node is shown in figure 5.

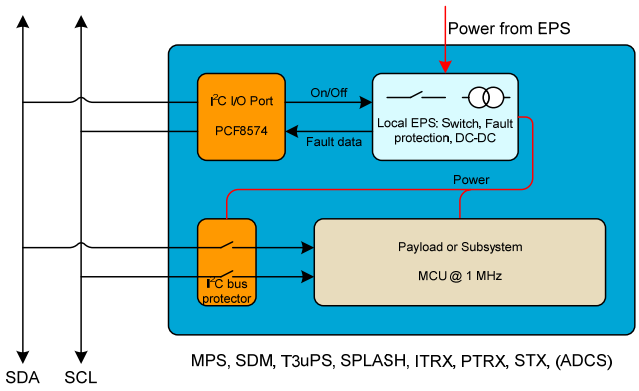

Fig. 5: Bus Node Architecture of Delfi-n3Xt

Another innovation in the design is the inclusion of bus protector circuits for 12 nodes on the bus, mostly the microcontrollers in each subsystem. This bus protector is conceptually a timer (Fig.

6) that monitors both the SDA and SCL lines. When any of these lines is turned low for a long period of time, the protector will disconnect the node from the central bus, avoiding a global hanging of the system. The timer is based on a simple RC circuit which can be tweaked to conformity by choosing the appropriate values of capacitance and resistance. The I^2C I/O ports are not interfaced with this protection under the assumption that it is an industry-proven chip that will cause no hangs.

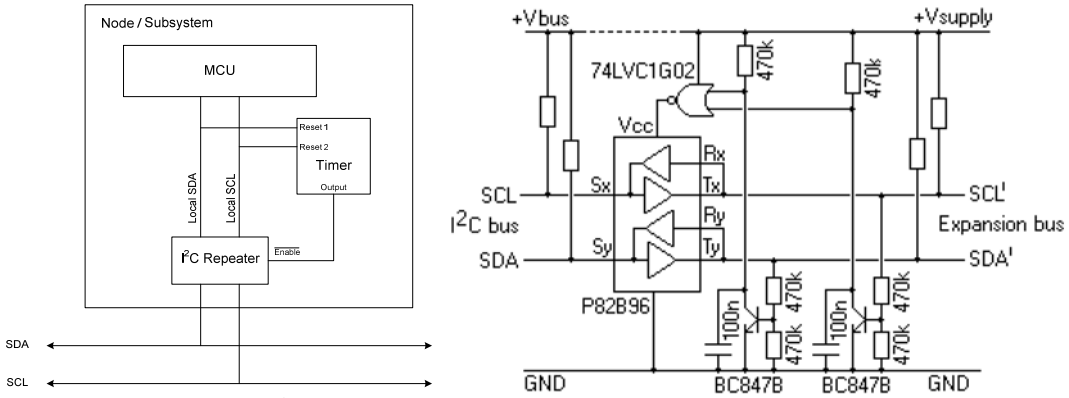

Fig. 6: I^2C Bus Protector Circuit, concept and implementation

Physical implementation of the bus is currently under discussion; however some considerations can be made on the type of wiring and connectors that would be ideal for the design. Due to the high flexibility, reliability and low volume, the preferred wiring would be Flex PCB's, but this is considerably more expensive than Nomex or Teflon wiring and has to be manufactured by a company. Some cost savings can be achieved by combining the EPS and data buses in the same wiring. A 10 pin wire allows for enough redundancy (4 I2C lines, 6 for EPS) and with a proper pin assignment crosstalk can be reduced. Preliminary calculations show that capacitance for this setup would be approximately 315pF, which is acceptable under the I^2C specifications and a resistance of around 0.44 ohms which also seems reasonable for correct operation of EPS. Since the experience with Harwin connectors in Delfi-C^3 was good, this would also appear to be a good design choice for Delfi-n3Xt.

3.5 Data Bus Service Layer Software Implementation

The software on board the satellite may or may not use an operating system to provide basic functions to the user. However when the final version of the software is implemented, the service layer software should remain the same and be used homogeneously throughout all the nodes in the system. Great simplification is achieved by using MSP430 microcontrollers throughout all bus subsystems; otherwise, a completely new service layer would have to be written for each different processor, which complicates testing and integration.

Each node must initialize the service layer engine via the n_initI2C function that receives two parameters: the node address and a mode (slave or master). This function will set up the I/O pins of the MSP430 and stop any ongoing activity in the I^2C port. It will afterwards activate the following interrupts:

- TXRDYIE: port ready for transmission. When in slave mode, this interrupt is activated when a master is requesting data or when in master mode and the I^2C module is ready to transmit data into the bus. The flag is cleared when the transmit buffer is full.
- RXRDYIE: receive ready interrupt flag. Activated when the I^2C module has received new data and is cleared when this data is read.
- ARDYIE: port access ready. When in master mode, this interrupt is activated either after all data has been sent or a specified number of bytes has been received and read from the

buffer. When in slave mode, the interrupt flag is set when a STOP condition is detected and, if receiving, all data is read from the buffer.

- OAIE: own address detection. This interrupt is available only when in slave mode and is activated when the address of this node is detected in the bus.
- NACKIE: no acknowledgement. The flag to this interrupt is set whenever an acknowledgement is expected but not received. This interrupt is available only in master mode and mostly used for error detection purposes.

The master MSP430 has to configure the I^2C clock via the combination of three registers: I2CPSC, I2CSCLH and I2CSCLL, all of which are combined as divider of the corresponding input clock for the I2C module (I2CIN). The relationship between the bus speed and the register values is illustrated in figure 7.

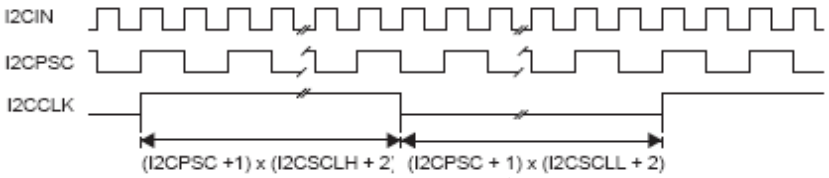

Fig. 7: MSP430 Registers for I^2C Bus Speed

To request data from a particular subsystem, the OBC will first send a command to the node so that it can prepare the requested data or execute the desired function. Soon after, a read operation will be made to the same node in order to read the corresponding data. The amount of time to wait for each command and data return will be defined by upper layer software. To write or send data to a subsystem, the OBC will call the n_I2Csend function that requires parameters for the slave address, a pointer to the outgoing data buffer, an expected data length and a timeout value. This timeout value must not be confused with the time to wait between the request of data from a payload to the actual return of data, but instead for the time (in milliseconds) to correctly perform the low level I^2C transaction. Failure to finish the transaction within the allotted time will result on an error return value. To read data from a particular subsystem, the OBC will call the n_I2Crecv function, which receives the exact same parameters as n_I2Csend. The possible return values that may be obtained from these functions are:

- I2C_OK: transaction performed ok and within time
- I2C_ERR_XMIT: transmission/reception error, usually when a STOP signal is detected but not all expected data was received.
- I2C_ERR_TIMEOUT: timeout during transaction.

To prepare data for a read from the OBC, slaves call the n_I2C_slave_packet function which receives a pointer to the outgoing data buffer and its length in bytes. The function will copy the data to another more encapsulated buffer which will be written to the bus when requested by the master. To perform accurate timing of I^2C transactions, timer A of the MSP430 is utilized as a general timer. The timer is set up in continuous mode and will cause an interrupt every time it reaches the value set up in the TACCR0 register. With a base clock at 1 MHz, and a value of TACCR0= ###, there will be a timer interrupt every ### seconds. The timer interrupt will add 1 to a global counter. Whenever a certain software component wishes to keep a timer, it will ask for the value of this global counter and by measuring the value of the global counter at any time, will be able to keep an accurate track of the elapsed time. A global array is implemented by which several of these sub counters can be stored and therefore used by many software modules at the same time.

The Interrupt Service Request for UART0 handles all interrupts related with the I^2C module and monitors each and every flag that has been preconfigured in the n_initI2C function as described previously. Before validating the source of the interrupt, it will restart the individual timer counter

of the service layer, since a call to the ISR indicates activity in the module and therefore there is no hang (and thus, no timeout). During reception and transmission, the master's ISR will just read from or write to the supplied buffers in the n_I2Csend or n_I2Crecv functions. For the slave, however, once the state moves from I2C_STATUS_RX and falls into the ARDYIFG interrupt flag, it means all data has been received and the program will jump to a callback function, which is part of the upper software layer and written for that particular subsystem.

4. Conclusions

Careful analysis of the results from the Delfi-C3 design and mission prompted several redesign options for the second nanosatellite of the Delfi Programme in TU Delft. The Delfi-n3Xt will borrow several ideas from the previous design, most importantly the I2C protocol and the data bus global architecture. There will, however, be important changes to improve the reliability of the system, such as better cabling for the bus and electrical power lines, bus protectors and local controllers that can handle the 100 Kbit/s communication speed. The service layer software will handle timing and interrupts assigned to the I2C hardware modules in each controller and will be used through out the rest of the systems.

References

[1] S. de Jong, G.T. Aalbers, J. Bouwmeester. Improved Command and Data Handling System for the Delfi-N3xt Nanosatellite. In *Proceedings of the 59th International Astronautical Congress 2008*, Glasgow, Scotland, September 2008.

[2] R.J. Hamann, C.J.M. Verhoeven, A.A. Vaartjes, and A.R. Bonnema. Nano-satellites for micro-technology prequalification: The Delfi program of Delft University of Technology. In *Selected Contributions of the 6th IAA Symposium on Small Satellites for Earth Observations*, pages 319-330, Berlin, Germany, April 2007. Springer.

[3] Philips Semiconductors. The I2C-Bus Specification, version 2.1. http://www.philips.com. January 2008.

[4] D. Paret and C. Fenger. The I2C Bus from Theory to Practice. John Wiley & Sons Ltd, Chichester, New York, 1997.

[5] Texas Instruments. MSP430x1xx Family User's Guide, SLAU049F. http://www.ti.com. 2006.