BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

DATA LOADER FOR COMPLEX TESTING OF ON-BOARD SYSTEMS

DATA LOADER PRO KOMPLEXNÍ TESTOVÁNÍ PALUBNÍCH SYSTÉMŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — Bc. DAVID HRBEK
AUTOR PRÁCE

ADVISOR — doc. Ing. RICHARD RŮŽIČKA, Ph.D., MBA
VEDOUCÍ PRÁCE

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Hrbek David, Bc.**
Obor: Počítačové a vestavěné systémy
Téma: **Data loader pro komplexní testování palubních systémů**
Data Loader for Complex Testing of On-Board Systems
Kategorie: Vestavěné systémy

Pokyny:

1. Prostudujte problematiku současných palubních systémů letadel, zaměřte se na standardy ARINC související s "data load", především ARINC 615A.
2. Seznamte se s jednotkou pro satelitní komunikaci Aspire 400 od firmy Honeywell. Analyzujte softwarové konfigurace jednotky Aspire 400 pro testování i letový mód. Seznamte se s nástroji pro přechod mezi softwarovými konfiguracemi jednotky Aspire 400.
3. Navrhněte pro jednotku Aspire 400 data loader, který umožní automatizované nahrávání všech potřebných softwarových konfigurací.
4. Navržený data loader implementujte v souladu se standardem ARINC 615A.
5. Ověřte realizované řešení a demonstrujte funkčnost ve sledovaných aspektech.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Růžička Richard, doc. Ing., Ph.D., MBA**, UPSY FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

ABSTRACT

This master's thesis summarizes theory on how to perform data load onto on-board computers of aircrafts. Specifically, how automated data load of Honeywell's Aspire 400 satellite data unit is done. First part of the text describes requirements and possible ways of the data load process, including standards that are applicable to this topic in the aeronautical industry. The second part describes the implementation of the data load process on the aforementioned unit.

KEYWORDS

data load, Aspire 400 satellite data unit, ARINC standards

ABSTRAKT

Tato diplomová práce shrnuje teorii ohledně nahrávání dat (data load) do palubních počítačů letadel, konkrétně do satelitní datové jednotky Aspire 400 od firmy Honeywell. První část textu popisuje požadavky kladené na proces nahrávání dat a možné způsoby jeho provedení. Jsou zde také představeny standardy týkající se tohoto tématu v leteckém průmyslu. Druhá část se pak zabývá samotnou implementací procesu nahrávání dat na zmíněné jednotce.

KLÍČOVÁ SLOVA

nahrávání dat, satelitní datová jednotka Aspire 400, standardy ARINC

HRBEK, David. *Data Loader for Complex Testing of On-Board Systems*. Brno, 2018, 71 p. Master's Thesis. Brno University of Technology, Faculty of Information Technology, Department of Computer Systems. Advised by doc. Ing. Richard Růžička, Ph.D., MBA

DECLARATION

I declare that I have written the Master's Thesis titled "Data Loader for Complex Testing of On-Board Systems" independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive bibliography at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this Master's Thesis, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno ....

author's signature

ACKNOWLEDGEMENT

I would like to thank my supervisor, doc. Ing. Richard Růžička, Ph.D., MBA, for his guidance. I would also like to thank my supervisor at Honeywell, Bc. Petr Kartous, for his insights on the matter, and the company itself for allowing me to use their facility and equipment to work on this thesis.

Brno ....

author's signature

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

This master's thesis Data Loader for Complex Testing of On-Board Systems was implemented in cooperation with Honeywell company. The goal was to come up with a way of automated and reliable data load of Aspire 400 satellite data unit (SDU). Aspire 400 is an aircraft on-board unit, currently under development at Honeywell. The unit has to use various software configurations during its life cycle, therefore there were multiple data load scenarios that needed to be considered.

In this document, the data load process in general is described (see chapter 2), including some of the important applicable standards used in aeronautical industry (see chapter 3). After this theoretical part, the Aspire 400 product is described, with emphasis on the key components, their software parts, and the equipment used in the production process (see chapter 4). Another chapter is devoted to possible data load scenarios (see chapter 5). The description of the data load process and its implementation is split into several subprocesses, starting with data load of the low level software parts (see chapter 6), over the first time data load of the target software (see chapter 7 and chapter 8), to the final ARINC 615A compliant data load (see chapter 9). In the end, the results of this thesis are summarized, together with ideas for possible improvements (see chapter 10).

The main scope of the thesis were the data load scenarios used in production on the unit level. See Figure 5.1 in chapter 5 for clarification. Implementation of these scenarios is described in chapter 7 and chapter 9.

Due to the commercial nature of the Aspire 400 project, this document is kept at a higher level of abstraction where possible. Of course, details important to the data load problematics are described as much as possible with compliance to Honeywell's policy of publishing information. Nevertheless, some of the information within this document can be regarded as Honeywell's intellectual property and should be treated as such. Any usage of information about the Aspire 400 project should be consulted with Honeywell.

2 DATA LOAD DEFINITION

Data load is a process of writing specific software onto some device, called the target hardware. The software is usually low level, e.g. some kind of firmware or lightweight operating system. In the context of this document, the target hardware is an aeronautical equipment, but generally, it can be any piece of hardware that needs some software for operation.

Data load is performed using a data loader. Data loader can be some special hardware, but it can also be an ordinary PC. The only requirements are the support of the required interfaces, e.g. Ethernet, and software which is able to communicate with the target hardware.

Data loader can communicate with the target hardware directly, i.e. be connected straight to it, or it can be connected to a bus to which multiple devices, including the target hardware, are connected. In the latter case, data loader also needs to be able to address the correct device or devices on the bus.

The goal of the data load process is to write the desired data into non-volatile memory of the target hardware on appropriate addresses, so that the target hardware can boot up and operate using the loaded software. The non-volatile memory is nowadays usually electrically erasable programmable read-only memory (EEPROM) or flash memory.

It is data loader's responsibility to ensure the data load process is carried out properly, i.e. the data is written onto the target hardware correctly and the target hardware operates as expected. Therefore the data loader has to support at least error checking and should preferably support also error handling. If the data load process is unsuccessful, the data loader needs to clearly report this and should provide more information about the problem to its operator. The data loader can also try to revert the target hardware's configuration to the point it was in before starting the data load process.

The term data load is sometimes used not only for writing, or uploading, data onto a device. It can also be used to describe the process of getting, or downloading, data from the device. In the latter context, data loader can be used to get information about the device it is connected to. For example information about the purpose, type, configuration of the device, etc. can be obtained.

This is a general description of the data load process. It may seem to be quite vague, but there is no precise definition. However, there are standards with more specific requirements for the target hardware and the data loader. These are described in the next chapter.

3 ARINC STANDARDS

Aeronautical Radio, Incorporated (ARINC) is a company which was founded in 1929 by four starting United States airline companies. The goal of the company was to serve the communication needs of the transportation industry (see [1], page ii). Till 2007, ARINC was owned by shareholders, most of whom were airline companies, both U.S. and others. In July 2007, ARINC was bought from its shareholders by a private equity company called The Carlyle Group (see [2]) and in 2013 The Carlyle Group sold it to Rockwell Collins Inc. Since Rockwell Collins does business in avionics, it sold ARINC's Industry Standards Organization subsidiary to avoid any conflict of interest. It was bought by SAE International (see [3]). SAE originally stood for Society of Automotive Engineers, but nowadays it is a global association of engineers and technical experts in the aerospace, automotive and commercial-vehicle industries. One of their goals is voluntary consensus standards development (see [4]).

Therefore, the ARINC standards are currently issued by SAE ITC (ITC stands for Industry Technologies Consortia), more precisely by ARINC Industry Activities, an SAE ITC program (see [5], page ii). This program organizes aviation industry committees, one of them being the Airlines Electronic Engineering Committee (AEEC). AEEC is an international body of airline technical professionals that leads the development of technical standards for airborne electronic equipment, including avionics and in-flight entertainment equipment-used in commercial, military, and business aviation. The AEEC establishes consensus-based, voluntary form, fit, function, and interface standards that are published and which are known as ARINC standards (cited from [1], page ii). AEEC is the body that prepared the standards which are important for this thesis.

There are three classes of ARINC standards (cited from [1], page ii):

- ARINC Characteristics, which define the form, fit, function, and interfaces of avionics and other airline electronic equipment. ARINC Characteristics indicate to prospective manufacturers of airline electronic equipment the considered and coordinated opinion of the airline technical community concerning the requisites of new equipment including standardized physical and electrical characteristics to foster interchangeability and competition.
- ARINC Specifications, which are principally used to define either the physical packaging or mounting of avionics equipment, data communication standards, or a high-level computer language.
- ARINC Reports, which provide guidelines or general information found by the airlines to be good practices, often related to avionics maintenance and support.

Below, mostly standards that are related to data load are briefly described. The class of each of the standards is specified and there is also a link to the bibliography, where information about the original documents can be found. In section 3.1, ARINC 429 is described. In section 3.3, ARINC 615 is described. In section 3.4, ARINC 615A is described. And in section 3.6, ARINC 665 is described.

There are other ARINC standards applicable to the data load process. The reason these are not described in greater detail here is that they are very outdated. For example ARINC report 603 (see [6]) sets expectations for data loader of airborne computers. However, this standard was released in 1985 and expects the data loader to transfer data from a tape cartridge.

Two exceptions can be found in the listed standards, namely ARINC 600, described in section 3.2, and ARINC 781, described in section 3.7. These standards are not data load related, but they are important for this thesis. ARINC 600 is important because it contains specifications and requirements put onto units, such as Aspire 400, in order for them to be compatible with standard airplane racks. And ARINC 781 sets characteristics of aviation satellite communication systems operating in L-band. Aspire 400 is such a system and that is the reason why the standard is described in this document as well.

3.1 ARINC 429

ARINC 429 is a four part ARINC specification (see [7], [8], [9], and [10]) subtitled *Digital Information Transfer System* (DITS), or *Mark 33 Digital Information Transfer System*. It defines a data bus widely used in avionics.

The standard describes physical and electrical interfaces of the bus, and also a protocol supporting local area network (LAN) within the aircraft. The term bus may be considered to be misleading in case of ARINC 429, since bus usually supports multidirectional transfers of data, while ARINC 429 supports only one way transfers from one source to up to 20 recipients. If the connected device needs both to transmit and receive, it has to do so on separate lines. Each line constitutes of one twisted and shielded pair of wires.

Bits are being transmitted using the bipolar return-to-zero modulation. This means clocking is part of the transmission. When transmitted, logical one has voltage of 10 $\pm$ 1 V, logical zero has voltage of -10 $\pm$ 1 V. The bus supports two speeds of data transfer. High speed is 100 kbps a low speed is in range of 12 to 14.5 kbps. Data are transfered in 32 bit words. Words are separated on the bus by putting a gap of at least four bit periods between them. New word starts with the first bit transmitted after this gap. Most messages used on the bus consist of only one word, but packets of up to 512 words are allowed.

Each 32 bit word has to have 1 parity bit (bit 32, the most significant one) and an 8 bit label, which is stored in the 8 least significant bits (bits 8 to 1). The word is transmitted from the least to the most significant bit, i.e. label first, parity bit last. The parity bit allows simple error check at the receiver.

There are other bits that usually have set function within a word. Bits 10 and 9 can represent the so called source / destination identifier, which is used when the word needs to be addressed to a specific device. Bits 31 and 30, and in some cases also bit 29, represent sign / status matrix, which is used to report the hardware status of the device, but it can also be used to represent a predefined sign, e.g. plus / minus, north / south, east / west,

right / left, etc. More detailed information about the word formats can be found in [7], attachment 6 on pages 115 to 124.

The label is encoded as a 3 digit octal number, and its bits are actually flipped significance-wise, so the 2 least significant bits of the word form the most significant digit of the label, the next 3 least significant bits of the word form label's second digit, and the next 3 least significant bits of the word form label's third digit. This representation allows label to be in range of 0 to 377.

The label identifies the data type of the word, i.e. whether it is binary encoded, binary coded decimal encoded, or represents discrete values, and it also identifies what kind of information the data represent. For example, label with octal code 015 represents information about wind speed in binary coded decimal.

Since the amount of equipment that needs to communicate data within an aircraft increased a lot compared to the time the standard was first released, some labels are used in different contexts. To determine the context, the transmitting device is identified by the equipment ID. Equipment ID is encoded within a word as 3 hexadecimal digits, i.e. on 12 bits. The combination of label and equipment ID should always identify a unique type of message. The full list of these types can be found in [7], attachment 1 on pages 21 to 54.

Description of the binary encoded decimal labels and equipment IDs can be found in [7], attachment 2A on pages 55 to 60. The description states value ranges and resolution of the data, how many significant bits there are, in what units is the encoded number, etc. The same description for the binary encoded data can be found in [7], attachment 2B on pages 61 to 81. Definition of the words representing discrete values can be found in [8] (the whole document).

The data carried by a word can represent either numeric value, alphanumeric data encoded using ISO alphabet number 5, or graphic data. The last one is used to transfer data which are rendered on a display, e.g. a map in an aircraft.

ARINC 429 standard also defines techniques for file data transfer. File data can be transfered using either character oriented protocol, or bit oriented protocol, which is also called Williamsburg protocol. Two versions of bit oriented protocol are used.

Version 1 is described in [9], section 2.5 on pages 5 to 24. The description explains the concept of link data units (LDUs), which is used to split the data file into pieces that are sent using the data words, and then reassembled again. By definition, 3 to 255 words make up 1 LDU, and the size of the transmitted file should not exceed 255 LDUs. The protocol also contains details about system address labels, word timing, types of words used to manage the data transfer, e.g. request to send, start of transmission, end of transmission, etc.

Version 2 is not used anymore. It has been superseded by version 3, which is derived from version 1. Version 3 is IEEE 802 compliant MAC protocol using ISO/OSI data link layer. Details about architecture of this version can be found in [9], chapter 3 on pages 26 to 51.

3.2 ARINC 600

ARINC 600 is an ARINC specification called Air Transport Avionics Equipment Interfaces (see [11]). It defines mechanical, electrical, and environmental interfaces between the so called line replaceable units (LRUs) and the racks or cabinets in which they are installed.

LRU is a term used in avionics for a piece of modular on-board equipment, which, in case of its failure, can be quickly replaced. The malfunctioning module is simply replaced with a working one, and the service time, during which the aircraft has to be grounded, is reduced. This is beneficial, since it is very expensive for the aircraft's operator to have it grounded. The malfunctioning unit can be then inspected and eventually repaired someplace else, e.g. its manufacturer's facility, and time is not so critical anymore.

Besides the definition of aforementioned interfaces, ARINC 600 also gives guidance for the design process and acceptance process of these interfaces. And interfaces between the racks or cabinets and the aircraft itself are covered in the same way, including control and regulation of power applied to on-board equipment. Interchangeability of LRUs and racks made by different suppliers is also discussed. Following ARINC 600 guidelines provides (taken from [11], page 3):

- a system of modularized equipment
- a system of modularized installation in racks and / or cabinets
- a family of low or zero insertion force electrical connectors to provide the electrical interface between the equipment and the aircraft wiring
- a system of effective environmental control of the equipment

The standard defines a so called modular concept unit (MCU), which is the basic unit for the packaging and installation concept of LRUs. MCU defines a fixed height, length, and width. All LRUs following ARINC 600 specification should have this height and length, and their width should be a multiple of width defined by MCU, i.e. the smallest LRU should have width of 1 MCU and bigger ones can have width of 2 MCUs, 3 MCUs, etc. Maximum weight of LRUs is defined as well.

Fig. 3.1: ARINC 600 connector on LRU backplane

Furthermore, ARINC 600 sets forth parameters for

maximum LRU thermal dissipation, cooling of on-board equipment within the racks, and requirements for attachment of LRUs to these racks, i.e. ways of physical mounting, forces the attachment has to withstand (vibration, shock, acceleration), etc.

However, the most important thing from ARINC 600 for the topic of this thesis is the definition of LRU's connector properties. It also covers the largest part of the ARINC 600 document itself ([11], attachments 17 to 21 and appendices 3 to 5). In these parts, the types of connectors that make up the whole ARINC 600 LRU connector are described, for example properties of pins for discrete signals, connectors for Ethernet, etc. An example of ARINC 600 connector can be found in Figure 3.1.

3.3 ARINC 615

ARINC 615 is an ARINC report called Airborne Computer High Speed Data Loader (see [12]). The last release of this standard was made in 2002. Today, it is quite outdated and it is described here mostly for legacy reasons and the description is not very detailed. Nevertheless, some devices still try to be compliant with this standard. Some parts of the standard are very similar to ARINC 615A standard, which is described in the next subsection. Since ARINC 615A standard is more important for this thesis, the similar parts are described there.

The standard sets guidelines for development of two types of data loaders. Portable data loader (PDL), and airborne data loader (ADL). Physical requirements for these devices are defined in the standard, same way they are defined for these devices in ARINC 615A.

The data loader is using ARINC 429 interface and should have at least two outputs and four inputs. It should be able to operate at both high speed (100 kbps) and low speed (12.5 to 14 kbps). The on-board computers should address it in the sent words using label with octal code 226 for high speed and label with octal code 300 for low speed.

The media for storing the loadable data are 3.5 inch floppy discs. The exact properties and format of a disc are described in [12], sections 3.2.1 and 3.2.2 on pages 7 and 8.

The standard describes two configurations files for the data loader. First is called *CONFIG.LDR*, and second is called *EXCONFIG.LDR*. At least one of them has to be present on the disc, and if there are both, *CONFIG.LDR* is processed first. The configuration file contains the physical parameters used for the communication with the target device, settings for the initial action taken after the initialization of the bus, total number of data discs required for the data load, sequence number of the current disc, etc. The full description of both types of configuration files can be found in [12], section 3.2.3.1 on pages 8 to 11, and section 3.2.3.2 on pages 11 and 12, respectively. If some of the required configurations are not set properly, default values are used.

File transfers are compliant with the file data transfer defined in ARINC 429. They use the so called command / response protocol. Three types of words are used in this protocol. Initial words, which are used to start and maintain the communication, intermediate

words, which carry the actual file data, and final words, which contain checksum used for an error check and close the transmission. The whole scheme of the protocol is described in detail in [12], section 3.5 on pages 21 to 29.

3.4 ARINC 615A

ARINC 615A is an ARINC report called Software Data Loader Using Ethernet Interface (see [1]). It describes a data load protocol implemented using Trivial File Transfer Protocol (TFTP) and Ethernet interface for the physical connection. It is the most important standard regarding this thesis.

ARINC 615A sets expectations for and gives guidance on development of software data loading equipment. As already mentioned, primary goal of data load is to upload software onto the target hardware. Secondary goal can be to download information from the target hardware.

Even though the standard is focusing on data load over Ethernet, other avionics buses using elements of Ethernet protocol are also mentioned as possible physical connection for implementation of ARINC 615A data load protocol. Namely ARINC 615A over AFDX (see section 3.5) and ARINC 615A over CAN bus are mentioned.

The standard defines three categories of data load functionality:

- Portable Data Loader (PDL), which is a mobile device that can be used to perform data load on the ground or brought on-board of an aircraft to perform data load
- Airborne Data Loader (ADL), which is a device installed on an aircraft
- Data Load Function (DLF), which is a software performing the data load itself

The standard describes physical requirements for PDL and ADL, e.g. their size, weight, controls and indicators, power supply and circuitry, non-operating and operating temperature spans, etc.

Some recommendations regarding the removable transport media for the loadable software are also made. The media include for example USB sticks, CDs and DVDs, and legacy carriers like 3.5 inch floppy disks.

The most important part of the standard is the definition of the load protocol. It defines functions that are necessary to be implemented both on data loader side and the target hardware side for them to be ARINC 615A compliant.

The load protocol defines means to (cited from [1], page 22):

- upload ARINC 665 software parts (see section 3.6) to target hardware
- download data from target hardware
- get configuration information from target hardware
- interrupt at any time any of the three previous operations (interruption request can be made by the operator or by the target hardware)
- obtain subscriber information, such as MAC address, IP address, and target hardware identifier

To the last point, the subscriber information can be obtained using the Find Identification of Network Devices (FIND) protocol. FIND protocol allows its initiator (operator using the data loader) to identify all available FIND hosts (ARINC 615A compliant target hardware) on the network. The operator can then select the desired target hardware device or devices from a list.

The implementation of the FIND protocol is done using UDP datagrams on port 1001. The initiator of the operation broadcasts or multicasts a request for a response from all available FIND hosts on the network. Then it registers all valid responses that come within 3 seconds.

FIND hosts respond with a unicast message to the data loader. Information about host's MAC address and IP address are part of the UDP/IP datagram and other information about the host are part of the UDP payload. FIND protocol does not implement any error handling. If the request or the response are not valid, they are ignored.

Two types of FIND packets are defined. Information request (IRQ) and information answer (IAN). Each has a two byte header defining whether it is an IRQ packet (value 1) or IAN packet (value 2) and a variable length data. IRQ carries only one byte ASCII string terminator (value 0x00) and one byte packet terminator (value 0x10).

IAN carries information identifying the host. It contains five strings separated by the one byte ASCII string terminator. These strings are:

1. Target Hardware Identifier
2. Target Hardware Type Name
3. Target Hardware Position
4. Literal Name
5. Manufacturer Code

All the other operations defined by ARINC 615A, excluding FIND, are implemented using TFTP protocol, i.e. they are based on downloading from or uploading files to the TFTP server, which is part of the data loader. There are two types of files that are exchanged between the data loader and the target hardware. First being protocol files that are generated during the load process, and second being the files with the loaded software itself. While standard TFTP port is 69, TFTP services for ARINC 615A are expected to run on port 59.

There are three types of operations defined in the standard, which are implemented using TFTP:

- Information Operation, during which the data loader acquires information about the configuration of the target hardware
- Uploading Operation, during which the data loader uploads files to the target hardware
- Downloading Operation, during which the data loader downloads file from the target hardware

According to the standard, data loader has to implement all of the above operations, plus the FIND operation defined earlier, whereas target hardware does not necessarily

need to implement the download operation. Furthermore, the operations cannot run in parallel, and active operation can be aborted upon a request from the operator.

TFTP options may be implemented to gain higher efficiency of the file transfer. In such case, the negotiation of these options has to be supported as well, though. If one side does not support any of the options, standard settings have to be used for this option or these options. Transfer should never fail due to non-implemented option.

Standard TFTP protocol is extended for the purpose of the load protocol. Wait and abort messages are defined by the load protocol. They are implemented using the TFTP error message. This message is part of an error packet and contains an error code number, and an ASCII error message. Defined error codes are integers from 0 to 8. Error code 0 definition is *Not defined, see error message*. Load protocol uses this error code and utilizes its error message to define its own string encoded messages.

Wait message contains error string *WAIT:x*, where x is the wait time in seconds. The maximum wait time is 65 535 seconds. This message can be generated in response to a TFTP transfer request by either the data loader or the target hardware. The device receiving this message should abort the TFTP transfer and initiate it again after the specified delay.

Abort message contains error string *ABORT:xxxx*, where *xxxx* is string of four hexadecimal digits containing a status code. The status code can for example mean that the operation was aborted by the data loader, or by the operator.

Any target hardware instance in an aircraft is defined by an identifier called *THW_ID_POS*. *THW_ID* (target hardware identifier) is defined in ARINC 665 standard and *POS* (target hardware position) is represented by 0 to 8 alphanumeric characters. Both these strings are part of the IAN packet payload received upon a FIND request.

The *THW_ID_POS* identifier is used as a name for the generated protocol files, using different suffixes. The full list of the protocol files can be found in the original ARINC 615A document (see [1], table 6.4-1 on page 66). These files contain all the protocol overhead information like the protocol version supported by the target hardware, status of an ongoing operation, including a heartbeat signal of the target hardware, result indicators of the finished operations, etc.

3.5 ARINC 664

ARINC 664 is a seven part ARINC specification (see [13], [14], [15], [16], [17], [18], and [19]). It defines an Ethernet data network suitable for an aircraft installation. Each of the aforementioned documents describes some area, e.g. system concepts, Ethernet physical and data link layer, Internet-based protocols and services, etc.

Basically, the main goal of this standard is to set requirements and restrictions that have to be met in order for the standard commercial Ethernet networks and Internet protocols to be eligible for the use in aircrafts.

ARINC 664 standard is not a key standard for this thesis, therefore it is not described in greater detail here. However, AFDX network was mentioned in section 3.4 as a possible layer, upon which ARINC 615A standard can be implemented, hence at least a short description of AFDX follows.

Avionics Full-Duplex Switched Ethernet (AFDX) is a trademark of Airbus company. Airbus has it patented for safety-critical applications. AFDX network is able to provide deterministic quality of service (QoS) on a dedicated bandwidth.

There are two types of devices connected to an AFDX network. End systems and switches. AFDX implements the so called virtual links, which make an abstract layer and simulate a bus similar to the one defined by ARINC 429. Using virtual links, one source end system can create a unidirectional logical link to one or more destination end systems. Redundancy is used in the background, which means end systems actually communicate over multiple independent networks. In case of switch or link failure in one network, the connection shall not be interrupted.

3.6 ARINC 665

ARINC 665 is an ARINC report called Loadable Software Standards (see [5]). It defines the format of the loaded software. This definition includes the rules for part numbering, content, labeling, and formatting of loadable software parts (LSPs) and media set parts (MSPs). Subset of LSPs are loadable software airplane / aircraft parts (LSAPs). Compliance with the ARINC 665 standard assures, that software can be processed by standardized data loaders.

Each LSP should have exactly one part number (PN), which should be agreed upon by the aircraft manufacturer and the software supplier. Whenever a change is made to an LSP, PN should be changed as well. The PN format is defined as *MMMCC-SSSS-SSSS*, where (cited from [5], page 6):

- *MMM* is a unique, upper-case alphanumeric identifier called manufacturer's code, that is assigned to each software supplier
- *CC* are two check characters generated from the other characters in the PN
- *SSSS-SSSS* is a software supplier defined unique product identifier consisting of upper-case alphanumeric characters, except for alpha characters *I*, *O*, *Q*, and *Z*.

However, ARINC 615A compliant data loaders should not check the PN format in order to achieve higher backward compatibility and flexibility.

ARINC Industry Activities assigns manufacturer's code upon application. It also administers the already existing codes and a list of them can be found on ARINC Industry Activities website.

CC denotes 8 bit cyclic redundancy code (CRC) written as two hexadecimal digits. CRC is computed from the ASCII values of the rest of the PN characters.

An LSP consists of a header file and one or more data files. Furthermore, it can contain support files. Each file within an LSP should have a unique name with maximum length

of 255 characters including an extension. The filename of the header file should start with the three character manufacturer's code and the rest should be unique for each LSP from this manufacturer. Characters that can cause problems on some platforms, like spaces, *, /, etc., are restricted. Also, the only difference between two filenames cannot be in the use of uppercase and lowercase characters.

Each type of LSP file should have an extension. For example, header filename should end with *.LUH*, data filename should end with *.LUP*, etc. The full list of extensions for all file types can be found in [5], table 3.2.2-1 on page 28. Support files can have any user defined extension, as long as it does not conflict with the reserved ones.

The content of LSP header file is thoroughly described in [5], section 2.2.3.1 and its subsections on pages 9 to 20. For the data files and support files there are no expectations regarding their content or format. These types of files can optionally be compressed to save space and speed up the loading process, or they can be encrypted.

The standard also defines batch file part (BFP), which can be utilized to predefine a set of LSPs that should be loaded into one or more target hardware devices (positions).

MSPs in the context of ARINC 665 standard are the physical media, that are used to transport LSPs, and eventually BFPs. They also have PNs. PN should be agreed upon by the aircraft manufacturer and the software supplier. It should not be longer than 15 characters. The PN should uniquely identify the particular combination of physical media and the software content.

Each member of an MSP is identifiable by MSP's PN and the member sequence number, which should be from range of 1 to 255. Members of one MSP should use the same physical media, e.g. USB sticks, CDs, etc. LSP files can be distributed over more MSP members. However, individual files should never be split.

Each member of an MSP has a list of all contained LSPs stored in *LOADS.LUM* file, a list of all contained files stored in *FILES.LUM* file, and a list of all BFPs stored in *BATCHES.LUM* file. These files should be stored in the root directory of the MSP member. The full definition of the content and format of these files can be found in [5], section 3.2.3.1 and its subsections on pages 28 to 32, section 3.2.3.2 and its subsections on pages 32 to 37, and section 3.2.3.3 and its subsections on pages 37 to 40, respectively.

The standard also describes in detail the way MSPs should be labeled. Label should for example contain the MSP's PN, sequence number, content description, supplier identification, etc. The full description can be found in [5], section 3.3 and its subsections on pages 42 to 44.

3.7 ARINC 781

ARINC 781 is an ARINC characteristic called Mark 3 Aviation Satellite Communication Systems (see [20]). It sets forth the desired characteristics of satellite communication systems which are using Inmarsat satellites and operate in L-band (band from 1518 MHz

to 1559 MHz for reception and 1626.5 MHz to 1660.5 MHz and 1668 MHz to 1675 MHz for transmission).

The communication system consists of multiple parts, which are individually described in the standard. To simplify it, the system can be viewed as a satellite data unit (SDU), an antenna system, and an SDU configuration module (SCM). The standard broadly discusses the radio frequency (RF) parameters put onto the whole system and its individual parts (e.g. frequency ranges, limits for RF output power, power of intermodulation products, error vector magnitude, spurious emissions, etc.). Interfaces, both those provided by the SDU for cockpit and cabin services, and those for interconnection of the system parts, are also described, as well as the physical parameters, power supply, cooling, and many other parameters.

The standard also explains Inmarsat services, their types (Classic Aero, Swift 64, Swift-Broadband), parameters, etc. Services that should be provided by the SDU, for example aircraft communications addressing and reporting system (ACARS), are discussed, too.

There is also a brief mention about the data load. ARINC 781 states, that the SDU should be designed so that all embedded software components can be loaded through industry standards ARINC 615 and ARINC 615A data loaders. It should also be possible to download the owner requirements table (ORTs) from the SDU to a data loader. SDU software files should be compliant with ARINC 665 (taken from [20], pages 70 and 71).

4 ASPIRE 400

Aspire 400 is a satellite data unit (SDU) developed by Honeywell company. The target market for this unit consists of small and medium aircrafts. The purpose of this unit is to provide an aircraft with air-to-ground and ground-to-air connectivity. Basically, Aspire 400 can be viewed as a modem. Aspire 400 SDU is also a line replaceable unit (LRU, defined in section 3.2).

In this chapter, the internal structure of the SDU is described in section 4.1. This description is concerned with the structure that is important for the data load process. In section 4.2 and section 4.3, software configurations used on the two keys components of the unit are described. Possible data load scenarios tied with these configurations are described in chapter 5. The testing environment used in production of Aspire 400 SDUs is described in section 4.4.

4.1 SDU structure

Since Aspire 400 is currently an ongoing commercial project, the description of the SDU below goes only into detail necessary for the matter of this thesis. The full hardware structure of the components and their connections cannot be revealed, since these details could be used by Honeywell's competitors.

From the data load point of view, there are two important components in an SDU. Multiprocessor Card (MPC) and Dual Aeronautical BGAN Card (DABC, BGAN stands for Broadband Global Area Network). There is also an SDU Configuration Module (SCM), a separate component outside of the SDU box, which contains some customer specific data. These components are described below.

SDU provides multiple ways of connectivity, including ARINC 429 interface, RS-232 and RS-422 COM port serial interfaces, and Ethernet interface. These can be utilized in the data load process. Most connectors of an SDU are physically placed on its backplane in the ARINC 600 connector. Some of them are also accessible on the front panel.

Both MPC and DABC are connected to a so called backplane, which is another card within the SDU. Its main purpose is simply to provide interconnections between MPC and DABC and SDU's ARINC 600 connector.

MPC, DABC, and backplane card are also sometimes called shop replaceable units (SRUs). Like LRU, SRU is a term used in avionics. It denotes hardware on a lower level than LRU. While LRU can be quite easily replaced in the field, piece for piece, SRU usually has to be replaced in the LRU manufacturer's facility, where the LRU is disassembled in order to replace the SRU. Hence the name shop replaceable unit.

A simple visualization of the loadable components of an Aspire 400 SDU and their interconnections can be found in Figure 4.1. It shows a simplified structure of the unit, with emphasis on the information important for the data load.

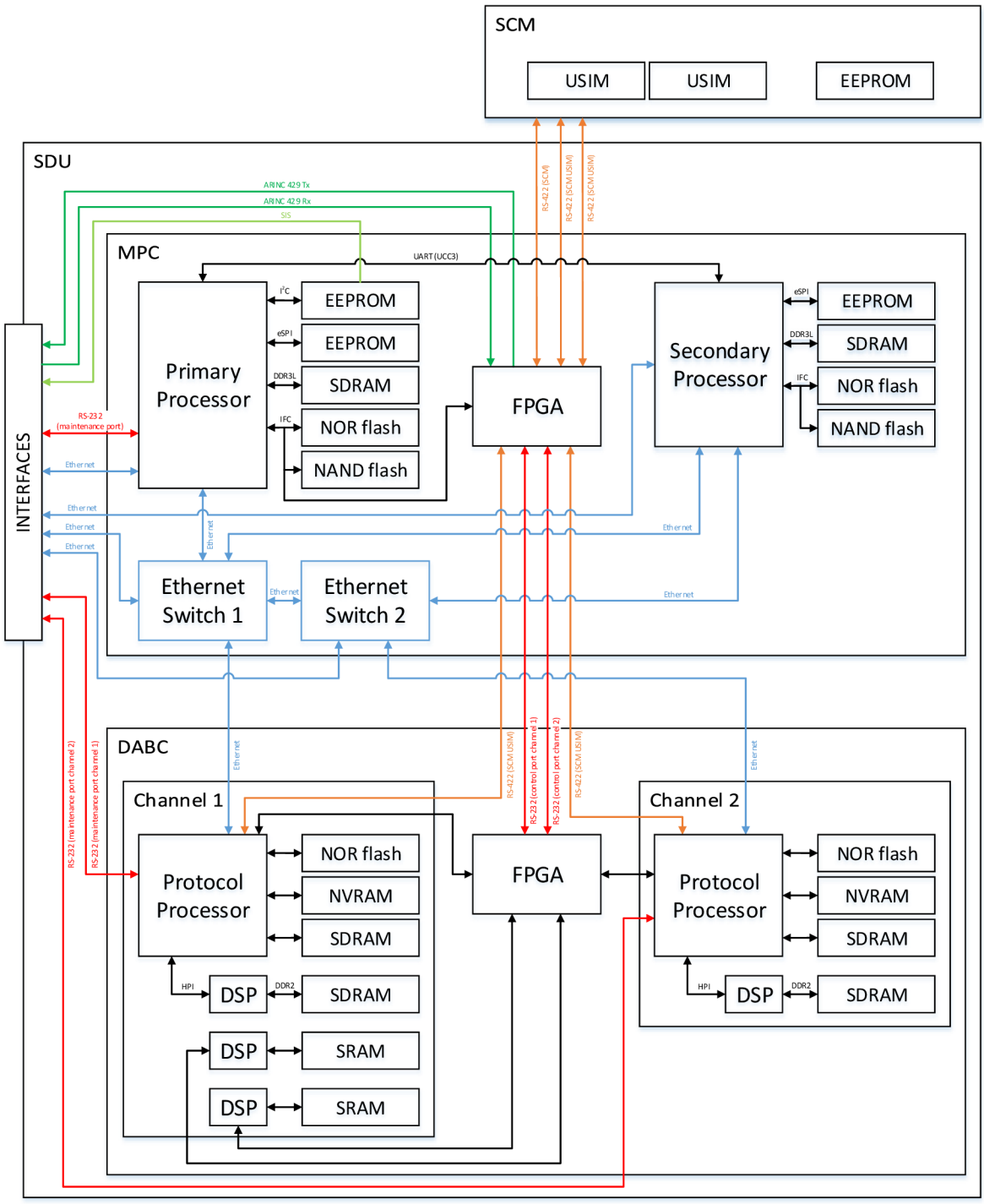

Fig. 4.1: Aspire 400 SDU loadable components and their interconnections

4.1.1 MPC

Main processor card (MPC) is the brain of Aspire 400 SDU. It has two processors that require data load. These processors are identical from the hardware point of view. They are both a system on chip with two 64 bit processor cores using Power Architecture

instruction set architecture. They provide high-performance data path acceleration and network and peripheral bus interfaces useful for aerospace applications. A block diagram of the system can be found in Figure 4.2. The diagram is taken from the processor's data sheet, but it cannot be cited in order to keep the processor model undisclosed.

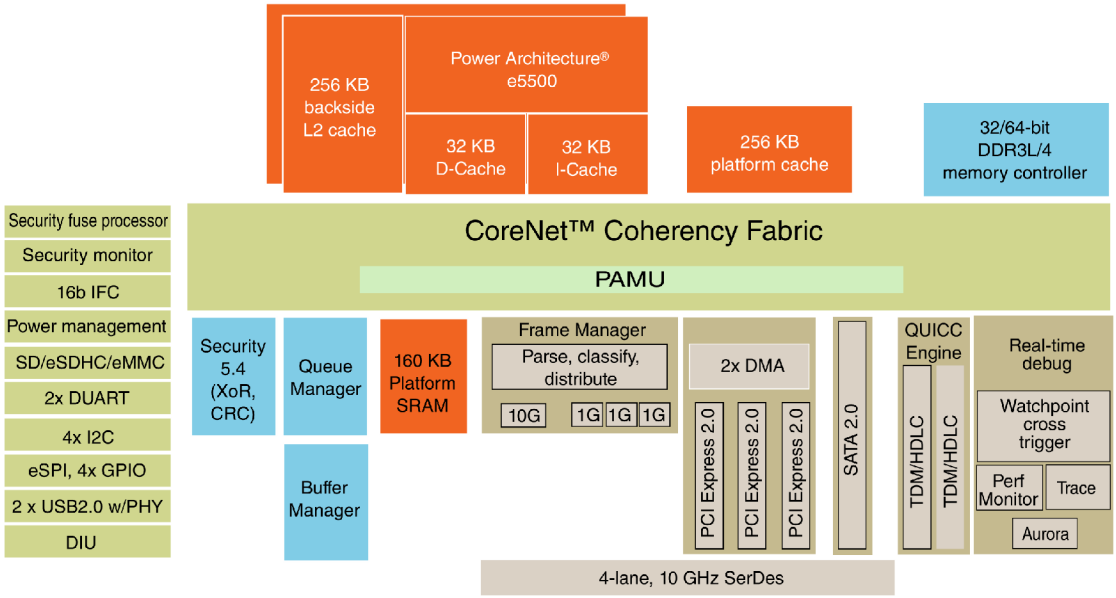

Fig. 4.2: MPC processor block diagram

From the functional point of view, the processors have different tasks in the SDU. However, their individual domains are not important for this thesis and therefore they remain undisclosed. For the data load problematic, it is important to state that one of the processors, hereafter referred to as primary processor, is booted first during the boot up of the SDU and controls the other components, namely the other processor, hereafter referred to as secondary processor, and the DABC.

Each of MPC's processors has four external memories. Three non-volatile and one volatile. The non-volatile memories are one Micron 128 MB NOR flash, one Micron 1 GB NAND flash, and one Microchip Technology 64 KB EEPROM. The first two memories are connected to the processor via the Integrated Flash Controller (IFC) bus. This bus is 16 bit wide and clocked at 100 MHz. It provides a NOR flash controller, a NAND controller, and a General Purpose Chip Select Machine (GPCM) controller (see [21], slide 4). The EEPROM is connected via Enhanced Serial Peripheral Interface (eSPI) bus.

The volatile memory is a 1 GB DRAM error-correcting code (ECC) protected DDR3L SDRAM, namely two 512 MB Micron chips are used. The L in DDR3L stands for low-voltage (memory is operating at 1.35 V instead of standard 1.5 V). ECC protection detects and corrects all single-bit errors and detects all double-bit errors. The ECC is a 256 MB Micron chip, which is not user accessible. All memories are organized in a virtual address space addressed with 32 bits for each processor.

In a typical boot up scenario, when the power is turned on, the system starts execution from a non-volatile memory (e.g. EEPROM or NOR or NAND flash). After that, the

code is copied from a persistent storage into RAM and execution continues from there (see [21], slide 3). Therefore the processor needs to be able to communicate with the non-volatile memory before any software configurations are made. In case of MPC, the boot up process is started from the EEPROM and NOR flash and the program data are copied to RAM from the NOR flash.

There is one more Microchip Technology 64 KB EEPROM, which is a part of the Standalone Identification System (SIS) interface. It is used to store the unit level configuration information. It is accessible both from the primary processor via an Inter-Integrated Circuit (I^2C) bus, and externally using the SIS interface. This interface allows user to check the configuration stored in its EEPROM without the necessity to have the SDU powered on. The SIS interface is made up by 8 pins. The connector is a 9 pin D-Sub connector placed on the SDU's front panel. The SIS interface can only read out of the EEPROM. The primary processor has to be used in order to write data into it.

The primary processor provides an RS-232 port called MPC maintenance port and both processor are reachable via Ethernet interfaces, either directly or via Ethernet switches which are part of MPC as well. All these connectors are accessible on SDU's backplane (they are part of the ARINC 600 connector).

The primary processor and the secondary processor are interconnected via a serial link which is realized by universal asynchronous receiver / transmitter (UART) using UCC3 (Unified Communications Controllers) through the QUICC Engine (see subsection 4.2.4).

The Ethernet switches on MPC are made by Atheros. The direct Ethernet connections to MPC's processors are realized using PHY chips (circuitry implementing physical layer of the OSI model), namely serial gigabit media-independent interface (SGMII). The Ethernet connections via switches are either SGMII or reduced gigabit media-independent interface (RGMII). Both switches and both PHYs are controlled by the primary processor over MDC/MDIO serial bus. The active device is selected by a 1:4 multiplexer.

MPC also contains a Microsemi FPGA, which is controlled by the primary processor via the IFC bus. This FPGA, among other things, implements the ARINC A429 interface, and it also controls reset signals to other components in the SDU (the secondary processor and DABC).

Other important parts of MPC are three pin header JTAG connectors, one for each processor (16 pins) and one for the FPGA (10 pins). There are also another two 10 pin header connectors, one for each processor's EEPROM. These connectors are used for testing and data load, as described in more detail in chapter 5 and chapter 6.

4.1.2 DABC

Dual Aeronautical Broadband Global Area Network Card (DABC) is the modem part of Aspire 400 SDU. This component is connected to an antenna mounted on an aircraft and using this antenna serves as the transmitter and receiver of the radio frequency (RF) signals. Based on the type of antenna, an amplifier might be also used, or it can be part of the antenna itself.

DABC also provides processing of RF signals, which includes modulation and demodulation, encoding and decoding, implementation of protocol stacks for Inmarsat services, etc. Inmarsat's SwiftBroadband (SBB) network is used for communication. SBB is a global IP-based packet-switched network providing aircraft connectivity with speed up to 432 kbps per channel (see [22]). It uses Inmarsat satellites to operate.

DABC, as the word Dual in its name suggests, has two independent channels. Each one has its hardware and is loaded separately. More information about the data load of DABC can be found in chapter 8. DABC is sometimes also called channel card (CC), since its purpose is to provide RF communication channels.

From the hardware point of view, which is again kept at a level necessary for the data load process, DABC has one FPGA common for both channels, each channel has one general purpose processor, one channel has one DSP and the other channel has three DSPs.

The FPGA is made by Xilinx. Apart from other things, the so called control processor is implemented for each DABC's channel within this FPGA. Its main purpose is to select the application that is supposed to be started during DABC's boot up process. The control processor is commanded via an RS-232 control port. It is also connected to the other components, which can be commanded via the control port as well.

The general purpose processor is a high performance low power system on chip based on MIPS32 instruction set. In DABC, it is called a protocol processor, since its main purpose is to run applications processing Inmarsat protocols. A block diagram of the system can be found in Figure 4.3. Same as with the MPC processor, the diagram was taken from the processor's data sheet, but the document cannot be cited because it naturally contains the model name of the processor.

There are three memory chips connected to this processor. One non-volatile 32 MB Micron NOR flash connected via the SRAM controller, one volatile 32 MB Micron SDRAM connected via the SDRAM controller, and one 256 KB Cypress Semiconductor SRAM connected via the SRAM controller. The last memory is volatile by nature, but in DABC, a condenser is used to make the data in it persistent. The condenser should last at least one minute, but in reality, it can hold the data much longer. Anyway, the data are persistent through a restart of the card, therefore this memory can be viewed as a sort of NVRAM. The protocol processor is also connected to the RS-232 maintenance port and to the Ethernet port.

The DSPs are made by Texas Instruments. The one used on both channels has a 128 MB Micron DDR2 SDRAM. It is connected to the protocol processor via Host Port Interface (HPI) bus and to the FPGA via GPIO lines. HPI is a parallel port through which the protocol processor can directly access the memory space of the DSP, including memory-mapped peripherals (see [23]). Protocol processor acts as a master on the bus.

The other DSPs used only on the first channel are identical and both have a 256 KB Cypress Semiconductor SRAM, which is organized as 128K 16 bit words. These DSPs are connected only to the FPGA via GPIO lines.

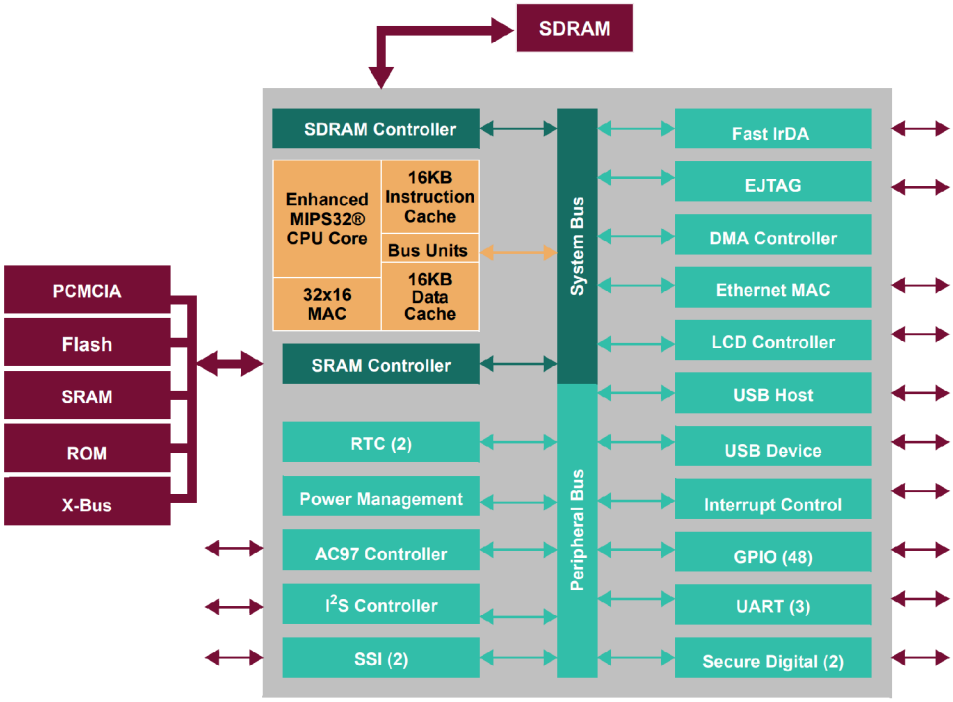

Fig. 4.3: DABC processor block diagram

A 40 bit virtual address space is used to map all DABC components into it to simplify the access to them.

Like MPC, DABC also has JTAG lines which can be utilized for testing and data load. Unlike MPC, on DABC there is only one dedicated JTAG connector (for the RF part) and other lines (for the channel 1 and channel 2 protocol processors and for the FPGA) are part of DABC's backplane connector, i.e. defined pins of this connector are dedicated for this functionality.

4.1.3 SCM

SDU configuration module (SCM) is a separate module containing an EEPROM memory in which some important unit-specific configuration data used by flight code are stored. It contains for example serial number of the unit, information about both hardware and software configuration and customer data. Customer data are stored in the so called owner requirements table (ORT).

SCM also contains slots for Universal Mobile Telecommunications Service (UMTS) Subscriber Identity Modules (USIM) cards, which are used by Inmarsat to connect to its Swift Broadband network. USIM cards are necessary for the DABC to be able to operate.

SCM is connected to an SDU via RS-422 serial interface and power is also provided by the SDU. The advantage of having SCM as a separate module is that it can stay in the aircraft while SDUs are swapped. When a new SDU is used, ORT does not need to be loaded again since it is part of the SCM, and USIMs can also remain untouched.

4.2 MPC Software Parts

In this section, software used on MPC is described. There are multiple possible software configurations of MPC consisting of a combination of the software parts described below. More information about these configurations can be found in chapter 5.

4.2.1 FPGA Configuration

A file with data for the MPC's FPGA is necessary to program it. Microsemi flash programmer allows usage of either PDB or STP file formats. The file contains data for the boundary scan test a the FPGA configuration.

4.2.2 RCW

MPC's processors use a mechanism called pre-boot loader (PBL). PBL is automatically executed when the processor is powered on and its main task is to load the reset configuration word (RCW), which is stored in processor's EEPROM. RCW is 512 bits long and contains encoded information used to initialize the RCW status registers. The information encoded within RCW sets for example clock speed, RAM attributes, etc.

4.2.3 Miniboot

Miniboot is a simple executable code that verifies the checksum of U-Boot (see subsection 4.2.4). It is stored in processor's NOR flash and run after the processor is powered on and RCW is loaded.

Miniboot first tries to verify checksum of the primary U-Boot image and if it is correct, Miniboot hands execution over to this U-Boot image. If this checksum is not correct, Miniboot tries to verify checksum of the secondary U-Boot image. If this image is correct, Miniboot starts its execution. If not, the SDU halts.

4.2.4 U-Boot

U-Boot, or Universal Boot Loader, is an open source project, which provides firmware for embedded systems. The core development is done by DENX Software Engineering company from Germany. The versions used in Aspire 400 project are customized at Honeywell. The purpose of U-Boot is to perform hardware specific initialization and testing (e.g. RAM test).

There are two main advantages of U-Boot. First is that it can boot up a system already loaded into the device's memory, and, unlike in most bootloaders, user can specify the addresses in memory used by the boot commands.

The second is it provides a command line interface via RS-232 port. This interface can be accessed when U-Boot startup process is interrupted by a keystroke during the prompted time period. The interface supports commands for writing to, or reading from the memory, modification of the environment variables, transferring files over the RS-232

serial interface (using for example YMODEM file transfer protocol), or Ethernet interface (using for example trivial file transfer protocol, i.e. TFTP), etc.

On MPC, there are two identical U-Boot images for each processor stored in its NOR flash. As already described in subsection 4.2.3, one image is primary and the other one is secondary. Integrity of an image is checked before it is started by computing its checksum. Normally, only primary image is used (unless it is corrupted).

However, one exception to this duplicity exists. The environment variables are stored in the so called U-Boot environment memory space in the NOR flash, and this space is unique. The environment variables are loaded upon U-Boot's startup and they contain values which determine the behavior of U-Boot. In case the U-Boot environment is found to be corrupted (again checked by a checksum), U-Boot sets all environment variables to default values. And in case the U-Boot image is found to be corrupted, this information is stored in the environment variables.

U-Boot also loads two microcodes (sometimes also spelled as μcodes). Each controls behavior of a certain hardware block within the processor. First is FMan, or Frame Manager, which processes Ethernet frames to provide classification and intelligent distribution and queuing for incoming traffic. Second is QUICC Engine, which serves for high-performance multiprotocol processing, e.g. Unified Communications Controllers (UCC). Both microcodes are provided by NXP, both are stored twice in the processor's NOR flash and their checksums are checked by U-Boot.

4.2.5 HBIT

Hardware built-in test (HBIT) is a software specifically designed to allow testing of all the components of an Aspire 400 SDU. The target of this testing is to make sure the hardware of the tested SDU is functioning correctly, i.e. all the components of the printed circuit boards and their interconnections are in place and are working as expected.

Basically, the purpose of HBIT is to provide an interface that allows setting or reading out variables. A typical test scenario is when a set of variables is set in a predefined way, and another set of variables is read out to see if the hardware reacts to the setup as expected. There are many different variables to cover all the test scenarios. There are discrete signals, analog signals, data sent over various buses, RF setups, etc.

HBIT is loaded onto MPC, but it also has the ability to control DABC in order to set it up for the RF test scenarios.

For some tests, the environment outside of the tested unit also needs to be set up. Most typically, voltages and currents are measured, so probes have to be set up correctly. Or some inputs and outputs need to be looped, temperature has to changed for the test, etc. This is not done by HBIT itself, but by the testing platform (see section 4.4).

HBIT also provides functionality called Continuous built-in test (CBIT). As the name suggests, CBIT is a version in which testing is continuous, i.e. the tested variables are read out with a defined frequency until the process is stopped. Compared to that, in HBIT, the

variables are read out on demand. The tested values that are read out can be compared programmatically, they can be logged, they can be visualized, etc.

4.2.6 Flight Code

Flight code is the full feature version of the software that is used on board of an aircraft. Prior to any regular in-flight usage, this software has acquire proper certification. It undergoes the so called qualification process, during which it is inspected and tested by all the interested aviation authorities, and, if it complies with all the requirements and passes the tests, it is certified.

In case of Aspire 400 SDU, more precisely the MPC, flight code is a Linux-based module system. Each module takes care of some specific functionality. It is basically a process. The so called message event service (MES) is implemented to provide an inter-process communication between the modules. MES also provides means for securing the communication, i.e. encoding and decoding the messages.

Flight code modules are distributed on MPC's processors. Each processor takes care of different parts of the SDU's functionality. But the functional domains of flight code are not important for the topic of this thesis, therefore they are not described further. Only the modules important for this thesis, e.g. data load controller (DLC), are described in greater detail in chapter 9.

Flight code is released in a form of flattened image tree (FIT). It is an image of the whole system, including all configurations. This image is part of the loadable package compliant with ARINC 665 standard. The standard in general was described in section 3.6 and for its application on Aspire 400 project see chapter 9.

4.3 DABC Software Parts

DABC is commanded by the MPC. Nevertheless, different configurations for DABC exist and the code is loaded separately. Moreover, as mentioned earlier, DABC has two separate channels, and each of them is loaded separately as well.

The DABC software is released in a form of image files (*.img* suffix). Based on the type of application, the file contains data for one or more of DABC's components. The application is usually loaded onto all DABC components it uses during one instance of the data load process. This ensures the software for individual components is compatible. But it is also possible to load individual components with a specific combination of software versions. This is especially useful for some extensive debugging, when user can create a customized software version for the component of interest and load just that one.

Multiple applications can coexist in DABC's non-volatile memory (protocol processor's NOR flash) at the same time. The active one is picked during the boot up using the so called loader program, which is described in the next subsection. Records about available applications are kept in a special table stored in the NOR flash as well. This so called

PDB table contains names and versions of applications, together with their checksums and addresses in the NOR flash.

Some environment variables can also be stored in protocol processor's NVRAM. Factory values are stored in the NOR flash and they are loaded from there to the NVRAM, where they can be modified. The reason for this approach is that the NOR flash always holds the factory data as a form of a backup and when modified, the change is done in the NVRAM, so no writes to NOR flash are necessary. This reduces the number of writes to this memory, which reduces its wear-off speed.

The list of DABC applications in this section is not exhaustive. Other types of DABC images, mainly for different testing purposes, also exist. However, these are not used during production testing and therefore they are not listed here.

A proprietary language called Binary Command Language (BCL) is used to command DABCs. It is used to communicate with DABC from any external device or component. A library for translation of BCL commands into binary and vice versa has to be available in order to use it. All of DABC's functionality is accessible using BCL commands. The commands can be sent either via control port or maintenance port (RS-232) or via Ethernet (TCP/IP stack).

Each BCL message has mandatory header and optional data payload based on the message type. The header contains information about sender (BCL address of the sending component) and recipient (BCL address of the target component). It has also information about the type of the message, its length, CRC, and other properties.

4.3.1 FPGA Configuration

Like on MPC, a file with data for the DABC's FPGA is necessary to program it. On DABC, Serial Vector Format (SVF) file is used. This file contains instructions that perform the boundary scan test a configure the FPGA into the desired state. SVF files are ASCII encoded.

4.3.2 Loader Program

After power is applied to DABC and the reset signal is turned off, loader program is initiated. This program reads the PDB table with information about all available applications mentioned above from the NOR flash and presents a list of available applications via the control processor interface (control port RS-232). Selection is done using ASCII encoding, i.e. application is selected by typing its name over the control port, terminated with a carriage return character. When a correct application name is supplied, the loader program copies the application from the NOR flash into RAM and hands over the control to the application.

All applications can actually be twice in the NOR flash. This is for security reasons. When application is loaded to DABC (written to its NOR flash) a copy of it can be made. It is used in case the primary image gets corrupted. Before the loader program copies the application into RAM, it computes CRC of the image it is about to copy to check

the image is correct. If the CRC does not check out with the one stored in the PDB table, user is informed about the error, but if the secondary image is available, the loader program tries to copy the application from there (it performs the CRC check again for the secondary image). Secondary image is only used if the primary is either not present or its CRC is not correct. A flow chart of the loader program functionality is shown in Figure 4.4.

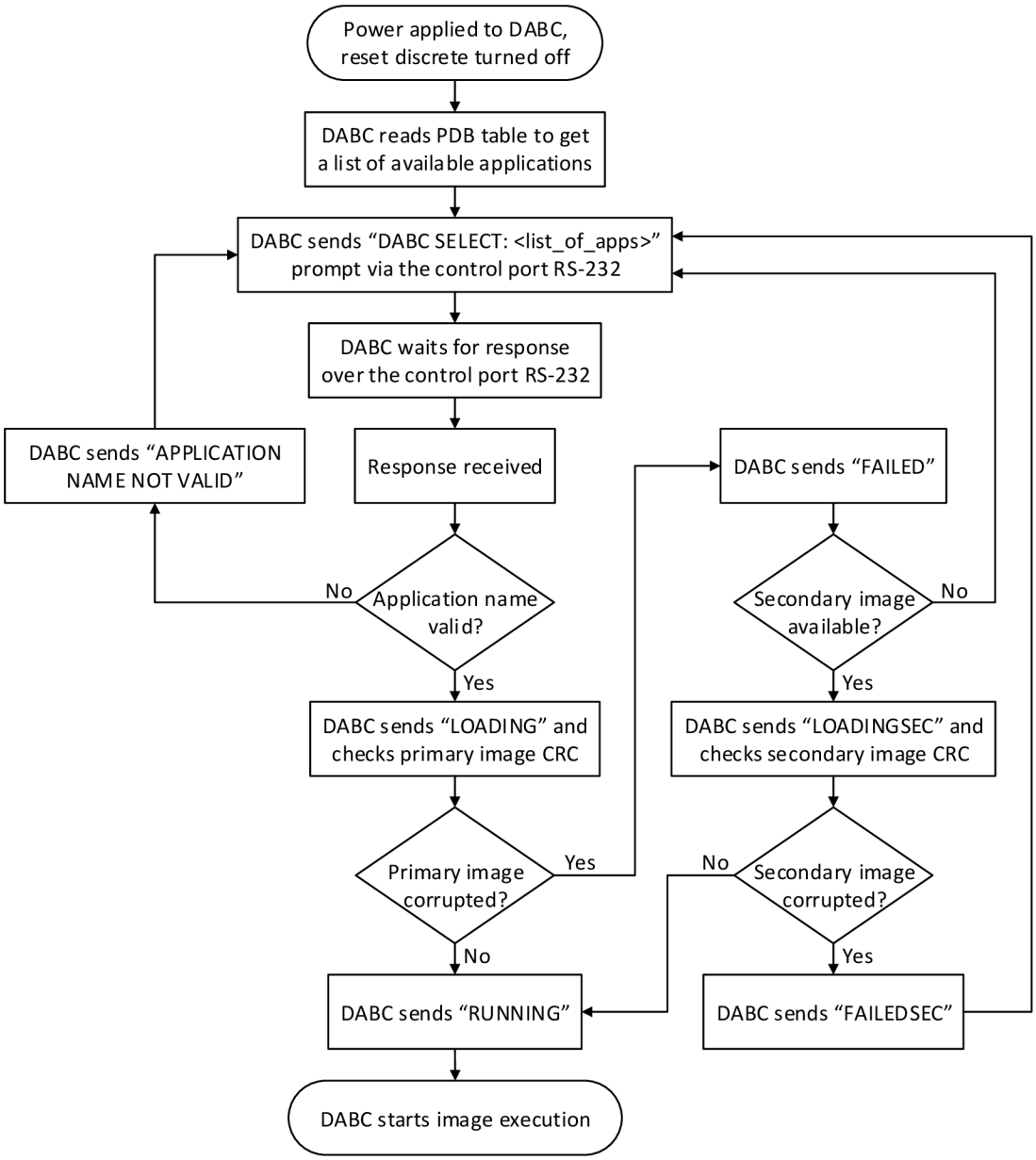

Fig. 4.4: DABC application selection

4.3.3 EBOOT

Emergency boot (EBOOT) is an application which allows overwriting DABC's NOR flash using BCL. Hence data load can be performed when this application is running on the channel that is about to be loaded. EBOOT is using only the protocol processor to run.

EBOOT also supports duplication of the other applications' images in the channel's NOR flash, which was mentioned in the previous subsection. When the BCL command to perform the duplication is received by the protocol processor, EBOOT tries to copy the other images present in the NOR flash from their primary position to their backup position. It first checks to see if the duplicates are already present and are exactly the same as the primary images. If so, no duplication is performed, both to speed up the process, and to omit unnecessary writes to the NOR flash. If not, EBOOT either performs the duplication or returns an error message when something goes wrong, for example if there is not enough space for the duplicate in its designated area in the NOR flash. There is also a BCL command that performs the opposite action, i.e. wipes the duplicates out of the NOR flash. EBOOT does not duplicate itself and it is the sole application that is able to perform this duplication and / or wiping.

Also, in contrast with loader program, when an application is running (not only EBOOT, but any), the channel can be controlled not just via control port RS-232, but via maintenance port RS-232 and Ethernet port, too. Other than that, EBOOT does not support any of DABC's functionality.

4.3.4 IBIT

Initiated built-in test (IBIT) application is used for testing DABC in operational use. This means that IBIT implements all functionality required to control DABC's hardware. IBIT is used during the production testing to command DABC to transmit and / or receive data via its RF module. BCL commands are used to achieve this.

4.3.5 SwiftBB

SwiftBB, or SwiftBroadband, abbreviated SBB, is the full feature flight code application used in an aircraft. It is designed to provide means of communication over Inmarsat's Broadband Global Area Network (BGAN). It handles the RF signals, implements all the necessary protocols, etc. In order to be able to use BGAN, the implementation of this application has to comply with SBB protocols defined by Inmarsat.

4.4 ATE

Automated test equipment (ATE) is an apparatus used for production testing of SDUs. The testing is as automated as possible to speed up the process. Ideally, an operator only plugs the unit under test (UUT) into the ATE and starts testing. All tests should be performed and evaluated automatically, including all necessary configurations of the

testing environment. The operator only needs to check the final status of the tested SDU to see if it has passed or failed, and if it has failed, the ATE should also give reasons of failure, so the SDU can be possibly fixed.

An ATE for Aspire 400 project should support testing of two SDUs simultaneously. The testing process is coordinated by a computer that is part of an ATE. This computer is running Microsoft Windows operating system and a program called TestStand from National Instruments is used to run and evaluate the test sequences.

TestStand has the ability to call various adapters and interfaces. Its advantage is that it can unify calls into various libraries, programming languages, etc. This layer is abstracted from ATE's operator, and he or she is presented only with quite simple and clear interface showing which test sequences have passed or failed. TestStand also creates a test report, can log the measured values into database, and more.

Other necessary tools are also installed on the ATE's computer. For example a TFTP server enabling an SDU in U-Boot command line mode to download data from it.

In the production process, ATE also serves as the data loader for the load of HBIT onto MPC and IBIT onto DABC in the beginning of testing, and for the first-time load of flight code after the testing is finished. U-Boot has to be already present on MPC, as well as EBOOT on DABC, when the SDU it tested via ATE, since ATE tests SDUs (i.e. tests at the box level), while these applications have to be loaded at the card level, as described in more detail in chapter 6.

5 DATA LOAD SCENARIOS

During its lifetime, SDU has to go through multiple software configurations. First, when an SDU is produced, individual components are manufactured, i.e. printed circuit boards (PCBs) are made and assembled. Then some initial tests are performed on these components, for example automated optical inspection (AOI), automated X-ray inspection (AXI), in-circuit test (ICT), boundary scan, etc. At this time, the components are blank, therefore it is necessary to load some software onto them in order to use them. The software has multiple layers, starting with bootloaders at the lowest level. There can be more, building up on each other and extending the provided functionality. On top of bootloader, there is usually some operating system and at the top level, there are the final applications.

In production, the low level software is usually loaded by the component manufacturer. During development, when changes even to the low level software might be required, or the software might get corrupted by improper work with memory, it can be sometimes necessary to flash the component at Honeywell, too. But in most cases, the software at the lowest level is loaded once onto a blank component and does not need to be changed further.

Once the individual components are loaded with at least the low level software, the unit could be theoretically assembled and shipped for SDU level production testing. But in order to make sure the components work correctly prior to the SDU assembly, software designed specifically for testing of all required features is loaded onto them and functional testing at the card level is performed. This testing software is HBIT for MPC and IBIT for DABC (these software parts were described in subsection 4.2.5 and subsection 4.3.4, respectively). Only after both MPC and DABC pass, SDU is assembled and the testing process goes further.

The SDU level production testing is performed at Honeywell and uses HBIT and IBIT as well. It can happen that versions of these software parts used during the card level testing are the same as versions required for the SDU level testing. In such case, testing can proceed right ahead. However, it is more likely that the card manufacturer is provided with a different version of HBIT and / or IBIT by Honeywell, and it is therefore necessary to load these software parts once again.

If the unit passes the tests at the SDU level, the final software configuration (flight code) is loaded onto it. This configuration supports the full functionality and contains the customer specific data, too. At this moment, production is finished and the unit is shipped to customer, ready for operation. Of course, there are also some possible post production data load scenarios, namely update of the flight code version and error identification.

A flowchart of one SDU's lifecycle is visualized in Figure 5.1. It is of course slightly simplified. Data loads are highlighted in red. The dashed ones on the SDU level are those that might not be necessary if correct versions of software parts are used for the card level testing. There is also a blue box highlighting those that were in scope of this thesis implementation. All the data load scenarios are also listed per component in the following sections with more detailed description.

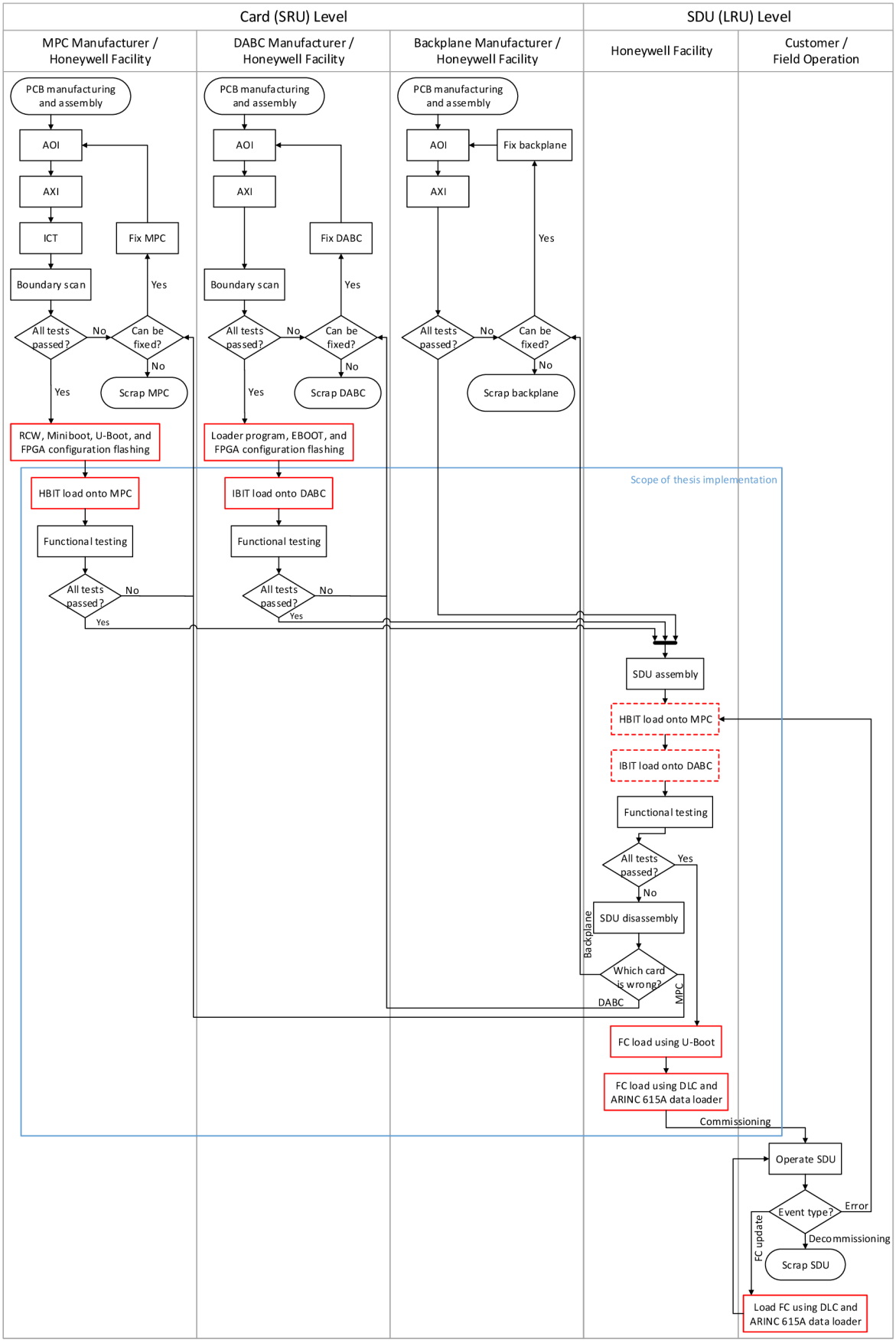

Fig. 5.1: Aspire 400 SDU lifecycle with data loads highlighted

5.1 MPC Data Load Scenarios

The possible and meaningful data load scenarios for MPC are the following:

1. Blank MPC $\longrightarrow$ MPC with RCW, Miniboot, and U-Boot
2. U-Boot $\longrightarrow$ U-Boot
3. U-Boot $\longrightarrow$ HBIT
4. HBIT $\longrightarrow$ HBIT
5. HBIT $\longrightarrow$ flight code
6. Flight code $\longrightarrow$ flight code
7. Flight code $\longrightarrow$ HBIT (and IBIT)

The first scenario has to be performed at the card level, i.e. before the SDU is assembled, since the JTAG connectors need to be accessible in order to flash the MPC. As mentioned earlier, in normal production, this data load scenario is usually covered by the MPC manufacturer. Once U-Boot is present on MPC, more advanced ways of data loading can be used. Nevertheless, it might be necessary to go over this scenario again if the RCW, Miniboot, and / or both U-Boot images get corrupted. In such case, the MPC needs to be flashed again at the card level to ensure its correct functionality. Details of how this scenario is carried out can be found in section 6.3.

The second scenario is not very likely to occur, but in case U-Boot version needs to be changed on an MPC already loaded with working U-Boot, U-Boot offers the capability to reload itself. The way to do that is very similar to the way the next scenario is done and it is mentioned in chapter 7.

The third scenario takes place after the unit is assembled and ready for the box level testing. HBIT (see subsection 4.2.5) is an embedded software designed specifically for testing the SDU's hardware functionality at the box level. How is this data load performed is described in chapter 7. If HBIT gets corrupted, but U-Boot and layers underneath do not, HBIT can be reloaded in the same manner. Also if HBIT needs to be updated to a different version, the same approach is taken.

The fourth scenario might be omitted if the HBIT version loaded in the third scenario is the same as the target version of this scenario. But if it is not, the original HBIT has to be reloaded with the required one. This is done exactly the same way as if there was no HBIT present on MPC.

The fifth scenario is done once the box level testing of the SDU is successfully finished. HBIT is erased and flight code is loaded onto MPC. U-Boot is used to do this yet again, so details about this scenario are also in chapter 7.

The sixth scenario is necessary after the initial load of the flight code onto the SDU in production, as well as if an update of flight code is released and the SDU needs to be updated in the field. U-Boot is only capable to load components based on MPC, but flight code also needs correct version of software on DABC and correct data in the SCM based ORT table. U-Boot cannot ensure this, but the data load controller (DLC) implemented in flight code can. Therefore, flight code is first loaded only onto MPC using U-Boot to

put DLC and its supporting components in place. Once available, the DLC functionality, in cooperation with an external ARINC 615A compliant data loader, is used to load DABC and SCM with the required data. The external data loader can be for example the DataLoader application developed as a part of this thesis. The approach using DLC together with DataLoader is closely described in chapter 9.

The seventh scenario might occur when a hardware error is found on an SDU. In such case it is necessary to identify the error and whether it is possible to fix it. Combination of HBIT and IBIT is the software configuration designated to identify hardware errors, so these software parts are loaded onto the SDU again. The advantage of DLC is that is is able to write to any memory address on MPC, DABC, and / or SCM. Thus it is able to reload HBIT onto MPC and IBIT onto DABC (if the correct version of IBIT is not already present), as well as it would be able to update U-Boot if needed. One setback is that HBIT and flight code cannot coexist on MPC, so by the reload of HBIT, flight code ceases to operate and after the error is identified and the SDU repaired, it needs to be loaded again as described in the fourth and the fifth scenario.

5.2 DABC Data Load Scenarios

The possible and meaningful data load scenarios for DABC are the following:

1. empty DABC $\longrightarrow$ DABC with loader program and EBOOT
2. EBOOT $\longrightarrow$ EBOOT
3. EBOOT $\longrightarrow$ IBIT
4. EBOOT $\longrightarrow$ SwiftBB

Just like on MPC, the first scenario has to be performed at the card level. Even though the necessary JTAG lines are available on the backplane connector, they cannot be accessed at the box level. Again, in normal production, this data load scenario is usually covered by the DABC manufacturer. Once the loader program and EBOOT are present on DABC, more advanced ways of data loading can be used. If the loader program and / or EBOOT get corrupted for any reason, the DABC needs to be flashed again at the card level to ensure its correct functionality. Details of how to perform the flashing can be found in section 6.4.

The second scenario does not happen very often, only when the EBOOT version has to be changed. Then it reloads itself. The process is the same as if it was any other image type and it is described in chapter 8.

The third scenario takes place before the card level functional testing and might be repeated after the SDU is assembled and ready for the box level testing. IBIT (see subsection 4.3.4) can be viewed as a counterpart to HBIT on MPC. It is a software allowing DABC to be commanded to transmit and / or receive as requested. This is used by HBIT to perform all the RF tests on the SDU level.

The fourth scenario is performed as a part of the flight code data load onto MPC, namely the second phase using the data load controller. DLC uses the so called chan-

nel card interface (CCIF) implemented on MPC, which also provides commands to load a software image onto DABC. This interface basically utilizes the same BCL commands used to load any DABC software.

Naturally, any DABC software image including those not mentioned in this text can be loaded either using directly the approach described in chapter 8, or the DABCupgrade application described in the same chapter. This application wraps the raw BCL commands approach.

6 LOADING LOW LEVEL SOFTWARE

In this chapter, the processes of loading the low level software parts onto both MPC and DABC are described. These loads are necessary after the cards are manufactured and blank, or when the low level software parts get damaged for some reason.

The initial flashing (i.e. writing the low level software code into non-volatile memories) is normally done as a part of the acceptance test procedure (ATP) for both MPC and DABC. The approach is quite similar for both cards, with only slight differences.

Apart from the production solutions, some engineering ways to perform the flashing are also mentioned. However, these are not very suitable for the production process, where the goal is to have the procedure as simple as possible, so that the operator can simply follow a checklist step by step and does not need to have any deeper knowledge about the device. The engineering solutions are usually more complicated than that and there is a risk they could cause more damage than good if used improperly.

The flashing is done using the so called in-system programming (ISP), which is a process utilizing a special JTAG interface, which has to be present on the targeted chip. Therefore the process has to be supported on the hardware level and this has to be kept in mind while selecting the hardware parts for the final solution. Luckily, many chips today provide the JTAG interface, since it is a de facto industry standard for the low level testing.

There are more types of JTAG interfaces, however, the most known and the one used both on MPC and DABC is defined by the IEEE 1149.1 standard (see [24]). The hardware principle of the JTAG interface, boundary scan, and ISP are described in the following sections.

6.1 Boundary Scan

Boundary scan test is a structural test of the component utilizing special circuitry added to the chips and the printed circuit board (PCB). It is an alternative to testing using the so called bed of nails fixture or flying probe. The bad of nails fixture is custom made fixture for the tested component that has pins pointing exactly in places where test points are on the tested board. When the component is put into this fixture, contacts are made in these places and logical values (voltage levels) can be injected or read out by the fixture's pins. Using this approach, defined parts of the circuitry can be tested to see if they are operating as expected.

The bad of nails fixture naturally has its benefits and drawbacks. The benefit is that the testing is rather fast and can be parallelized quite well. Drawbacks are that the fixture is expensive to make, it can only serve the one PCB and in case its design is changed, the fixture needs to be changed as well. And since there are physical contacts between the fixture and the tested board, they can wear-off. Plus there is a risk of damaging either the fixture or the tested board with careless manipulation when putting the board in place or taking it out. In general, making the bed of nails fixture is more beneficial for testing

larger amounts of boards, because of the higher speed, and the high price is spread across more units.

The flying probe improves some of the mentioned drawbacks of the bed of nails. It is a moving arm equipped with pin points, programmed to get over to the correct position on the board and make an on-demand connection. It is obvious that this solution is more flexible in case any changes need to be made. On the other hand, the testing is slower, and the drawback of the need for having a physical connection stays. This is problematic with modern day PCBs, where components get smaller and smaller and their density on the board higher. Especially with multilayer PCBs, it can get extremely hard to design them to have all the required test points reachable.

An alternative for the structural testing based on connections made via test points is the boundary scan. Its principle can be found in Figure 6.1. There are three chips, each having 5 inputs and 5 outputs and some internal logic, and they are connected in a series. The circuitry providing the boundary scan functionality is highlighted in red.

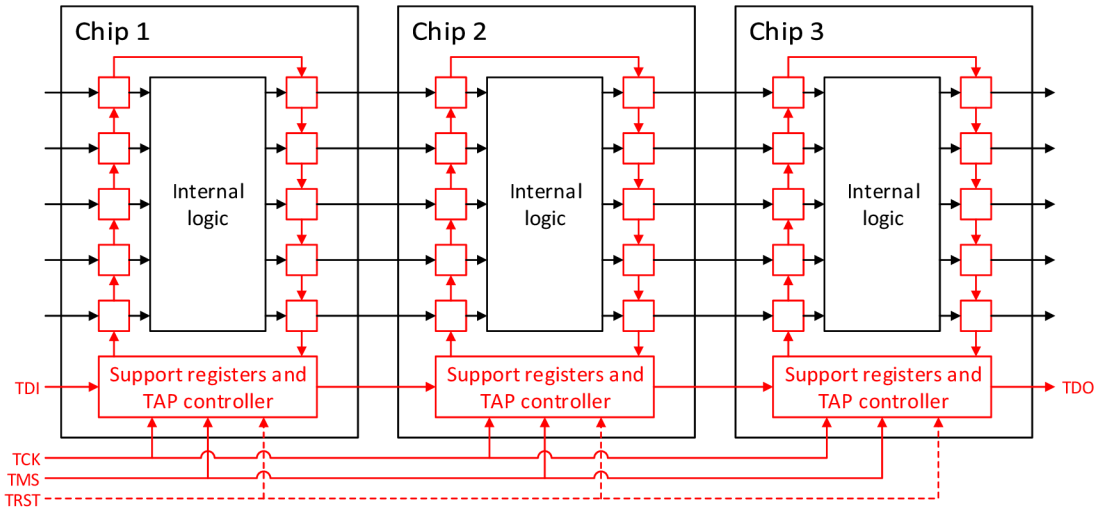

Fig. 6.1: Principle of boundary scan testing and in-system programming

The basic idea behind the boundary scan is quite simple. In between every input and output pin of the tested chip, a special logic called boundary scan cell is inserted. There are two multiplexers and two D flip-flops in every boundary scan cell, providing this cell with 4 modes of operation:

1. Normal
2. Capture
3. Update
4. Shift

In normal mode, the boundary scan cell only passes the data between the pin and the internal logic of the chip as if it was not even there. Capture mode samples the normal input data into the first register. Update mode puts the test input data on the normal

output (through the registers). And shift mode sends the bit on the test input to the test output, which is test input of the next boundary scan cell.

As can be seen in Figure 6.1, the boundary scan cells are connected in a series, making up a so called Boundary Scan Register. The shift operation allows any bit sequence to be shifted into this register. This way, the cells on the inputs of the internal logic can be filled with the desired bits (e.g. an instruction) and then these bits can be sent in. They can also be captured on the other side of the internal logic circuitry, be shifted out and compared to the expected outcome.

Not only the internal logic of the chips having boundary scan support can be tested. When the boundary scan logic is connected in a daisy-chain like it is in the figure, the wiring between the components can be tested as well, when the test vector is inserted on the output pins of one chip and captured on input pins of another chip. Even if some logic is between components supporting boundary scan, it can be tested (at least to some extent).

There are 4 mandatory signals and 1 optional signal defined by the IEEE 1149.1 standard as the interface for the boundary scan. This interface is called Test Access Port (TAP) and the signals are:

- Test Data Input (TDI)
- Test Data Output (TDO)
- Test Clock (TCK)
- Test Mode Select (TMS)
- Test Reset (TRST)

All of them except TDO are inputs. TRST is optional and is not used neither on MPC nor DABC. The behavior of the boundary scan cells is synchronized using the TCK signal and controlled using the TMS signal and a 16 state finite state machine called the TAP Controller.

The support registers in the red box on each chip in Figure 6.1 are at least the Instruction Register and Bypass Register, but usually there are more. Based on the TMS and the TAP Controller state, TDI is directed to one of the registers. The Instruction Register holds the current instruction for the test. Based on the instruction, data sent to TDO are selected. For example if BYPASS is the active instruction, bits from TDI are simply shifted through the Bypass Register to TDO.

TRST is optional, and usually not implemented, because the test can be restarted synchronously by the TAP Controller in at most 5 cycles of TCK.

The benefits of boundary scan are that circuits having the necessary hardware support can be tested quite thoroughly, with focus on smaller blocks at a time. Plus there is no need for physical access to any pins or test points, which can be a great advantage in dense, multilayer PCBs. Only the TAP interface is necessary.

On the other hand, the boundary scan logic naturally takes some place on the chip and makes a bit more expensive. But since it is mostly serial the hardware overhead is not so high. However, the serial nature of boundary scan makes it a bit slow, especially

for long test chains, where it takes many clock cycles to prepare the test data, read it out, etc. Also the maximum TCK rate for the chip is usually much lower than in normal operation and the lowest one of all the chips in the chain has to be used. Normal TCK rates are in tens of MHz.

6.2 In-system Programming

The in-system programming (ISP), sometimes also called in-circuit serial programming (ICSP), is a method of writing data into memory (specifically non-volatile memory) used usually after the PCB assembly, when the memory is blank. Some code for the processors or FPGA configurations have to be stored in the memory for the functional components to be able to operate usefully.

ISP utilizes the boundary scan circuitry described in the previous section to achieve this. It basically emulates the memory read and write operations on the memory bus the way they would be performed normally, i.e. it addresses the memory cell, prepares the data, and enables the write operation. Or it can read out the data to perform a check that the data are correct.

Both MPC's and DABC's low level software parts are programmed using ISP.

6.3 Flashing MPC

On MPC, the following software parts are considered to be low level:

- FPGA configuration
- RCW
- Miniboot
- U-Boot

FPGA configuration is loaded onto the FPGA and the other parts are loaded onto both processors, i.e. RCW is stored into the processor's EEPROM and Miniboot and U-Boot into its NOR flash. Each processor has 16 pin TAP interface and the FPGA has a 10 pin one.

External hardware and software that is capable to drive the TAP signals correctly is required. For example CodeWarrior TAP kit and CodeWarrior IDE can be used for the processors. But these tools are more suitable for development and debugging, rather than production. For production, the tools listed in the next section for DABC flashing are better. FPGA can be programmed for example using a Microsemi FlashPro4 or FlashPro5 JTAG hardware programmer.

Once the software parts named above are present, it is possible to use the data load methods described in chapter 7. It is of course still possible to load other software parts the way presented above, too. But it is not usual, nor convenient, since this method is quite slow, it is necessary to have the JTAG controller, and once the SDU is assembled, the JTAG connectors are not accessible. This applies for DABC as well.

6.4 Flashing DABC

On DABC, the following software parts are considered to be low level:

- FPGA configuration
- loader program
- EBOOT

Like on MPC, FPGA configuration is loaded onto the FPGA and the other parts are loaded onto protocol processors on both channels, i.e. stored in their NOR flash. Since DABC has the TAP interface for all components (except RF) integrated into the backplane connector, a special test jig board with more standard JTAG pin header connectors is used. This board is connected to the DABC via the backplane connector and the JTAG connectors are wired to the defined pins.

For example JTAG's JT 37x7/TSI hardware controller and JTAG ProVision software can be used to perform the flashing. Once the software parts named above are present, it is possible to use the data load methods described in chapter 8 or chapter 9.

7 MPC DATA LOAD USING U-BOOT

In this chapter, the process of loading data onto MPC card using the U-Boot console is described. As was mentioned in subsection 4.2.4, U-Boot provides, among other things, an interactive console controlled via RS-232 interface. A program called SerialParser was developed to automate communication with U-Boot over RS-232 as a part of this thesis.

Using appropriate commands, U-Boot can perform writing anywhere in the memory space of the processor it is running on, including for example NOR flash. This is used when loading both the primary processor and the secondary processor on MPC. It is implied that U-Boot is also capable to overwrite itself, which can be used with advantage when U-Boot update is necessary. The current version of U-Boot is running from RAM, as was described in subsection 4.2.4, and after the processor is restarted, the new version is loaded from the NOR flash to RAM and started.

Apart from commanding the processor using U-Boot, some preconditions have to be met on the initiator side for a successful data load. The initiator has to be able to transfer the loaded data to the processor's RAM upon a request from U-Boot. TFTP is used in most cases, which means the initiator has provide TFTP server that is able to serve the U-Boot requests. It is also necessary to have the correct Ethernet topology for TFTP to work. Alternatives to TFTP also exist, for example YMODEM protocol over RS-232 can be used, in which case initiator has to support it.

In section 7.1, requirements for SerialParser and its implementation are laid down. In section 7.2, the design of the program is presented. In section 7.3, the implementation itself is described. And in section 7.4, the procedure for verification and validation of the program is described.

7.1 Requirements

One of the goals of the SerialParser development was to create a program that is reusable for automation of any communication over RS-232, not only U-Boot. Another was to provide not only the features necessary for this scenario, but features that were inspired by RS-232 communication scenarios on other projects, or features that were considered nice-to-have, too.

The requirements for the SerialParser were gathered with these goals in mind. Prior to design and implementation, the following requirements were set:

1. SerialParser shall monitor given COM port for a predefined pattern (RX pattern, e.g. login prompt) and once detected, reply with a predefined response (TX pattern, e.g. username)
2. SerialParser shall provide an option to define an empty TX pattern
3. SerialParser shall control a number of replies for each pattern, supporting infinite processing (e.g. one RX pattern can be processed only once, while another RX pattern can be processed every time it comes)

4. SerialParser shall provide an option to define multiple TX patterns assigned to the same RX pattern (in this case it would be assumed the first pattern pair would only be processed a finite number of times)
5. SerialParser shall provide an option to exit or continue on each RX pattern
6. SerialParser shall display the communication (both RX and TX patterns) to a GUI element (window)
7. SerialParser shall provide an option to display or hide the window when started
8. SerialParser shall provide an option for the user to enter commands (TX patterns) from the window (it is expected the RX pattern processing will be on hold in this case)
9. SerialParser shall provide an option to exit on a command from the user
10. SerialParser shall implement a command to initiate the user control as a security precaution (e.g. a predefined key has to be pressed prior to allowing the user to enter commands)
11. SerialParser shall store the entire communication history to a log file (including the user-entered TX patterns)
12. SerialParser shall provide control over the type of line endings (carriage return and line feed, or line feed only)
13. SerialParser shall provide error handling (return appropriate error codes and messages) and timeouts (to exit in case the exit pattern is not found)

During the design and implementation, these requirements were followed. Some of them were a bit vague. In such cases, the decision on their final implementation was taken in the development process. Some of the requirement were covered in a broader way, while some were found to be unnecessarily demanding and were reconsidered.

7.2 Design

The design of the SerialParser application is described in this section. The description is split into multiple functional parts. The design covers the requirements defined in the previous section and the implementation is based on this design.

The basic flow of the SerialParser application is the following:

1. Load settings (if not passed as parameters)
2. Open log file
3. Load patterns
4. Open COM port
5. Wait for input data
6. Check for RX patterns in the input data
7. Reply with TX pattern if RX pattern is found
8. Stop on exit pattern or timeout, otherwise go to step 5
9. Perform cleanup (close COM port, close log file, deallocate dynamic memory)

SerialParser runs in a console window. This covers requirement 6.

7.2.1 Settings File

SerialParser requires a couple parameters which make up its settings. These parameters can be passed to it in form of an INI file. INI file should contain predefined sections and tags with values, based on which SerialParser sets itself up. These parameters cover the COM port settings (port number, baud rate, data bits, parity, and stop bits), logging settings (if logging is allowed both to console and a log file, where the log file should be stored), console window visibility setting (this covers requirement 7), timeout setting (global timeout for the whole application), and path to XML file with patterns.

7.2.2 Patterns File

XML format was chosen to describe patterns. The patterns XML file has to contain a `patterns` root element, which can have any number of `pattern` child elements. Each `pattern` element can have the following attributes:

- `repeat`
- `repeat_every`
- `skip_first`
- `exit`
- `timeout`

These attributes are not mandatory and if they are not defined, default values defined below are used.

First attribute sets the number of times reply is sent when the given RX patten is found. It can also be set to `inf`, which means this RX pattern will be replied infinitely (default value). This covers requirement 3.

Second attribute makes the parser reply only to some occurrences of the given RX pattern, for example every second, third, or fifth occurrence. By default, every RX pattern is responded (value is 1).

Third attribute causes the first n occurrences of RX pattern to be ignored, where n is the number defined by the parameter. By default, no occurrences are ignored (value is 0).

Fourth attribute can be set to `true` or `false`. If it is `true`, SerialParser stops execution if the given RX pattern is found (it replies first). By default, SerialParser does not exit on patterns (value is `false`). This covers requirement 5.

Fifth attribute sets the timespan in which the RX pattern has to be found after its parent RX pattern was found. This is explained in greater detail further in this subsection. By default, no timeout is set (value is 0).

Apart from these attributes, `pattern` element has to contain `rx` child element and may contain `tx` child element. These elements contain the RX and TX pattern strings. If `tx` element is not defined or it is empty, no reply is sent by SerialParser when the corresponding RX pattern is found. This covers requirement 2. In case `rx` element is empty, SerialParser gives warning when parsing the patterns and this pattern is ignored.

To allow user full control over the pattern strings, they are parsed as if they were C strings, i.e. escape sequences defined by two characters (e.g. `'\'` and `'n'`) are replaced by real escape sequences (`'\n'` for the given example). Since user has control over each and every character in the pattern, requirement 12 is covered.

SerialParser allows for multiple patterns to have the same RX pattern. Replying to such patterns is compliant with their setup given by the aforementioned attributes. It is possible for different TX patterns to be replied to one RX pattern immediately after each other. In such case, the order of replies is not defined. It is upon user to control this behavior and decide whether it is desired or set the attributes to the patterns so that they never overlap. This covers requirement 4.

`pattern` element can also have another `pattern` element as a child. If it does and its RX pattern is found, from that moment on, SerialParser starts looking only for the RX pattern of the child `pattern` element. This allows for the patterns to be defined recursively and timeouts can be used. As was mentioned, each `pattern` element can have `timeout` attribute, which sets in how many seconds this RX pattern has to found after its parent RX pattern. If it is not found within this time, SerialParser exits. Timeouts can be used for any pattern, which is defined deeper than in the `patterns` root element. They can prevent SerialParser from getting stuck if the expected flow of patterns is broken. Together with the global timeout, which is started when SerialParser starts waiting for the input data, this partially covers requirement 13.

7.2.3 Call Options

The usage of SerialParser on Aspire 400 project is realized via the TestStand application from National Instruments, which calls SerialParser as a dynamic link library (DLL). SerialParser DLL provides two entry point functions with defined interfaces.

First requires only one input parameter, which is the path (absolute or relative) to the INI file with settings. Second does not require the INI file, but it takes all the settings as parameters. The latter option makes it easier, if the settings for the SerialParser are created dynamically.

SerialParser can be also called from console as a standard executable. This case is basically the same as when calling the first DLL function. One parameter, the path to the INI file, is expected.

7.2.4 Communication Processing

The incoming input data from the COM port are read character by character and RX patterns are searched for with prefix matching. This means that if the current input character matches some RX pattern's first character, an instance of possible RX pattern match is created. When the next input character comes, it is checked whether it matches the second character of RX pattern. If so, next input character is checked again the same way. If not, the instance of possible match is dropped. This is done for every possible match instance with every input character.

If all characters in the RX pattern are matched, the pattern is processed based on its attributes. If this occurrence is supposed to be replied to, corresponding TX pattern is written to the COM port. And if the pattern is an exit pattern, the processing is stopped and SerialParser exits. This covers requirement 1.

7.2.5 User Control

User can take control over the communication by pressing the F4 key in the SerialParser console window. This covers requirement 10. Processing of incoming input data is paused in such case, which could theoretically lead to data loss if the input data buffer overflows. However, the assumption is that user only wants to take control when the device on the other side of the COM port is waiting for some input, i.e. it is not transmitting.

When user has control, he can send any message, with one exception, to the COM port by typing it and pressing Enter. This covers requirement 8. The message is sent including the newline character. The exception is when user types the predefined word (`"exitsp"`) to exit the application. In this case, SerialParser stops execution and exits, rather than sending this string to the COM port. This cover requirement 9.

7.2.6 Logging and Error Handling

Logging can be turned on and off by a setting. When on, info messages about the SerialParser initialization and found RX patterns and sent replies are printed to the console window, as well as the log file, if it is allowed. Every info message starts with a timestamp. If logging is turned of, only input data from the COM port are printed. This covers requirement 11.

As far as error handling goes, there are not many possible ways how to do this for the communication over COM port itself. Possible problems during SerialParser initialization are of course checked and reported. However, when waiting for the input data, the only way to check that input is correct is using the timeouts. If the expected input is not parsed in the given time, it is derived that an error occurred. This covers requirement 13.

It is definitely better to give enough margin to the timeouts, since there is no control over the device on the other side of the COM port and therefore the communication is not exactly precise time-wise.

SerialParser returns error code and if called from DLL also the corresponding error message. Warnings are only logged.

7.3 Implementation

The SerialParser was primarily designed to be used on ATE during production testing. Therefore Windows are the targeted operation system, and as was mentioned in section 4.4, the TestStand application from National Instrument is used at the top level to run the production tests. Therefore LabWindows/CVI IDE (version 2015) by the same developer was picked for the implementation of the SerialParser.

This IDE contains an ANSI C compiler capable of compiling the source code into either executable (*.exe* file) or dynamic-link library (*.dll* file). It also provides libraries, some of which were utilized in the SerialParser, namely the RS-232 library, the INI file library, and the CVI XML library.

The RS-232 library contains functions to open and close COM port, as well as read data from it or write data to it.

The INI file library provides functions to parse an INI file. It can load the file into an internal structure and then parse individual sections and their tag, value pairs. It is used to get the settings values stored in the INI file.

The CVI XML library provides functions to parse a standard XML file. In SerialParser, it is used to parse the XML file with patterns, which are then stored in an internal structure in memory.

Otherwise the implementation is very straightforward, based on the described design. The DLL entry point functions are called `StartParser` and `StartParserIni`. First expects the settings to be passed as parameters, the latter expects path to an INI file, from which it loads the settings.

When SerialParser is started using the normal executable, `main` function is the entry point as in any C program. Since it expects the same input parameter as the second DLL entry point function, the `StartParserIni` DLL function is just a wrapper that calls `main` the main function.

Custom structures are used to store various data and a few macros were defined to make logging and error handling easier. After getting the settings from the INI file (if SerialParser is not called from DLL with parameters given), XML patterns are loaded and COM port is opened.

Then the program loops in the main loop, where it awaits input data, user action, and / or timeout. As was already mentioned, input data are processed character by character. This is done in another loop which is active until there are data available in the COM port input buffer or a predefined number of characters was processed. The second condition is to keep SerialParser interactive, because if data were coming continuously all the time, SerialParser would be processing them and would not check for user actions and / or timeouts. Of course it would be possible to resolve this by using multiple threads and callbacks or signals. But since the application is not time critical, this solution seems overly complicated and error prone. Actually, SerialParser is suspended in every iteration of the main loop by putting it to sleep for a short time. This is to prevent it from taking too much CPU time while waiting for the input data.

In the first version, the input data were not processed character by character, but as strings read out from the COM port input buffer. However, this solution was found to be problematic, because when testing with a real SDU, it was discovered that it generates a lot of null characters (`'\0'`). Null characters are also string terminators in C, therefore when the input COM port buffer contained useful data, then some null characters, and then useful data again, when it was read out as a string, only the first part of the useful

data was obtained and the second part was lost. This problem was not discovered when testing only in simulated environment. When processing the input one character at a time, null characters are ignored.

The program stops execution of the main loop once an exit pattern is found, user takes control and sends the exit command, or either the global or pattern timeout runs out. Timeouts are checked by comparing difference between current timestamp and timestamp taken when the main loop was started (for global timeout) and timestamp taken when the last RX pattern was found (for pattern timeout).

When the program is out of the main loop, it deallocates all allocated memory, releases all other resources (log file, COM port), and returns with appropriate error code (0 for success). If called from DLL, it also copies the error message to an output parameter. The cleanup phase is done always, even if an error occurs during the initialization phase and the program does not even start the main loop.

7.4 Verification and Validation

For development, an environment simulating the communicating device was used. This environment consisted of a COM port emulator and a simple Python script mimicking the target device. The open-source Null-modem emulator (com0com) was used for the COM port emulation (see [25]). This tool can create a pair of virtual COM ports linked to each other. SerialParser was then commanded to connect to one of these COM ports and the Python script to the other.

The Python script generates defined output, which server as an input for the SerialParser. The XML patterns are made up to test all features provided by the SerialParser (e.g. `repeat`, `repeat_every`, `skip_first`, and `exit` attributes, timeouts, etc.). The log from SerialParser is parsed by the Python script to check the SerialParser behaved as expected.

The functionality was also validated by using the SerialParser to load HBIT onto both MPC processors. In Figure 7.1, a sequence diagram with expected RX patterns coming from the MPC processor and the TX pattern replies from the SerialParser is shown. The sequence is the same for both the primary and the secondary processor. The patterns are numbered and a short explanation for each of them can be found in the list following the figure. They are defined recursively, i.e. each RX pattern is looked for after the previous RX pattern is found. Each pattern also has a defined timeout, therefore when the communication does not go down as expected, the SerialParser times out and reports on which pattern it happened, i.e. which RX pattern was not found. This helps user with debugging of the problem.

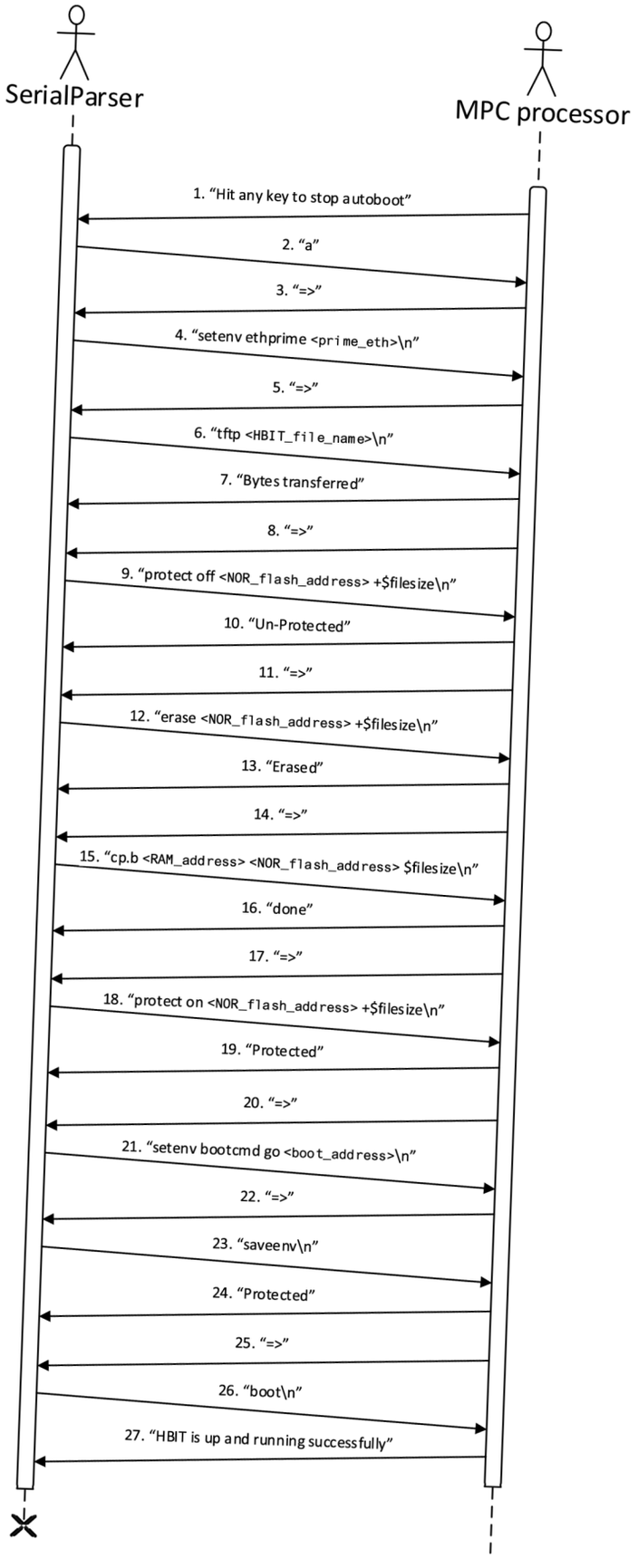

Fig. 7.1: Sequence diagram of HBIT data load onto MPC processor

1. U-Boot boot sequence is waiting for an interruption
2. Interrupt U-Boot boot sequence by sending an `'a'` key
3. U-Boot prompt is waiting for a command
4. Command to set the primary Ethernet interface is issued (to make sure TFTP works properly)
5. U-Boot prompt is waiting for a command
6. TFTP command to download the HBIT binary into MPC processor's RAM is issued
7. Check the TFTP download was performed (string `"Bytes transferred"` appears as a part of the output upon a successful operation)
8. U-Boot prompt is waiting for a command
9. Command to un-protect a block of NOR flash memory where the HBIT binary shall be copied is issued
10. Check the memory was un-protected (string `"Un-Protected"` appears as a part of the output upon a successful operation)
11. U-Boot prompt is waiting for a command
12. Command to erase the block of the NOR flash memory is issued
13. Check the memory was erased (string `"Erased"` appears as a part of the output upon a successful operation)
14. U-Boot prompt is waiting for a command
15. Command to copy the HBIT binary from RAM to NOR flash memory is issued
16. Check the binary was copied (string `"done"` appears as a part of the output upon a successful operation)
17. U-Boot prompt is waiting for a command
18. Command to protect the block of NOR flash memory is issued
19. Check the memory protection was turned on (string `"Protected"` appears as a part of the output upon a successful operation)
20. U-Boot prompt is waiting for a command
21. Command to set the new boot start address is issued
22. U-Boot prompt is waiting for a command
23. Command to save the environmental variables is issued
24. Check the environmental variables were saved, i.e. the NOR flash memory block where the environmental variables are stored was un-protected, erased, written to, and protected again (string `"Protected"` appears as a part of the output upon a successful operation, but this time the SerialParser has to look for the second occurrence of this string, since it first occurs in the string `"Un-Protected"`)
25. U-Boot prompt is waiting for a command
26. Boot command is issued for U-Boot to run the booting sequence with the new setup
27. Check that HBIT was booted (string `"HBIT is up and running successfully"` appears as a part of the output upon success)

8 DABC DATA LOAD

As already mentioned in section 4.3, DABC's application is selected using the loader program. The loader program is loaded onto DABC via a JTAG programmer as described in chapter 6, namely section 6.4.

To load other application images, DABC needs to run the EBOOT application, which allows the usage of a set of BCL commands that write into the protocol processor's NOR flash. Therefore the first step of the data load process is to reboot the DABC's channel that is about to be loaded into EBOOT. There are two hardware ways and one software way to achieve reboot into EBOOT.

First hardware way is to turn off and back on the power supply. This restarts both channels of DABC and each starts the loader program, in which EBOOT can be selected over the control port. Second hardware way is to use a restart discrete signal. When the discrete is turned on, the channel is stopped and when the discrete is turned off again, the channel starts the loader program. Each channel has its own restart discrete. Inside an SDU, both the power supply and restart discretes of DABC are controlled by MPC. The software way to reboot DABC is using BCL command which tells DABC to reboot itself into the specified application.

Once in EBOOT, writing into the NOR flash can be performed. First, BCL command asking to unlock the NOR flash for writing is sent. Then BCL commands to program individual blocks of memory are issued. These commands contain the address of the target block, length of data, computed CRC for a security check, number of allowed repetitions in case of an error, and the data itself. BCL responses are generated for these commands. First when the command is accepted (or declined with a reason) and then once the writing into memory is finished (or fails, in which case error code is returned). Parallel writing into multiple memory blocks can be utilized to speed up the process. This is especially useful when using fast transport link to transfer the data (e.g. Ethernet).

All applications are stored in the protocol processor's NOR flash, since it is the only non-volatile memory on DABC. Even code for the DSPs (if the application uses them), is stored there. This code is copied to the DSP's RAM (SDRAM or SRAM) during the application startup.

Like with the data load on MPC using U-Boot, the data can be loaded via COM port (RS-232), or over Ethernet (TCP or UDP). The same advantages and disadvantages apply for both these possibilities, as on MPC (i.e. using RS-232 is slower but more reliable in extreme conditions).

Besides using the BCL commands directly, there is a Honeywell proprietary application called DABCupgrade, which can be used for the data load as well. This application wraps the communication described above and provides a simple command line interface. Some of the possible command line arguments are:

- `<path_to_image_file>`, which specifies absolute or relative path to the *.img* file
- `-image <name>`, where `<name>` specifies the image name to be used (useful in case there are multiple images packed within one *.img* file)

- `-menu <option>`, where `<option>` specifies the menu option to be used (some applications contain code for multiple components and the menu option picks whether all the code or only parts of it should be loaded)
- `-confirm`, which tells DABCupgrade to skip the confirmation prompt and the final screen (useful for full automation)
- `-com<number>`, where `<number>` specifies the COM port number over which data load should be performed
- `-tcpip <ip>:<port>`, which tells DABCupgrade to perform data load over Ethernet (TCP) using the specified `<ip>` and `<port>`
- `-udp <ip>:<port>`, which tells DABCupgrade to perform data load over Ethernet (UDP) using the specified `<ip>` and `<port>`

Using a proper combination of these arguments can simplify the DABC data load process to one call of DABCupgrade. It also returns code based on the result of the operation. This code can be checked and it gives more information in case of an error.

A wizard-like window application called DABCwinupgrade also exists. It provides the same functionality as the command line version, but uses text input fields, select lists, buttons, etc., so it is not that suitable for automated use.

9 ARINC 615A COMPLIANT DATA LOAD

This chapter describes the process of data load compliant with ARINC 615A standard (see section 3.4) and its implementation that was done as a part of this thesis. As was mentioned in subsection 4.2.6 and section 5.1, flight code software for Aspire 400 SDUs is going to implement the functionality for data loading according to ARINC 615A standard. But to be able to utilize this functionality, a data loader compliant with this standard has to be used.

There are some ready made commercial data loaders available on the market. However, these tools are quite expensive. And the testing department at Honeywell needs such a tool for almost every test station they build. Therefore it was decided that it would be beneficial to come up with a Honeywell in-house solution, i.e. a data loader, or data load function (DLF) to be more precise, that is capable of cooperation with the data load functionality implemented in flight code. This solution should be following ARINC 615A standard, even though not all the functionality is necessary at the time being, hence it can be a bit more lightweight than the offered commercial solutions.

The developed application was named simply DataLoader. Its goal is to provide user with means to perform data load of ARINC 665 compliant packages onto ARINC 615A compliant devices. In case of Aspire 400, the data package usually contains the flight code FIT image and the ORT table with customer specific data.

Using the terminology of the standard, SDU is the target hardware, ATE used in production is the data loader (neither portable nor airborne), and the DataLoader application is DLF.

9.1 Requirements

The DataLoader application has been developed as an engineering solution so far, therefore no formal requirements for it were made. The scope of the application is loosely defined by ARINC 615A standard, although not all the properties discussed in this standard apply nor for the DataLoader, nor the ATE. For example the physical parameters and the transport media types defined by ARINC 615A for the data loader are not considered.

As in case of the SerialParser application, the primary goal of the DataLoader application is for it to be usable on the Aspire 400 ATE by calls from the TestStand application. However, possibility of reusing it on other projects with no or only minor modifications was kept in mind during the design, as this was a requirement. In order to achieve greater usability, the application provides not only an interface for TestStand, but also a console interface and a simple GUI.

Some of the formal requirements from the Honeywell's requirements database for the data load functionality inside of the Aspire 400 flight code were also considered during the design and implementation of the DataLoader to make these tools coherent.

9.2 Design

The design of the DataLoader application is described in this section. The description is divided into several main areas that were needed to be considered.

The typical flow of the DataLoader application is the following:

1. Search LAN for available SDUs (if specific IP address is not supplied)
2. Get SDU(s) information
3. Initiate the uploading operation as defined by ARINC 615A with the selected SDU and software parts for loading
4. Monitor the ongoing uploading operation (abort is supported)
5. Report the result of the uploading operation upon finish

9.2.1 SDU Side

As was already stated, ARINC 615A data loading functionality is implemented in flight code on SDU side. Unfortunately, due to changes made to the original schedule, version of Aspire 400 flight code with this functionality is not yet released. Only the design and partial functionality of some components are currently at hand. The DataLoader implementation that was done as a part of this thesis is based on the available documentation, requirements, the fact that the interface should be standardized, and on other older flight code implementations for different Honeywell projects that also deal with data loading according to ARINC 615A. It cannot be ruled out that some changes will need to be made to the DataLoader to work with Aspire 400 flight code correctly. However, the core design should stay the same, only minor tweaks are expected.

According to the flight code design, data load shall be implemented by the so called data load controller (DLC) module. This module implements the external interface of ARINC 615A defined services and cooperates with other flight code modules to perform the data load. A very brief description of the SDU internal functionality follows. However, it cannot go into much detail in order to keep sensitive information undisclosed.

A so called load installer module, which shall actually perform the writing of loaded data into the component's non-volatile memory, is going to be implemented for each of the loadable components. This module will provide an interface utilized by DLC. Since the data load process is driven by MPC and other components need to be loaded as well, flight code needs to provide means of communication with these components. For example for DABC, there is the so called channel card interface (CCIF), which translates all commands for DABC into BCL and vice versa, including the commands the load installer module shall use. For SCM, in which primary ORT tables with customer data are stored, its load installer module shall utilize the so called ORT controller. And DLC is also going to use the SIS interface on MPC to update the software version numbers in SIS EEPROM after a successful data load.

Apart from the installer modules, DLC is also going to cooperate with SDU controller, through which it shall check that data load is allowed (SDU is in a data load mode) and

through which it shall reboot the SDU during the data load process, too. All mentioned inter-modular communications are going to use the message event service (MES).

For the ARINC 615A interface, DLC shall run a separate thread that shall control a socket on port 1001, where it is going to wait for FIND protocol requests, which are described in the next subsection. If any request comes, this thread shall issue an answer. For the rest of the data load process, DLC needs to act as both TFTP server and TFTP client. Only upload of data onto the SDU is planned to be supported at this time. Download operation, also defined by ARINC 615A, but only as optional, shall not be implemented.

DLC is also going to be responsible for receiving the ARINC 665 data package, unpacking it, verifying it, and distributing the data to appropriate load installer modules. From that moment on, the loading itself shall be performed by load installer modules and DLC shall only send the periodic upload information status to the external data loader (e.g. the DataLoader application). Once the load installer modules shall be done, DLC is going to validate the loaded software parts and inform the external data loader of the result.

9.2.2 FIND Protocol

Find Identification of Network Devices (FIND) is a protocol defined by ARINC 615A and serves to discover ARINC 615A compliant devices on a local network. It was already described in section 3.4, but the key points are summarized here as well. It uses dedicated port 1001 and UDP datagrams to exchange the protocol packets. There are two types of packets. Information request (IRQ) broadcasted by the data loader, and information answer (IAN). IAN contains some information that identifies the device, namely the target hardware identifier (THW_ID), target type name, target position, literal name, and manufacturer code. And of course the IP address is known from the packet, too. After broadcasting IRQ, the data loader gathers all IANs that come within 3 seconds.

DataLoader uses FIND operation to discover SDUs available for data load. It first gets a list of Ethernet adapters of the computer it is running on. Then it sends the IRQ packet to the broadcast address of each of the adapters. A callback function processes any IAN packet that comes within the defined timespan. The packet is parsed and the properties of the device are stored in an SDU object. This way a list of SDU objects ready for data load is created.

There is another supported way of initiating FIND operation in the DataLoader, when an IP address is passed as a parameter. In such case, the IRQ packets are not broadcasted onto all local networks, but only one is sent (using unicast) to the given address. This approach is not defined by the ARINC 615A standard, yet it is quite useful during the production testing, when IP addresses of the SDUs connected to ATE should be known. IRQ packet is sent to the IP address where an SDU should reside and IAN response gives assurance that the SDU is indeed in place, ready for other operations, and the SDU object with its information is created.

9.2.3 TFTP

Apart from the FIND protocol, all ARINC 615A data load functionality is implemented using TFTP services. TFTP (at least the core functionality) is specified by RFC 1350 (see [26]). Other RFCs enhancing the original functionality are linked from within this document.

The data load process is called upload operation by the ARINC 615A standard and it is based upon exchange of files defined both by the ARINC 615A and ARINC 665. The TFTP read and write requests (RRQs and WRQs) are issued for the files in defined order, and they are transferred using data packets after the request is acknowledged by an ACK packet, or the operation is terminated by an error packet. Both transaction sides (i.e. the DataLoader and flight code) have to support the role of both TFTP client and TFTP server. Their current role depends on the state of the data load process.

A minimalistic data load sequence is shown in the sequence diagram in Figure 9.1. It captures the packets exchanged between the data loader (e.g. the DataLoader application) and the target hardware (e.g. the DLC implemented in SDU's flight code). The packets are numbered and briefly explained in the following list:

1. FIND information request is issued by the DataLoader to the SDU (either by broadcast or unicast)
2. FIND information answer containing SDU parameters is sent by SDU's DLC
3. TFTP read request for load uploading initialization (`.LUI`) file is issued by the DataLoader (TFTP client at this moment)
4. The request is acknowledged by DLC (TFTP server at this moment)
5. TFTP transfer of the load uploading initialization file from DLC to the DataLoader is done
6. TFTP write request for load uploading status (`.LUS`) file is issued by DLC (TFTP client at this moment)
7. The request is acknowledged by the DataLoader (TFTP server at this moment)
8. TFTP transfer of the load uploading status file from DLC to the DataLoader is done and the DataLoader checks the status provided within the file
9. TFTP write request for load uploading request (`.LUR`) file is issued by the DataLoader (TFTP client at this moment)
10. The request is acknowledged by DLC (TFTP server at this moment)
11. TFTP transfer of the load uploading request file from the DataLoader to DLC is done
12. TFTP write request for load uploading status (`.LUS`) file is issued by DLC (TFTP client at this moment)
13. The request is acknowledged by the DataLoader (TFTP server at this moment)
14. TFTP transfer of the load uploading status file from DLC to the DataLoader is done and the DataLoader checks the status provided within the file
15. TFTP read request for load upload header (`.LUH`) file is issued by DLC (TFTP client at this moment)

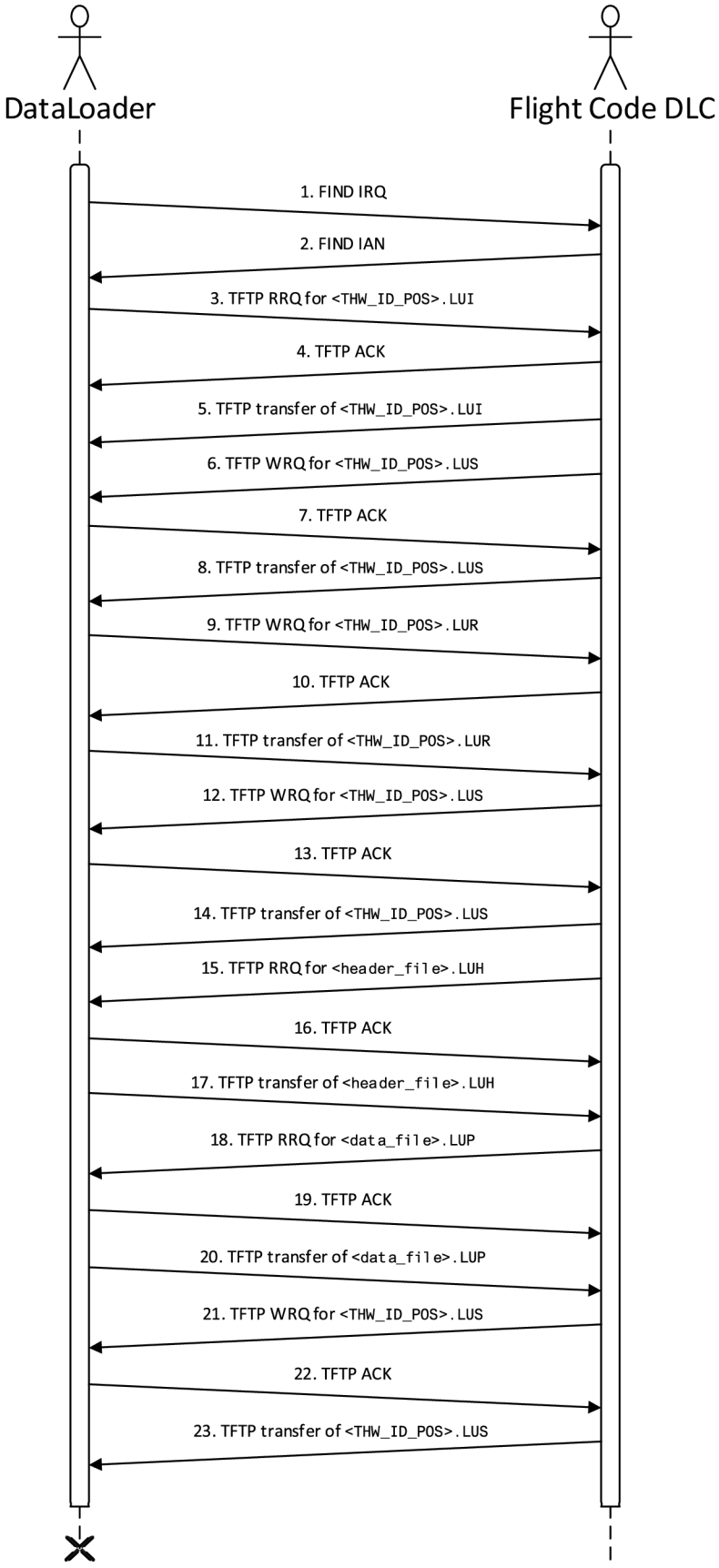

Fig. 9.1: Sequence diagram of data load process initiated by DataLoader

16. The request is acknowledged by the DataLoader (TFTP server at this moment)
17. TFTP transfer of the load upload header file from the DataLoader to DLC is done
18. TFTP read request for load upload part (`.LUP`) file is issued by DLC (TFTP client at this moment)
19. The request is acknowledged by the DataLoader (TFTP server at this moment)
20. TFTP transfer of the load upload part file from the DataLoader to DLC is done
21. TFTP write request for load uploading status (`.LUS`) file is issued by DLC (TFTP client at this moment)
22. The request is acknowledged by the DataLoader (TFTP server at this moment)
23. TFTP transfer of the load uploading status file from DLC to the DataLoader is done and the DataLoader checks the status provided within the file

The sequence is minimalistic, because more information might be exchanged during the upload operation. Namely the `<THW_ID_POS>.LUS` files can be pushed by the target hardware to the data loader more often, since they serve not only to give status after an operation is finished, but also as a heart beat signal letting the data loader know the target hardware is still working on the current operation. These heart beats are made with a predefined periodicity.

Another thing that could occur during the upload operation is an abort by user, propagated to the target hardware via the data loader. If an abort request occurs, the data loader does not reply with an ACK packet upon the next `<THW_ID_POS>.LUS` write request. Instead, it refuses the transfer with a TFTP error packet with error code 0 and an abort error message, which was described in section 3.4. The target hardware then tries to send the `<THW_ID_POS>.LUS` file again, this time to let the data loader know that the abort request was acknowledged and fulfilled. In the DataLoader application, user is allowed to abort an ongoing loading sequence from any application interface.

As was mentioned in section 3.4, the `<THW_ID_POS>` variable used in the file names is set by the ARINC 615A standard and is created by concatenation of the target hardware identifier, underscore, and the target hardware position. The other filenames (`<header_file>` and `<data_file>`) can be arbitrary, including even the extensions. The `.LUH` and `.LUP` are only recommended by the ARINC 665 standard.

9.2.4 Console Interface

DataLoader supports multiple interfaces. When started as a standard executable (*.exe* file) without any parameters, the GUI interface is presented. If parameters are supplied, DataLoader is started as a console application. It depends on the combination of the parameters, whether the console is interactive (i.e. requires user inputs), or tries to complete the whole data load process on its own.

9.2.5 GUI

To make the DataLoader application more user-friendly in case of its manual usage, it also provides a simple GUI. This GUI consists of the main application window, where there

are buttons which allow user to initiate the FIND operation (either standard one, or an IP address can be supplied via a text box), open a file explorer window where user can pick the `.LUH` header file that should be used for the data load, start the upload operation or abort an ongoing one, and show a new window with log outputs.

Apart from these functional buttons, GUI has a list of upload operation phases with a small light next to each phase. These light indicate the operation status. There is also a progress bar with a text box where the overall result of the data load is showed once it is finished.

9.2.6 TestStand Interface

One of the adapters for calling external programs that TestStand provides is the .NET adapter. When there is a compiled .NET application, either as an executable (*.exe* file) or a dynamic link library (*.dll* file), this adapter allows TestStand to invoke any public method within this application.

TestStand can pass parameters to the methods and receive the return values or output parameters in a standard way. TestStand supports numbers, strings, booleans, and their arrays as types of variables. On top of that, it allows user to create custom types, most typically containers holding a combination of the standard types. When the .NET method uses these standard types as either input or output, TestStand can work with them as with any other variables.

TestStand also has an object reference type, which is basically a pointer to memory. This type can be used to hold reference to a .NET object, which is created by a call to its constructor. With this object reference, its methods can be invoked and the instance can also be passed to any other method that takes object of the given type as a parameter. TestStand has to use the .NET methods to perform any operation with the object, e.g. use `ToString()` method to get the object represented as a string.

Using TestStand to control a .NET application is in its core actually very much like programming it normally in Visual Studio. Public classes and their public methods are accessible, methods are called, and objects are created and cleared by the .NET garbage collector when reference to them ceases to exist.

To make access to the DataLoader functionality from TestStand as easy as possible, a special static class serving as an interface was designed. This class uses overloaded methods to provide the following functionality:

- FIND operation
- Upload operation
- Abort of an ongoing upload operation

The FIND operation can be initiated either with no parameters, in which case FIND IRQs are broadcasted via all Ethernet adapters of the computer where the DataLoader is running, or with a string containing IP address at which SDU should be looked for. As a result, either an array of SDU objects, or an array of strings containing IP addresses of available SDUs is returned to TestStand.

The second group of methods starts and performs the upload operation itself. It is more convenient to run this step in the TestStand sequence in a new thread, because the operation is blocking (i.e. it returns control after the data load is finished). To start an upload operation, TestStand needs to supply the DataLoader with either the SDU object or a string with SDU's IP address, and a string with location of the `.LUH` header file. Upon completion, error code and an error message string are returned to TestStand.

The abort of the upload operation can be initiated from TestStand by calling the method and supplying either an SDU object or a string with IP address. The DataLoader checks whether there is an ongoing upload operation for the given SDU and if there is, it aborts it and returns error code and error message string back to TestStand.

9.3 Implementation

The DataLoader application was developed in Microsoft Visual Studio IDE (version 2010), using C# programming language. The implementation follows the design described above, so there is not much to be discussed in this section.

A ready made implementation of TFTP service from another project was used. Otherwise, standard .NET libraries were utilized. GUI was created using Microsoft's Form Designer tool.

Threading was used where it was necessary, for example for monitoring an abort request to an ongoing upload operation, or the GUI interface in order to keep it responsive at all times.

9.4 Verification and Validation

Since appropriate version of flight code software for Aspire 400 was not yet available at the time this report was written, verification and validation could be done only partially, using another type of ARINC 615A compliant SDU from a different Honeywell project. DataLoader was verified and validated on this SDU, nevertheless, the process shall be repeated with Aspire 400 SDU once it can be done.

In the verification and validation process using the other SDU, the already present flight code was first deleted and then loaded using U-Boot as if it would be in a standard production process. After that, flight code was reloaded using the DataLoader and its functionality was confirmed by running a set of built-in tests.

Initially, there was a plan to do the early verification and validation in a simulated environment like with the SerialParser. But in contrast with the SerialParser, where the COM port emulation and data generated on this virtual connection were fairly easy to provide, simulating an ARINC 615A compliant SDU has proved to be quite complex and complicated, hence this idea was dropped.

10 CONCLUSIONS

In this chapter, the results of this thesis are summarized, with emphasis on the benefits for Honeywell company. The implementation that was done is also discussed, including some problems that occurred during the development, known issues, and possible improvements.

To summarize the outcomes of the thesis, the official assignment was met to the extent allowed by the circumstances. It was for example not possible to test the implemented DataLoader application with the Aspire 400 SDU due to replanning of priorities in the Aspire 400 flight code development, where version fully implementing the data load functionality was not released yet. Therefore the functionality of the DataLoader application could only be tested on an ARINC 615A compliant SDU from another Honeywell project. Even though the transition to the Aspire 400 SDU should be easy, there is a slight risk some issues will rise in the process as they did with the SerialParser.

The timing of the thesis also turned out to be problematic, because the plans for the Aspire 400 project changed significantly after the assignment was made. Since the data load functionality is more important in the later phases, more urgent issues, for example on the production ATE, were prioritized, and work on the data loads was postponed.

As was already mentioned, lacking both hardware and software for the real operational testing was another problem. Since Aspire 400 is still under development, only a few engineering SDUs are available and none was available in Brno for a long time. Once the SerialParser was tested on a real SDU, problem with the way the data are read from the COM port mentioned in section 7.3 was revealed and the implementation had to be changed. The problem didn't occur during the previous testing neither in the simulated environment, nor using the MPC card in a special test jig.

But these problems aside, this thesis has significant positive impact on the way data loading is done during the production process. The designed applications are helping to automate data loads in an easy and reliable way, making the production process faster and more robust. Both are planned to be integrated in the TestStand sequences for production testing on Aspire 400. The applications should also be easily reusable, so other projects can benefit from them as well. And the theoretical part of the thesis can serve as a quick guide into the data load problematics on all levels.

There is naturally some space for improvement of the applications, too. Additional functionality could be added. For example for the SerialParser, it would be useful in some situations to have an ability to branch the flow of recursive patterns based on the incoming data, i.e. make a sort of conditional patterns. One use case where this could be handy is when some setup needs to be checked prior to following actions. With the conditional patterns, the SerialParser could for example check the value of an environmental variable and if it would not be set as expected, parser could issue a command to set the value correctly. At this moment, parser needs to do the setting step always in order to make sure the value is correct.

Another idea to improve the patters for the SerialParser would be to allow their definition using regular expressions. Using such an extension, it would be easily possible to

search for RX pattern with variable substrings in it. Such RX pattern has to be currently split into multiple shorter and static RX patterns.

The plans for the near future of the DataLoader are to test it throughly with Aspire 400, once all the necessary equipment is available. The full support for the downloading operation could also be implemented if required. At the time being, its support is not planned in Aspire 400 flight code, therefore the DataLoader has no reason to support it either. But this could of course change in the future.

It is also planned for the DataLoader to serve the data load needs of other projects, hence it will be necessary to test the application with other types of SDUs and their related equipment and fix any eventual bugs.

BIBLIOGRAPHY

[1] Airlines Electronic Engineering Committee. (2007). *ARINC Report 615A-3: Software Data Loader Using Ethernet Interface.* Annapolis, Maryland, USA: Aeronautical Radio, Inc.

[2] Karp, A. (2007, July 5). Carlyle Group to buy ARINC from airline shareholders. *Air Transport World.* Retrieved January 2, 2018, from `http://atwonline.com/operations/carlyle-group-buy-arinc-airline-shareholders`

[3] Haber, G. (2013, December 24). Arinc's new owner to sell off two subsidiaries. *Baltimore Business Journal.* Retrieved January 2, 2018, from `https://www.bizjournals.com/baltimore/news/2013/12/24/arincs-new-owner-to-sell-off-two.html`

[4] *About SAE International.* (n.d.) Retrieved January 2, 2018, from `http://www.sae.org/about/`

[5] Airlines Electronic Engineering Committee. (2016). *ARINC Report 665-4: Loadable Software Standards.* Bowie, Maryland, USA: SAE Industry Technologies Consortia.

[6] Airlines Electronic Engineering Committee. (1985). *ARINC Report 603-1: Airborne Computer Data Loader.* Annapolis, Maryland, USA: Aeronautical Radio, Inc.

[7] Airlines Electronic Engineering Committee. (2012). *ARINC Specification 429P1-18: Digital Information Transfer System (DITS) – Part 1 – Functional Description, Electrical Interfaces, Label Assignments and Word Formats.* Annapolis, Maryland, USA: Aeronautical Radio, Inc.

[8] Airlines Electronic Engineering Committee. (2004). *ARINC Specification 429P2-16: Mark 33 Digital Information Transfer System (DITS) – Part 2 – Discrete Word Data Standards.* Annapolis, Maryland, USA: Aeronautical Radio, Inc.

[9] Airlines Electronic Engineering Committee. (2009). *ARINC Specification 429P3-19: Mark 33 Digital Information Transfer System (DITS) – Part 3 – File Data Transfer Techniques.* Annapolis, Maryland, USA: Aeronautical Radio, Inc.

[10] Airlines Electronic Engineering Committee. (2012). *ARINC Specification 429P4: Digital Information Transfer System (DITS) – Part 4 – Archive of ARINC 429 Supplements.* Annapolis, Maryland, USA: Aeronautical Radio, Inc.

[11] Airlines Electronic Engineering Committee. (2017). *ARINC Specification 600-20: Air Transport Avionics Equipment Interfaces.* Bowie, Maryland, USA: SAE Industry Technologies Consortia.

[12] Airlines Electronic Engineering Committee. (2002). *ARINC Report 615-4: Airborne Computer High Speed Data Loader.* Annapolis, Maryland, USA: Aeronautical Radio, Inc.

[13] Airlines Electronic Engineering Committee. (2006). *ARINC Specification 664P1: Aircraft Data Network – Part 1 – Systems Concepts and Overview.* Annapolis, Maryland, USA: Aeronautical Radio, Inc.

[14] Airlines Electronic Engineering Committee. (2009). *ARINC Specification 664P2-2: Aircraft Data Network – Part 2 – Ethernet Physical and Data Link Layer Specification.* Annapolis, Maryland, USA: Aeronautical Radio, Inc.

[15] Airlines Electronic Engineering Committee. (2009). *ARINC Specification 664P3-2: Aircraft Data Network – Part 3 – Internet-Based Protocols and Services.* Annapolis, Maryland, USA: Aeronautical Radio, Inc.

[16] Airlines Electronic Engineering Committee. (2007). *ARINC Specification 664P4-2: Aircraft Data Network – Part 4 – Internet-Based Address Structure & Assigned Numbers.* Annapolis, Maryland, USA: Aeronautical Radio, Inc.

[17] Airlines Electronic Engineering Committee. (2005). *ARINC Specification 664P5: Aircraft Data Network – Part 5 – Network Domain Characteristics and Interconnection.* Annapolis, Maryland, USA: Aeronautical Radio, Inc.

[18] Airlines Electronic Engineering Committee. (2009). *ARINC Specification 664P7-1: Aircraft Data Network – Part 7 – Avionics Full-Duplex Switched Ethernet Network.* Annapolis, Maryland, USA: Aeronautical Radio, Inc.

[19] Airlines Electronic Engineering Committee. (2010). *ARINC Specification 664P8-1: Aircraft Data Network – Part 8 – Interoperation with Non-IP Protocols and Services.* Annapolis, Maryland, USA: Aeronautical Radio, Inc.

[20] Airlines Electronic Engineering Committee. (2017). *ARINC Characteristic 781-7: Mark 3 Aviation Satellite Communication Systems.* Bowie, Maryland, USA: SAE Industry Technologies Consortia.

[21] Zhongcai Z. (2012, June). *Introduction to Integrated Flash Controller* Freescale Technology Forum. Available at `https://www.nxp.com/files-static/training_pdf/FTF/2012/americas/WBNR_FTF12_NET_F0109.pdf`

[22] *SwiftBroadband.* (n.d.) Retrieved January 3, 2018, from `https://www.inmarsat.com/service-collection/swiftbroadband/`

[23] Texas Instruments. (2006). *TMS320C6000 DSP Host Port Interface (HPI) Reference Guide.* Texas Instruments.

[24] IEEE Computer Society. (2013). *IEEE Standard for Test Access Port and Boundary-Scan Architecture.* New York, New York, USA: The Institute of Electrical and Electronics Engineers, Inc.

[25] *Null-modem emulator.* (n.d.) Retrieved May 6, 2018, from `https://sourceforge.net/projects/com0com/`

[26] Sollins, K (1992, July). *The TFTP Protocol (Revision 2).* Retrieved May 20, 2018, from `https://tools.ietf.org/html/rfc1350`

LIST OF ABBREVIATIONS

ACARS	Aircraft Communications Addressing and Reporting System
ACK	Acknowledgment
ADL	Airborne Data Loader
AFDX	Avionics Full-Duplex Switched Ethernet
AOI	Automated Optical Inspection
AXI	Automated X-ray Inspection
ARINC	Aeronautical Radio, Incorporated
ASCII	American Standard Code for Information Interchange
ATE	Automated Test Equipment
ATP	Acceptance Test Procedure
BCL	Binary Command Language
BFP	Batch File Part
BGAN	Broadband Global Area Network
CAN	Controller Area Network
CBIT	Continuous Built-In Test
CC	Channel Card
CCIF	Channel Card Interface
CPU	Central Processing Unit
CR	Carriage Return
CRC	Cyclic Redundancy Code
DABC	Dual Aeronautical BGAN card
DDR	Double Data Rate
DITS	Digital Information Transfer System
DLC	Data Load Controller
DLF	Data Load Function
DLL	Dynamic Link Library
DSP	Digital Signal Processor
ECC	Error-Correcting Code
EEPROM	Electrically Erasable Programmable Read-Only Memory
eSPI	Enhanced Serial Peripheral Interface
FC	Flight Code
FIND	Find Identification of Network Devices
FIT	Flattened Image Tree
FPGA	Field-Programmable Gate Array
GPCM	General Purpose Chip Select Machine
GPIO	General-Purpose Input / Output
HBIT	Hardware Built-In Test
HPI	Host Port Interface
I^2C	Inter-Integrated Circuit
IAN	Information Answer

IBIT	Initiated Built-In Test
ICSP	In-Circuit Serial Programming
ICT	In-Circuit Test
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IFC	Integrated Flash Controller
IRQ	Information Request
ISO	International Standards Organization
ISP	In-System Programming
JTAG	Joint Test Action Group
LAN	Local Area Network
LDU	Link Data Unit
LF	Line Feed
LRU	Line Replaceable Unit
LSAP	Loadable Software Airplane / Aircraft Part
LSP	Loadable Software Part
LUH	Load Upload Header
LUI	Load Uploading Initialization
LUP	Load Upload Part
LUR	Load Uploading Request
LUS	Load Uploading Status
MAC	Media Access Control
MCU	Modular Concept Unit
MDC	Management Data Clock
MDIO	Management Data Input / Output
MES	Message Event Service
MPC	Main Processor Card
MSP	Media Set Part
NVRAM	Non-Volatile Random Access Memory
ORT	Owner Requirements Table
OSI	Open Systems Interconnection
PBL	Pre-Boot Loader
PBL	Pre-Boot Loader
PCB	Printed Circuit Board
PDL	Portable Data Loader
PN	Part Number
PROM	Programmable Read-Only Memory
QoS	Quality of Service
RAM	Random Access Memory
RCW	Reset Configuration Word
RF	Radio Frequency

RFC	Request For Comments
RGMII	Reduced Gigabit Media-Independent Interface
RRQ	Read Request
SBB	Swift Broadband
SDRAM	Synchronous Dynamic Random Access Memory
SDU	Satellite Data Unit
SGMII	Serial Gigabit Media-Independent Interface
SIS	Standalone Identification System
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SRU	Shop Replaceable Unit
SVF	Serial Vector Format
TAP	Test Access Port
TCK	Test Clock
TDI	Test Data Input
TDO	Test Data Output
TFTP	Trivial File Transfer Protocol
TMS	Test Mode Select
TRST	Test Reset
UART	Universal Asynchronous Receiver / Transmitter
UCC	Unified Communications Controllers
UMTS	Universal Mobile Telecommunications Service
UMTS	UMTS Subscriber Identity Modules
UUT	Unit Under Test
WRQ	Write Request