HAW HAMBURG

BACHELORTHESIS
Ludwig von Feilitzen

SpaceWire driver and ECSS protocol handling for mission-critical applications running on a LEON4 processor

FACULTY OF COMPUTER SCIENCE AND ENGINEERING
Department of Information and Electrical Engineering

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

HAMBURG UNIVERSITY
OF APPLIED SCIENCES

Hochschule für Angewandte
Wissenschaften Hamburg

Ludwig von Feilitzen

SpaceWire driver and ECSS protocol handling for mission-critical applications running on a LEON4 processor

Bachelor Thesis based on the examination and study regulations
for the Bachelor of Engineering degree programme
Bachelor of Science Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg
Supervising examiner: Prof. Dr. Pawel Buczek
Second examiner: Prof. Dr. Lutz Leutelt

Day of delivery: 29/09/2020

Ludwig von Feilitzen

Title of Thesis

SpaceWire driver and ECSS protocol handling for mission-critical applications running on a LEON4 processor

Keywords

SpaceWire, ECSS, RMAP, GR740, LEON4

Abstract

The SpaceWire communication protocol simplifies the design of communication systems to be deployed in space. It can be used in conjunction with higher level protocols that further specify how two SpaceWire nodes can communicate with each other. In order to do this both nodes need the correct drivers which allows them to create and understand communication in the format of the protocol. This document is the result of a thesis project within the scope of which such a driver has been developed. The driver is tested with the help of a test-bench, the creation of which is also a part of the project. The driver is written for a GR740 System-on-a-Chip and adheres to the RMAP protocol specification.

Ludwig von Feilitzen

Thema der Arbeit

SpaceWire-Treiber und Handhabung des ECSS-Protokolls für missionskritische Anwendungen, die auf einem LEON4-Prozessor laufen

Stichworte

SpaceWire, ECSS, RMAP, GR740, LEON4

Kurzzusammenfassung

Das SpaceWire-Kommunikationsprotokoll vereinfacht den Entwurf von Kommunikationssystemen, die im Weltraum eingesetzt werden sollen. Es kann in Verbindung mit Protokollen höherer Ordnung verwendet werden, die näher spezifizieren, wie zwei SpaceWire-Knoten miteinander kommunizieren können. Dazu benötigen beide Knoten die richtigen Treiber, die es ihnen ermöglichen, die Kommunikation im Format des Protokolls zu erstellen und zu verstehen. Dieses Dokument ist das Ergebnis eines Dissertationsprojekts, in dessen Rahmen ein solcher Treiber entwickelt wurde. Der Treiber wird mit Hilfe eines Prüfstandes getestet, dessen Erstellung ebenfalls Teil des Projektes ist. Der Treiber wurde für ein GR740 System-on-a-Chip geschrieben und hält sich an die RMAP-Protokollspezifikation

Contents

List of Figures

List of Tables

1 Introduction

Communication systems in equipment operating in space face a special set of requirements since the environment imposes many different challenges, e.g. noise in the form of radiation. Therefore, much care must be taken before the equipment is deployed to ascertain that communication between the system parts can function reliably. The SpaceWire standard was created in order to facilitate the communication on such a system by providing an energy efficient, noise resistant and flexible method of communication. Being flexible, the SpaceWire standard allows higher level protocols to put further specifications on normal SpaceWire communication and thus acts a base for these higher level protocols. One of these protocols is the *Remote Memory Access Protocol*, which is abbreviated as *RMAP*. RMAP uses the SpaceWire standard to further define three different commands which allows one device to read and write to memory of another device when the two are connected via a SpaceWire network. In order for this type of communication to function both the messenger (also called source node) and the receiver (also called destination node) must have drivers adhering to the standard's specifications. These drivers can be implemented in either hardware or software.

1.1 Task Description

This document is the result of a thesis project with the main goal of creating and testing an RMAP software driver for a destination node. Whereas a driver for a source node would need to be able to send RMAP commands, a destination node driver needs to respond to incoming RMAP commands and transmit replies back to the source node. The driver shall run on a LEON4 processor on a System-on-a-Chip called GR740 developed by the company Cobham Gaisler. The GR740 is designed as the European Space Agency's (ESA) next generation microprocessor and is available on a development board with the name GR-CPCI-GR740. This is the development board used throughout this project.

In addition to the development of an RMAP software driver, a test-bench shall be created which should allow for convenient testing of the developed RMAP driver. In order to do this the test-bench should run on a separate machine which is able to communicate with the GR740 via a SpaceWire network. The test-bench shall be able to perform several tests at once and the tests shall be performed according to scripts created by a tester, as to automate the testing process as much as possible.

The remainder of this document contains background descriptions of all the system components, instructions on setting up a development environment, formal project requirements, design and implementation and a description of test-bench usage and project state.

2 Theory

In order to create a proper understanding of the project described in this document it is important to have an overview of all the system components. This section contains descriptions of the relevant components and modules and should be carefully reviewed before any further development using this project setup is attempted.

2.1 SpaceWire

A central part of this thesis project is the usage of SpaceWire. SpaceWire is a data-handling network developed for use on-board spacecraft. The standard covers the first two layers of the OSI model (physical and data link) and was developed to exhibit some specific characteristics:

- High speed
- Low power usage
- Simplicity
- Low implementation cost
- Architectural flexibility

These characteristics make SpaceWire a good choice for spacecraft communication, and has since its publication been adopted by the European Space Agency (ESA) and the National Aeronautics and Space Administration (NASA), among others. It was developed at the University of Dundee on a contract from ESA, who wanted to solve the problems of the predecessor of SpaceWire, a standard called IEEE 1355-1995. The main goals of the SpaceWire standard are to:

- facilitate the construction of high-performance on-board data handling systems

- help reduce system integration costs
- promote compatibility between data-handling equipment and subsystems
- encourage re-use of data-handling equipment across several different missions

The following subsections will provide an in-depth look into the SpaceWire standard. For further information on SpaceWire the reader is referred to the SpaceWire user guide from STAR-Dundee [12], from which the information in this section is derived.

2.1.1 An Overview

The two most important components of the SpaceWire standard are the SpaceWire link and the SpaceWire packet. The links are point-to-point data links which connects two SpaceWire nodes, or a node to a SpaceWire router. A node can be an instrument, processor, mass-memory unit etc. Each link is a full-duplex, serial data link and operates at rates between 2 Mbit/s and 200 Mbit/s. It uses two signals, data and strobe, in each direction to transmit a serial bit stream. Each signal is driven by low voltage differential signaling (LVDS) which means two wires are required for each signal. This results in four twisted pairs in each SpaceWire cable.

Bit synchronization in a link is achieved by sending the clock signal encoded in the serial data. This is the purpose of the strobe signal: by using an XOR operation on the data and strobe signal the clock is recovered. This is done to reduce the maximum clock-to-data skew. Synchronization is only done once, when the link is started. If synchronization is later lost it will be detected through a parity error mechanism and the link will restart.

By using a state machine SpaceWire makes it relatively simple to track the current link state and control operations like starting a link, keeping it running, transmitting data, checking if receiver is ready and recovering from link errors. The state machine is always controlled by the SpaceWire interface and is therefore transparent to the user application.

The information sent over a SpaceWire network is grouped into packets. The packets have a very simple structure and are therefore suited to be used as the base of higher level protocols, which can further define the fields of a SpaceWire packet. This is the case with the Remote Memory Access Protocol (RMAP) which is described in Section 2.2.1. The structure of a SpaceWire packet can be seen in Figure 2.1.

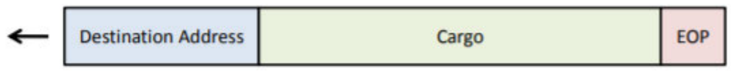


Figure 2.1: Structure of a SpaceWire packet [12].

The first part of the packet which is sent is the destination address. The destination address is a list of data characters which represent either the address of the destination node or the path that the packet has to take through the network to get there. In the case of point-to-point connections a destination address is not necessary. The next part of the SpaceWire packet is the cargo. This is the actual data to be transmitted from one node to another. Within the space of the cargo field higher level protocols are free to determine their expected structure. Any number of bytes can be transferred in the cargo of a SpaceWire packet. The final part of the packet is the special *End of Packet* (EOP) character which indicates the end of a packet. This is necessary as there are no limits to the size of a SpaceWire packet and the cargo length is not declared at the beginning of the packet. There is also the *Error End of Packet* (EEP) character, which indicates that the packet was not successfully transmitted from the source.

SpaceWire networks can be created by combining several point-to-point links, and using routers makes it possible to be very flexible regarding the network architecture. The architecture used in this thesis project is shown in Figure 2.2.

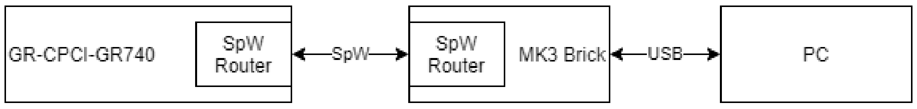


Figure 2.2: SpaceWire network architecture in this project.

Even though the only SpaceWire communication takes place between the SpaceWire MK3 brick (see Section 2.6) and the development board (see Section 2.5), please note that this is not a point-to-point connection, as there are routers between the source and destination. Many more architectures are possible and for a deeper look into these the reader is referred to Section 2.4 of the SpaceWire user guide [12].

2.1.2 SpaceWire Links

In this section, the operation of a SpaceWire link is described in further detail. The SpaceWire standard covers electrical properties, connectors, cables and logical protocols which are defined on six different levels:

- **Physical Level:** SpaceWire connectors, cables, cable assemblies and printed circuit board tracks
- **Signal Level:** Signal encoding, voltage levels, noise margins, and data signaling rates
- **Character Level:** Data and control characters used to manage the flow of data across a SpaceWire link
- **Exchange Level:** Protocols for link initialization, flow control, link error detection and link error recovery
- **Packet Level:** Definition of how data for transmission over a SpaceWire link is split up into packets
- **Network Level:** Structure of a SpaceWire network and the way in which packets are transferred from a source node to a destination node across a network. The network level also defines how link errors and network level errors are handled.

The following subsections will describe the first four levels. The packet level is sufficiently described by Figure 2.1 in combination with the discussion in Section 2.1.1, whereas the network level is described in further detail in Section 2.1.3.

Physical Level

The components on the physical level of the SpaceWire standard were developed to meet the electromagnetic compatibility (EMC) specifications of typical spacecraft. As previously mentioned, a SpaceWire cable contains four twisted pairs where each twisted pair provides one signal in one direction. Each of these pairs is surrounded by a separate shield, and the entire cable content is surrounded by an overall shield. The cables have a characteristic impedance of 100Ω differential impedance which is matched to the line termination impedance (see Figure 2.5). In general, the cables were designed to exhibit low signal attenuation, low cross-talk and good EMC performance. A SpaceWire cable can be used for data rates of 200 Mbit/s for distances up to 10 meters. It is possible to increase the wire gauge of the conducting wires in order to reduce cable attenuation if longer distances are required. The main drawback of the cable structure is the mass of the cable, namely around 87 g/m [12].

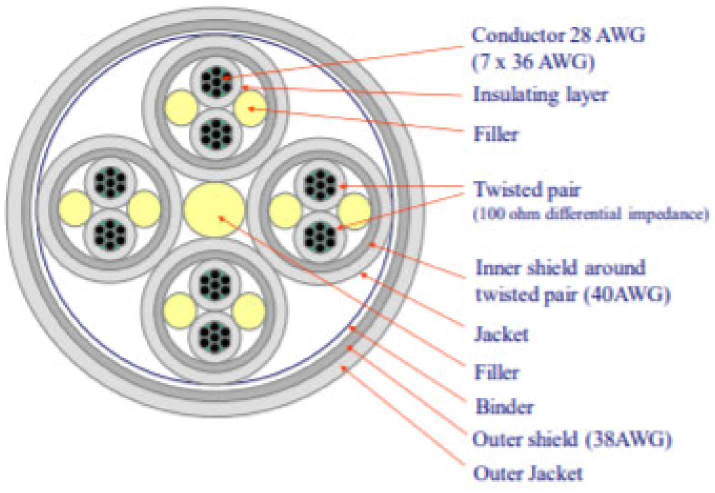

Conductor 28 AWG (7 x 36 AWG)
Insulating layer
Filler
Twisted pair (100 ohm differential impedance)
Inner shield around twisted pair (40AWG)
Jacket
Filler
Binder
Outer shield (38AWG)
Outer Jacket

Figure 2.3: Structure of a SpaceWire cable [12].

The SpaceWire connectors have eight signal contacts (for the four twisted pairs) as well as a screen termination contact as displayed in Figure 2.4. The connector is specified as a nine pin micro-miniature D-type connector. The outer shield connects to the backshell (part protecting the connection between connector and cable) at each end of the cable. The shield of each twisted pair is connected to pin 3 which acts as signal ground, however each such inner shield is only connected at one end of the cable, namely the end driving the signal. It is mentioned in the SpaceWire user guide by STAR-Dundee [12] that this arrangement is far from ideal and should in the future be revised to having all shields being terminated at both ends of each cable.

The SpaceWire standard also specifies requirements of PCB tracks, for information and guidelines regarding these the reader is referred to Section 3.1.4 of the SpaceWire user guide [12].

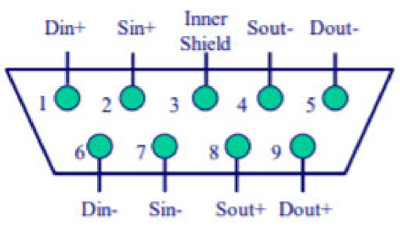


Figure 2.4: Pin-out of a SpaceWire connector [12].

Signal Level

As previously mentioned, the signaling technique used for SpaceWire is LVDS. This section describes LVDS in further detail, as well as a look into the data encoding process of SpaceWire.

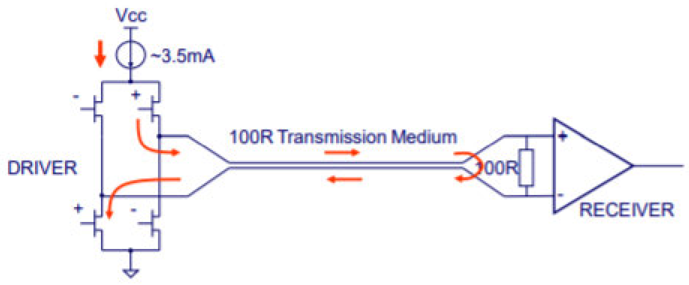


Figure 2.5: Low Voltage Differential Signaling (LVDS) operation [12].

A typical LVDS driver and receiver can be seen in Figure 2.5. The transmission medium can be either a cable or PCB traces with a differential impedance of 100Ω, which is matched by the 100Ω termination resistance. The matching of impedance is done in order to avoid signal reflections. The driver is modeled as a constant current source of about 3.5 mA which provides the current that flows along the transmission medium and through the termination resistance, thus creating a voltage at the receiver. Two pairs of transistors in the driver are used to control the direction of the current, which in turn affects the polarity of the voltage drop over the termination resistance. By toggling the current between +3.5 mA and -3.5 mA the receiver side comparator will detect the voltage polarity and output the corresponding logical value (HIGH at +3.5 mA, LOW at -3.5 mA). LVDS receivers have a high input impedance which means that most of the provided current flows through the termination resistor. Therefore the nominal voltage level at the receiver for LVDS signaling is approximately ± 350 mV. Some typical voltage levels and thresholds are shown in Figure 2.6.

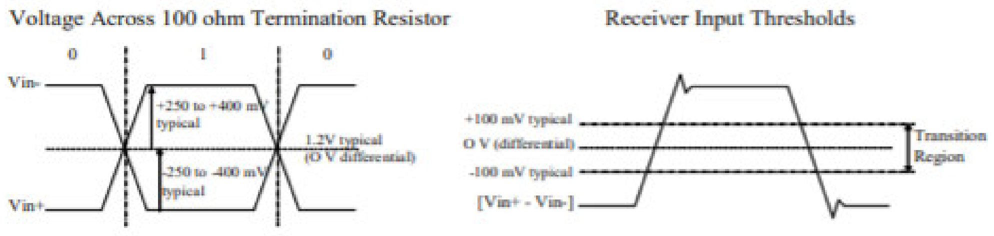


Figure 2.6: Typical LVDS voltage levels [12].

SpaceWire uses a coding scheme called *Data-Strobe Encoding*, which is also used in the Firewire standard. The data is sent unmodified while the strobe signal changes state whenever the data signal remains constant from one bit interval to the next. The result of this is that the clock signal can be recovered by performing an XOR operation on the data and strobe signals. An example of this process is displayed in Figure 2.7.

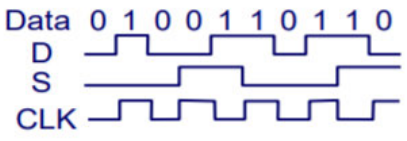


Figure 2.7: Data-Strobe encoding example [12].

Character Level

SpaceWire uses two types of characters, data and control characters. In addition to these there are two special control codes. This section contains descriptions on these characters and their respective functions.

- **Data characters:** The data characters contain an eight bit data value which is transmitted LSB first. The data characters also contain a parity bit and a data-control flag for a total of 10 bits. The parity coverage of a SpaceWire character is a bit unusual in that it covers the eight data bits (or two bit control code) of the previously sent character, in addition to itself and the current data-control flag. This is done in order to avoid incorrect decoding of a character when the data-control flag is in error, since the length of the two character types differ. The parity is set to produce odd parity. The data-control flag is set to zero to indicate a data character.

- **Control characters:** Control characters are a total of four bits in length. The first bit is the parity bit, which functions in the same way as for a data character. The following bit is the data-control flag, which is set to one to identify it as a control character. The last two bits are control code bits which allows for a total of four distinct control characters. One of these is the escape code (ESC) which can be used to form longer control codes.
- **Special control codes:** Currently, two longer control codes are specified: the NULL code and the time code. The NULL code is formed by an ESC character followed by flow control token (FCT). The time code is formed by an ESC character followed by a single data character.

The structures of all these types of characters are summarized in Figure 2.8.

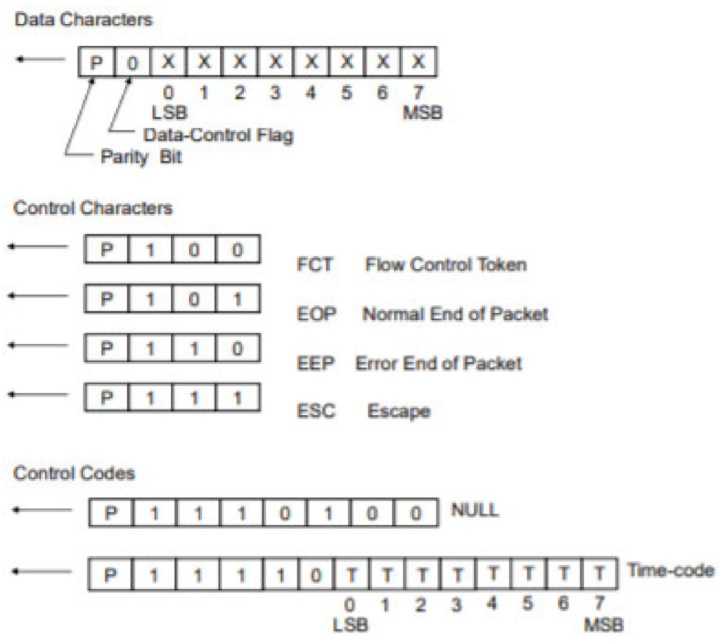


Figure 2.8: SpaceWire character types [12].

There are three main usages of SpaceWire characters, namely link control, sending data packets and sending time codes. The control characters NULL and FCT are used for link control and are also known as L-Chars (for Link Characters). They are used in the exchange level and are not passed on to higher levels. To send data packets the data characters and two types of control characters (EOP and EEP) are used. These are known as N-Chars (for Normal Characters). EOP and EEP are end-of-packet markers and are passed up to the packet level. EOP (end of packet) indicates a normal end of packet whereas EEP (error end of packet) indicates that an error took place during packet transmission. The time codes are used for distributing system time across a SpaceWire network and are further explained in Section 2.1.4.

Exchange Level

The exchange level is responsible for link initialization, flow control and error handling which in addition to the SpaceWire interface will be described in this section.

A SpaceWire link interface can be found at each end of a SpaceWire link. After initialization, the link interface can send and receive data packets and time codes. A block diagram of a SpaceWire interface is shown in Figure 2.9.

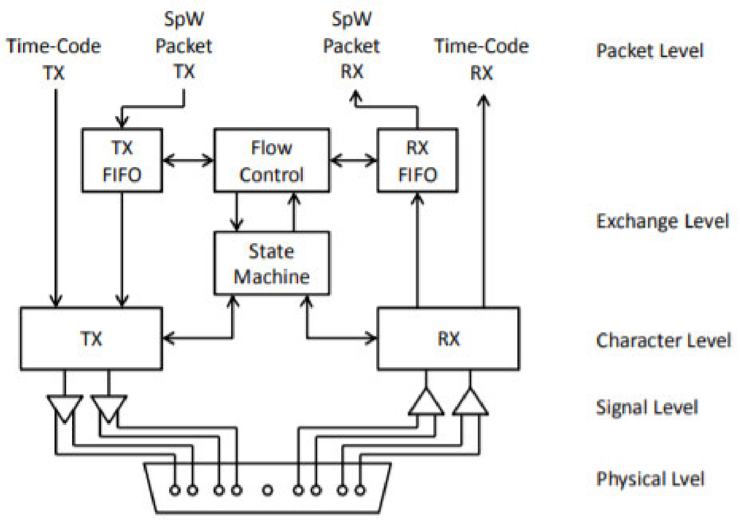


Figure 2.9: Block diagram of a SpaceWire interface [12].

For transmission, each character is passed to the transmission FIFO, starting with the data character containing the destination address. Incoming SpaceWire packets are received to the reception FIFO and can then be read out character by character by the application in control. Time codes bypass the FIFOs and are transmitted as soon as the current character is finished being sent. The state machine controls link initialization and manages recovery from any errors detected on the link. To prevent overflow of the receiving FIFO the flow control part of the interface contains circuitry to monitor the available FIFO space and uses FCTs to control the data being sent from the other end of the link.

The state diagram shown in Figure 2.10 best describes the order of events when starting a SpaceWire link, including the error handling processes. Following reset the SpaceWire interface enters the *ErrorReset* state in which both the receiver and transmitter hardware are reset. To ensure that the reset completes, the interface remains in this state for 6.4 microseconds before moving on to the *ErrorWait* state where it enables the receiver but holds the transmitter in a reset state. After 12.8 microseconds the interface moves to the *Ready* state, where it waits for a command to start the link. This command will be given by the application that controls the SpaceWire interface. The link can also be set to operate in auto-start mode, in which the state transition from *Ready* to *Started* can be triggered by the interface receiving a bit instead of an explicit command from the local application. This is very useful when starting a link remotely. Both link ends can be set to auto-start mode in order to allow either of them to initiate communication. The state transition will not be attempted regardless of other events if the *LinkDisabled* flag is set.

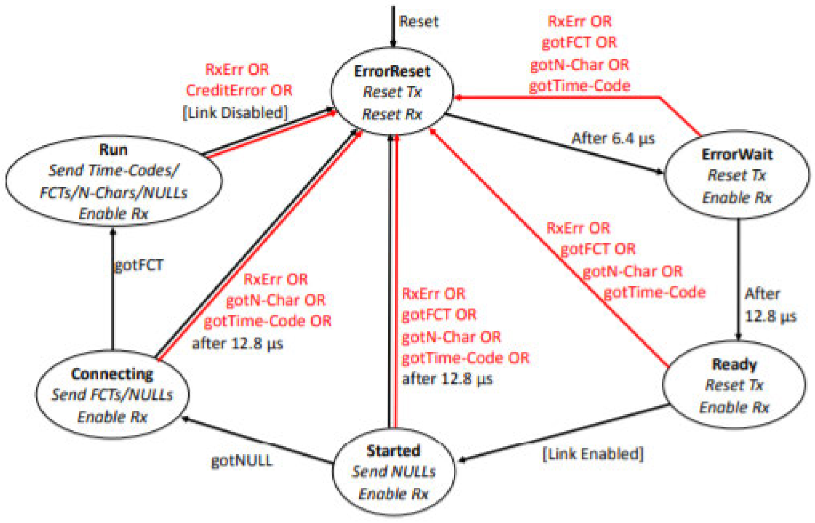


Figure 2.10: State diagram of SpaceWire link initialization and error recovery [12].

Once the interface is in the *Started* state it starts sending out NULL characters. If the other link end is sending out NULL characters (either by having been explicitly started or by being in auto-start mode and receiving a NULL) the first link end will detect this and move to the *Connecting* state. The other end should also have received a NULL at this point so it will move on to the *Connecting* state as well. In this state the interfaces will both send out a burst of FCTs. When receiving a FCT a final state transition to the *Run* state is made. This completes what is known as the NULL/FCT handshake and if successful both link ends are ready for communication. The interface will remain in this state until the link is disabled or an error is detected.

The handshake can fail in two ways, either by errors detected on the link or by timing out. In the *Started* state the handshake reaches its timeout after 12.8 microseconds of not receiving any NULL characters and resets, thus restarting the process. In the *Connecting* state timeout is reached after 12.8 microseconds of not receiving any FCTs. The possible types of link errors are:

- Disconnect error
- Parity error
- Escape error
- Credit error

A link disconnection is detected when after the reception of a data bit no new bit is received within the link disconnection timeout window of 850 nanoseconds. The parity error occurs when the parity bit is erroneous and the escape error occurs when an ESC character is followed by another ESC, EOP or EEP characters which are sequences that are not allowed. The credit error occurs when a N-Char arrives even though there is no room for it in the reception FIFO. The link flow control of the SpaceWire interface makes sure that no characters are sent from the other end if there is not space available for them at the receiving end. This is done by the receiving end indicating that is has space for eight more N-Chars by sending a FCT. The other end receives this FCT and therefore knows it can safely send eight N-Chars. The receiving end can report more than eight places of available buffer space by sending several FCTs. A SpaceWire interface can have up to seven outstanding FCTs, thus reporting space for up to 56 N-Chars. Through this mechanism, FCTs are in effect traded for N-Chars and allows the transmitting side to know if the other end is ready to receive data packets.

Within the context of Figure 2.10 a *RxErr* error is a disconnect, parity or escape error. As seen in the figure, transitions to the *ErrorReset* state are not only triggered by the four discussed error types but also by reception of a character that is invalid in the current state. For example, if the interface is in the *Connecting* state and receives a time code or N-Char, this is seen as an error and the link is reset.

For further information, see Section 3.4 of the SpaceWire user guide by STAR-Dundee for detailed explanations of the order of events during link initialization and error management [12].

2.1.3 SpaceWire Networks

SpaceWire allows flexible network architectures by combining routers and point-to-point links. This section contains descriptions on how packets are forwarded throughout a SpaceWire network.

SpaceWire Addressing

SpaceWire uses two different kinds of addressing, path addressing and logical addressing. Logical addressing involves specifying the destination address as a single byte with a value between 32 and 254 which uniquely identifies the destination node. This type of addressing is conceptually easy to understand and only requires one byte, however it comes with the drawback that each router in the packet's path must have been configured as to forward all incoming packets with a certain address to a certain port. Therefore, any changes to the network architecture might require reconfiguration of one or several routers. Path addressing does not have this drawback but instead might require several bytes in the destination address of a packet. When using path addressing the destination address consists of several bytes, each with a value between 1 and 31. Each byte represents the port that the packet should be forwarded out through by the next router it encounters. If a router receives a packet with the first byte having a value of 4, then it will remove this byte as it is no longer needed and proceed to forward the packet out of port 4. This exposes the next address byte for the following router, and the process is repeated until the packet reaches its destination. At that point the packet's addressing bytes should have been completely removed and the destination node will see the cargo part of the packet. If the network architecture changes, one must only modify the path addressing bytes at the beginning of each packet instead of re-configuring any routers. The addressing mode must not be known in advance by the routers, as path addressing always uses values between 1 and 31 and logical addressing uses values between 32 and 254 (the value 255 is reserved for future applications).

SpaceWire Routing

The SpaceWire routers mainly consist of a number of SpaceWire interfaces and a switch matrix. The concept is demonstrated in Figure 2.11.

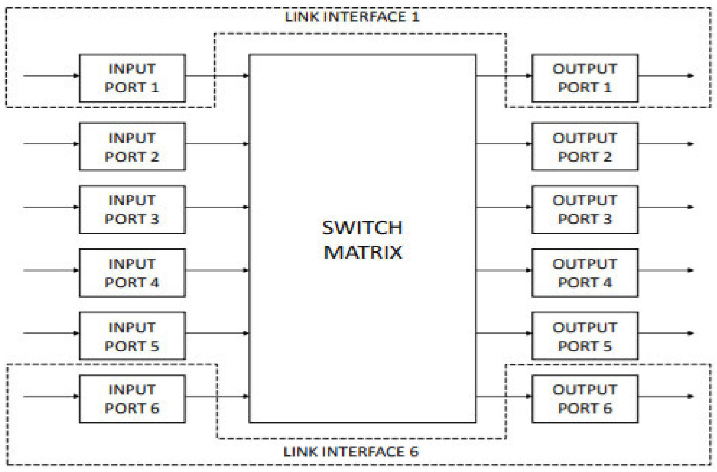
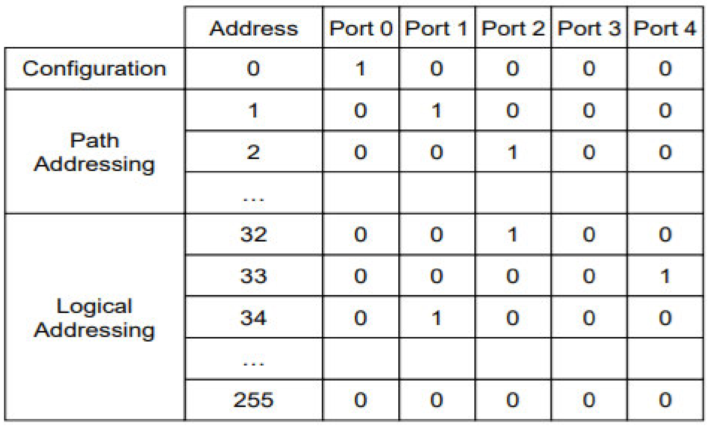


Figure 2.11: Switch matrix of a SpaceWire router [12].

By configuring the switch matrix incoming packets can be forwarded to one or several desired output ports. If path addressing is used the first byte of the incoming packet will be interpreted as the physical port number to which the packet should be forwarded, and if logical addressing is used the router will look up the entry in the configured routing table to determine which is the correct output port. Therefore, the routing tables of each router in a SpaceWire network which will use logical addressing must be configured before use.



	Address	Port 0	Port 1	Port 2	Port 3	Port 4
Configuration	0	1	0	0	0	0
Path Addressing	1	0	1	0	0	0
	2	0	0	1	0	0
	...					
Logical Addressing	32	0	0	1	0	0
	33	0	0	0	0	1
	34	0	1	0	0	0
	...					
	255	0	0	0	0	0

Figure 2.12: Example of routing table entries in a SpaceWire router [12].

A typical routing table is shown in Figure 2.12. The correct output ports for an incoming packet with a certain address are specified as a series of 1's and 0's acting as flags for the existing ports. For example, the shown routing table will lead to all incoming packets having a leading byte with value 33 being routed to port 4. It is also possible to define alternative output ports by setting several cells in one row to 1. This is called *Group Adaptive Routing* and allows the router to pick a different output port if the other ones are busy. This is useful when several ports lead (directly or indirectly) to the same destination node.

The destination address with value 0 is a special address, as it leads to the internal configuration port of a SpaceWire router. This port is used to remotely configure or access the status information of a router. An example of such a configuration is the routing table, which can be configured via this port.

2.1.4 Time-codes

SpaceWire provides a way to quickly distribute system time over a network and therefore synchronize the system components. The time information is given as *ticks*, which is an incrementing value which can be synchronized to the system in which the network is used. A time code consists of an ESC character followed by a data character. The time code tick count is found in the six least significant bits of the data character. The value of the two most significant bits within the data field is 0b00, as all other values are reserved.

SpaceWire interfaces have a separate time code interface in order to quickly propagate the time codes. These consist of two signals, *TICK_IN* and *TICK_OUT* as well as two eight-bit input and output ports. When a time code is to be distributed throughout a system, the so-called time master sets the eight input lines of the SpaceWire interface to the time code which should be sent followed by asserting the *TICK_IN* signal. This will cause the SpaceWire interface to save the presented value to its time-counter, which is an internal register which holds the last received (and valid) time code. At this point *TICK_OUT* is asserted to signalize that the interface received a new time code, and the value of the received time code is put on the eight output lines. If the SpaceWire interface which received the time code belongs to a router, the *TICK_OUT* signal propagates to the *TICK_IN* interfaces of all the router output ports, which causes all nodes connected to the router to update their time-codes as well. Through this mechanism time codes are distributed through the network. There is never more than one time master in a SpaceWire network, and it is nothing but a node or router which generates a periodic *TICK_IN* signal.

Time codes are only updated if they are deemed valid. During normal operation the time code values increase from 0 to 63 before rolling around and continuing. The incoming time codes should always be exactly one tick more than the time code currently residing in the node's time counter to be deemed valid. If the incoming time code is not valid the node still updates the internal time counter but does not assert the *TICK_OUT* signal, thus preventing an undesired time code updating loop if there is a circular connection. This means that if a router receives a time code which it propagates and receives again on another port through a circular connection, the value will be the same as in the time counter and therefore deemed invalid and is ignored.

Time codes can be used for synchronization, time distribution, event signaling and even as a means of distributing interrupts across a system due to its rapid propagation. For more information regarding time codes, see the SpaceWire user guide by STAR-Dundee [12].

2.2 The ECSS Protocol Suite

The European Cooperation for Space Standardization (ECSS) is an initiative which was created in order to develop user friendly standards for systems used in European space activities. It is a cooperative effort from the ESA, European national space agencies and European industry associations. SpaceWire is in fact maintained by ECSS and goes under the formal name ECSS-E-ST-50-12 [10]. There are also standards for protocols which can be used together with SpaceWire. These higher level protocols use the flexible structure of SpaceWire and imposes further definitions on the communication. This is referred to as protocols which are 'laying on top of' SpaceWire. ECSS maintains some of these higher level protocols and refers to them as the ECSS-E-ST-50-5x series. At the moment of writing this series contains two standards, namely the Remote Memory Access Protocol (RMAP/ECSS-E-ST-50-52) and the CCSDS packet transfer protocol (ECSS-E-ST-50-53). The protocol of interest within the scope of this thesis is RMAP which will be described in detail in the following subsection. CCSDS is the abbreviation for the Consultative Committee for Space Data Systems and the CCSDS packet transfer protocol simply aims to encapsulate so called CCSDS packets in SpaceWire packets and transfer them across networks [9]. For a more detailed look into the CCSDS packet transfer protocol the reader is referred to the ECSS-E-ST-50-53 document which can be found on the ECSS website under *Standards -> Active Standards*.

Although only two are mentioned, there are other higher level protocols which lay on top of SpaceWire. The ones mentioned so far are maintained by ECSS, whereas the others fall under the responsibility of other organizations. ECSS maintains an additional standard in the ECSS-E-ST-50-5x series called ECSS-E-ST-50-51 and this standard defines protocol identifiers for all protocols which are resting on SpaceWire. The standard contains a helpful table which provides an overview of the existing protocols and can be seen in Table 2.1.

Table 2.1: Protocol identifiers for different higher level protocols using SpaceWire [9].

Protocol Identifier	**Protocol**	**Specified in**
0	Extended Protocol Identifier	Clause 5
1	Remote Memory Access Protocol	ECSS-E-ST-50-52
2	CCSDS Packet Transfer Protocol	ECSS-E-ST-50-53
238	GOES-R Reliable Data Delivery Protocol	417-R-RTP-0500 Version 2.1, 16 January 2008
239	Serial Transfer Universal Protocol	SMCS-ASTD-PS-001 Issue 1.1, 24 July 2009

2.2.1 RMAP

RMAP is a protocol which allows reading and writing to specific memory locations of another device on a SpaceWire network. It was developed to be able to configure other network devices, control SpaceWire units and to gather data from those units, although it can be used for a wide range of applications. RMAP is commonly used to configure both SpaceWire routers and SpaceWire interfaces.

RMAP consists of three commands:

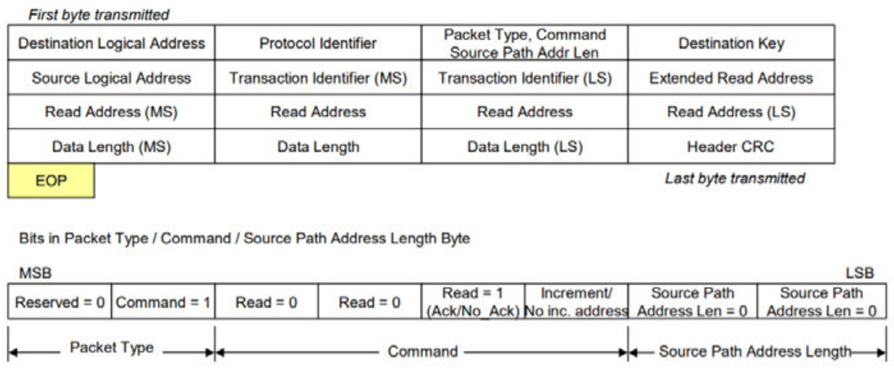
- Read
- Write
- Read-Modify-Write

Each of these commands are sent to a destination node, which may or may not reply depending on the type of command and the contents of certain command fields. The RMAP protocol defines all operations as posted, which means that the protocol itself does not specify that the source node must wait for an acknowledgment or reply. It is the responsibility of the user application to implement time-outs for missing replies if such functionality is requested [8]. The embedded application developed during this thesis project enables the GR740 to act as a destination node, which means that it will listen for incoming communication and then react and reply in a manner consistent with the RMAP standard. It is assumed to never be the source node, and therefore has no functionality for handling incoming replies. For a detailed graphical representation of how the RMAP commands are handled by the GR740 SoC, refer to Figure 5.4, Figure 5.5, Figure 5.6 and Figure 5.3. The flowcharts shown in the figures are derived from the descriptions in the RMAP specification draft [8] and provide a convenient overview of the protocol. If more detail is desired please see the specification.

In the following subsections each command will be discussed separately, with command structure and functionality in focus. Each command has two different versions to be used for either path addressing or logical addressing (see Section 2.1.3) and only the latter will be treated here since only logical addressing was used during the scope of this project.

The Read Command

The structure of a read command is shown in Figure 2.13. Each field is one byte long, which means that the read command is 16 bytes long excluding the EOP field.



First byte transmitted

Destination Logical Address	Protocol Identifier	Packet Type, Command Source Path Addr Len	Destination Key
Source Logical Address	Transaction Identifier (MS)	Transaction Identifier (LS)	Extended Read Address
Read Address (MS)	Read Address	Read Address	Read Address (LS)
Data Length (MS)	Data Length	Data Length (LS)	Header CRC
EOP			

Last byte transmitted

Bits in Packet Type / Command / Source Path Address Length Byte

MSB … LSB

Reserved = 0	Command = 1	Read = 0	Read = 0	Read = 1 (Ack/No_Ack)	Increment/ No inc. address	Source Path Address Len = 0	Source Path Address Len = 0

Figure 2.13: Structure of a RMAP read command [8].

- **Destination Logical Address:** Has a value between 32 and 254 to identify the destination node.
- **Protocol Identifier:** Identifies which higher level protocol is being used, value is 0x01 for RMAP.
- **Packet Type/Command/Source Path Address Length:** Contains several different fields and the byte structure can be seen at the bottom of Figure 2.13:
  - *Packet Type:* Has a value of 01b to indicate that the packet is a command and not a reply.
  - *Command:* The first bit is the Write/Read bit and has a value of 0 to indicate that the command is a read command. The next bit is the Verify Before Write bit and is 0 since there is no writing of data. The third bit is the Ack/No Ack bit and controls whether the destination node should send a reply back as a response to this command. This bit is set as it makes little sense to not request any data in a read command. The last bit in this field is the Increment/No Increment Address bit. When set this bit causes the read address in the destination node to increase after every byte has been read. If cleared, the read address is not incremented so successive bytes are read from the same memory location.

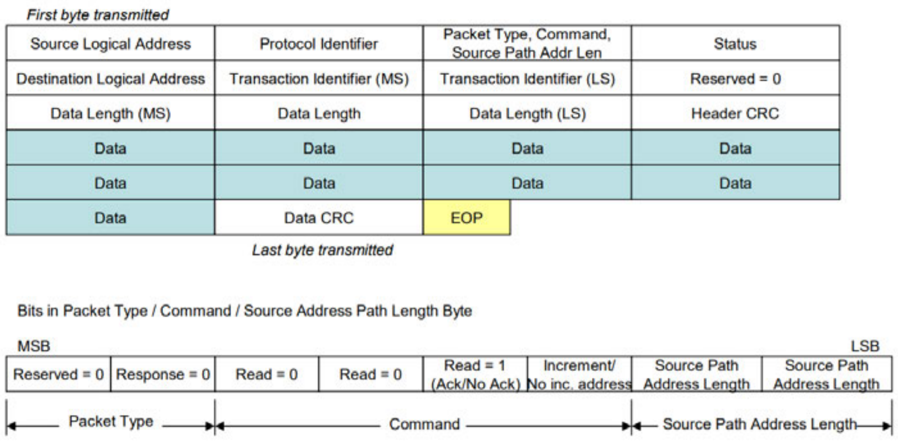
  - *Source Path Address Length:* Not applicable when using logical addressing and is set to 00b.

- **Destination Key:** A value used by the destination node to authorize the command. If the value is different than expected the command will be rejected and an error will be returned to the source node.
- **Source Logical Address:** The logical address of the source node.
- **Transaction Identifier (2x):** A two byte number which uniquely identifies the command in a sequence of commands. The reply will contain the same transaction identifier and the reply can therefore be properly matched to the sent command.
- **Extended Read Address:** An extension of the four Read Address bytes, it holds the eight most significant bits of the memory address to be read from. This extends the 32-bit memory address to 40-bit thus allowing access to one terabyte of memory space in every node.
- **Read Address (4x):** A four byte field specifying the memory location in the destination node at which to read data from.
- **Data Length (3x):** Three bytes which are used to specify how much data is to be read, in bytes.
- **Header CRC:** An eight-bit CRC which is used to confirm that the header has been received correctly before the rest of the command is executed.

The reply to a read command can be seen in Figure 2.14. Many fields are similar to the ones in the original command but there are also a few differences:

- **Status:** A value which contains a status indicating successful completion of command or if an error occurred. For a list of possible codes, see Section 2.2.1.
- **Reserved:** A field which is currently reserved and has a value of 0x00.
- **Data CRC:** Similar to the header CRC but covers the data bytes of the reply. Note that there is always a data CRC even if no data is sent back with the reply.

It is also worth noting that the semantics regarding source and destination does not change in the destination node. The command initiating node is still referred to as the source node, and the other as the destination node even from the destination node's perspective.



First byte transmitted

Source Logical Address	Protocol Identifier	Packet Type, Command, Source Path Addr Len	Status
Destination Logical Address	Transaction Identifier (MS)	Transaction Identifier (LS)	Reserved = 0
Data Length (MS)	Data Length	Data Length (LS)	Header CRC
Data	Data	Data	Data
Data	Data	Data	Data
Data	Data CRC	EOP	

Last byte transmitted

Bits in Packet Type / Command / Source Address Path Length Byte

MSB LSB

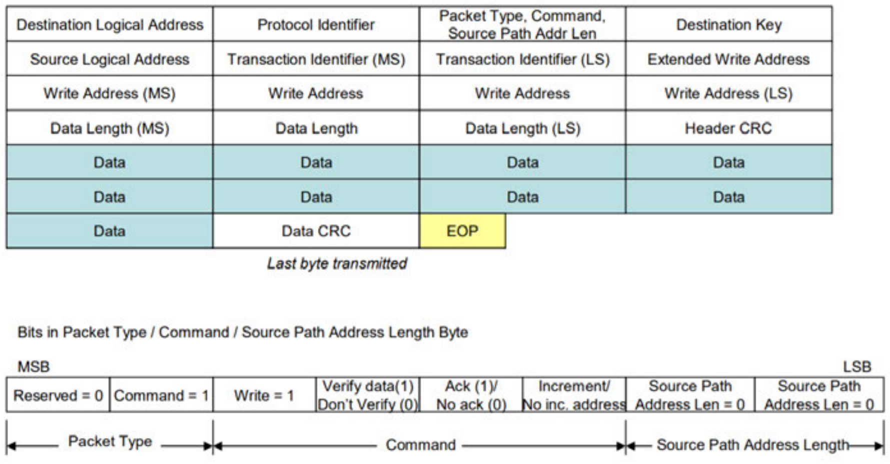
Reserved = 0	Response = 0	Read = 0	Read = 0	Read = 1 (Ack/No Ack)	Increment/ No inc. address	Source Path Address Length	Source Path Address Length

Packet Type — Command — Source Path Address Length

Figure 2.14: Structure of a RMAP read reply [8].

The Write Command

As seen in Figure 2.15 the RMAP write command shares several fields with the read command. The differences will be discussed here.



Destination Logical Address	Protocol Identifier	Packet Type, Command, Source Path Addr Len	Destination Key
Source Logical Address	Transaction Identifier (MS)	Transaction Identifier (LS)	Extended Write Address
Write Address (MS)	Write Address	Write Address	Write Address (LS)
Data Length (MS)	Data Length	Data Length (LS)	Header CRC
Data	Data	Data	Data
Data	Data	Data	Data
Data	Data CRC	EOP	

Last byte transmitted

Bits in Packet Type / Command / Source Path Address Length Byte

MSB LSB

Reserved = 0	Command = 1	Write = 1	Verify data(1) Don't Verify (0)	Ack (1)/ No ack (0)	Increment/ No inc. address	Source Path Address Len = 0	Source Path Address Len = 0

Packet Type — Command — Source Path Address Length

Figure 2.15: Structure of a RMAP write command [8].

- **Packet Type/Command/Source Path Address Length:**
  - *Command:* The Write/Read bit is set to 1 to indicate that this is a write command. The Verify Before Write bit controls if the destination node is to check that the data CRC is valid before attempting to write it to the desired memory location. If set, then the entire command needs to be buffered somewhere and if the command is too big for the available buffer space this can lead to a verify buffer overflow error. This process is shown in Figure 5.5. If the bit is not set, the data CRC is checked after the data has been written to memory. This is mainly used as a protective measure when writing to configuration registers. The Ack/No Ack bit can also be either 0 or 1 depending on if a reply is desired. If it is cleared then the source node will not be informed when an error occurs. The Increment /No increment address bit works in a similar way as in the read command, however here it controls if the data is written to consecutive memory locations (when set) or if the data bytes are written to the same location repeatedly (when cleared).
- **Extended Write Address:** Just as with the extended read address, this field extends the accessible memory space from 32 to 40 bits.
- **Write Address (4x):** The memory address at which to write the data to.
- **Data Length (3x):** The length of the data to be written, in bytes. This will be verified at the destination node before writing by comparing this number to the bytes sent in the packet.
- **Data CRC:** An eight-bit CRC used by the destination node to verify that the data was transmitted correctly.

The format of a write reply is simple and consists of only eight fields. It is demonstrated in Figure 2.16.

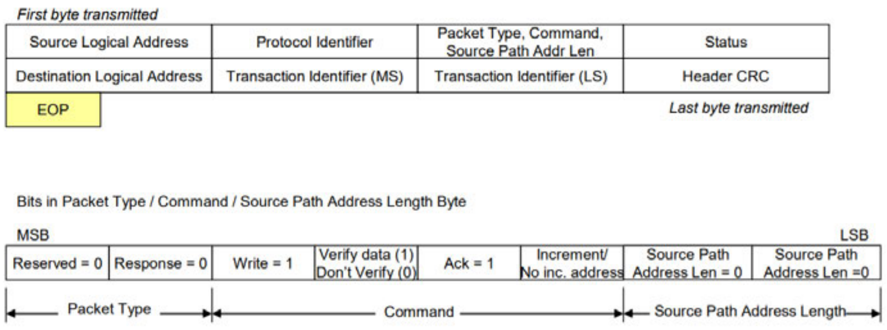

First byte transmitted
Source Logical Address
Protocol Identifier
Packet Type, Command, Source Path Addr Len
Status
Destination Logical Address
Transaction Identifier (MS)
Transaction Identifier (LS)
Header CRC
EOP
Last byte transmitted
Bits in Packet Type / Command / Source Path Address Length Byte
MSB
LSB
Reserved = 0
Response = 0
Write = 1
Verify data (1) Don't Verify (0)
Ack = 1
Increment/ No inc. address
Source Path Address Len = 0
Source Path Address Len =0
Packet Type
Command
Source Path Address Length

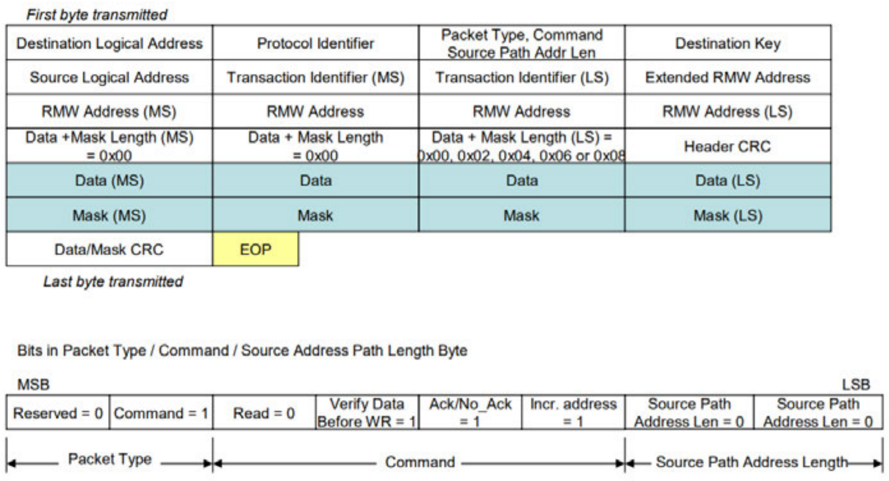
Figure 2.16: Structure of a RMAP write reply [8].

The Read-Modify-Write Command

The Read-Modify-Write (RMW) command is the most complex of the three commands. It performs both the read and write action whilst providing a masking mechanism for the write action. It causes the destination node to first read some data from a desired location and then it uses the mask to decide which data should be written to the same location. Finally, it sends a reply containing the data which was read in the first step. Exactly how the mask is used is up to the user application. The RMAP specification provides an example following the formula:

$$\text{Written Data} = (\text{Mask } \textbf{AND} \text{ Command Data}) \textbf{ OR } (\textbf{NOT} \text{ Mask } \textbf{AND} \text{ Read Data})$$

This is also the scheme which was used in this project. It is performed bit-wise and causes the written data to come from the sent data if the corresponding mask bit is set, otherwise it will remain the same as before the command.



First byte transmitted

Destination Logical Address	Protocol Identifier	Packet Type, Command Source Path Addr Len	Destination Key
Source Logical Address	Transaction Identifier (MS)	Transaction Identifier (LS)	Extended RMW Address
RMW Address (MS)	RMW Address	RMW Address	RMW Address (LS)
Data +Mask Length (MS) = 0x00	Data + Mask Length = 0x00	Data + Mask Length (LS) = 0x00, 0x02, 0x04, 0x06 or 0x08	Header CRC
Data (MS)	Data	Data	Data (LS)
Mask (MS)	Mask	Mask	Mask (LS)
Data/Mask CRC	EOP		

Last byte transmitted

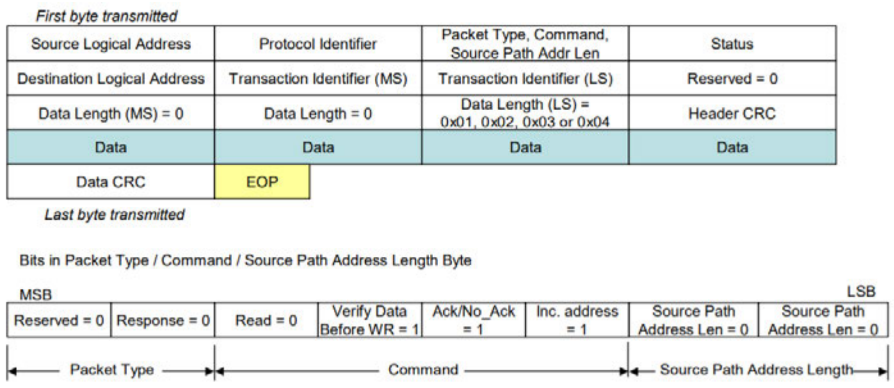
Bits in Packet Type / Command / Source Address Path Length Byte

MSB ... LSB

Reserved = 0	Command = 1	Read = 0	Verify Data Before WR = 1	Ack/No_Ack = 1	Incr. address = 1	Source Path Address Len = 0	Source Path Address Len = 0
Packet Type		Command				Source Path Address Length	

Figure 2.17: Structure of a RMAP RMW command [8].

As seen in Figure 2.17 most fields of the RMW command are the same or similar as in the previous two commands. The main difference is the addition of the mask bytes. There must be as many mask bytes as data bytes, and the Data/Mask Length field can only have values 0x00, 0x02, 0x04, 0x06 or 0x08. All other values in this field are invalid. This value specifies the total number of bytes in the combined data and mask fields. For example, if the value is 0x08 there are four data bytes and four mask bytes. The data/mask CRC appended at the end covers both the data and mask bytes, and just as with the data CRC in the write command it is always present.



First byte transmitted

Source Logical Address	Protocol Identifier	Packet Type, Command, Source Path Addr Len	Status
Destination Logical Address	Transaction Identifier (MS)	Transaction Identifier (LS)	Reserved = 0
Data Length (MS) = 0	Data Length = 0	Data Length (LS) = 0x01, 0x02, 0x03 or 0x04	Header CRC
Data	Data	Data	Data
Data CRC	EOP		

Last byte transmitted

Bits in Packet Type / Command / Source Path Address Length Byte

MSB ... LSB

Reserved = 0	Response = 0	Read = 0	Verify Data Before WR = 1	Ack/No_Ack = 1	Inc. address = 1	Source Path Address Len = 0	Source Path Address Len = 0
Packet Type		Command				Source Path Address Length	

Figure 2.18: Structure of a RMAP RMW reply [8].

For more information on the performed error checks and sequence of actions defined by the standard please see the RMAP specification [8] or Figure 5.4, Figure 5.5, Figure 5.6 and Figure 5.3 for a graphical overview.

RMAP Error Codes

The RMAP standard defines some 8-bit codes for error and status reporting in replies. Depending on the success of the command, the destination node will choose a value as appropriate and set this as the value of the status field in the reply. This enables the source node to be informed regarding the success of the operation. All the specified error codes can be found in Table 2.2

Table 2.2: RMAP error codes [8]

Error Code	**Error**	**Error Description**
0	Command executed successfully	
1	General error code	When the detected error does not fit into the other error cases
2	Unused RMAP packet type or command code	The packet type has an invalid value
3	Invalid destination key	The key did not match the value expected by the user application
4	Invalid data CRC	The data CRC is incorrect
5	Early EOP	The packet was shorter than was declared in the data length field
6	Cargo too large	The packet was longer than declared in the data length field
7	EEP	An EEP marker was detected. Indicates some sort of communication failure.
8	Reserved	
9	Verify buffer overrun	The Verify Before Write bit is set but there is not enough buffer space to hold the entire command while calculating the data CRC
10	RMAP Command not implemented or not authorized	The destination node did not authorize the requested operation.
11	RMW data length error	The data length of a RMW command is invalid
12	Invalid destination logical address	The destination logical address was not the value expected by the application
13-255	Reserved	All unused error codes are reserved

2.3 Real-Time Operating Systems

The embedded application developed during this thesis project uses a real-time operating system (RTOS). A RTOS is very useful for development of embedded code when several different tasks need to be performed in parallel. In this case, the use of the RTOS is not justified by the complexity of the developed code but rather by the fact that the RTEMS Cross Compiler (RCC) is the development platform. RCC includes and relies on a RTOS called Real-Time Executive for Multiprocessor Systems (RTEMS) and the examples and instructions for development is based on the use of this RTOS. Therefore, even though the developed RMAP driver amounts to one single task, which in itself does not warrant a RTOS, it is written using the RTEMS kernel. An advantage of this is that future extensions to this project can leave the RMAP driver task as it is and add other tasks to run in parallel to the driver, whilst taking proper care of task priorities and communication between them if necessary.

The following subsection gives a brief introduction to RTEMS. It does not aim to be an introduction to RTOS concepts in general. If the basic concepts of real-time operating systems seem new, it is strongly recommended to read up on these before continuing. The RTEMS C user guide [11] introduces the benefits of using a RTOS in it's introductory chapter.

2.3.1 RTEMS

RTEMS is a real-time executive which provides a high performance environment for embedded applications. In the RTEMS C user guide support for the following features are highlighted:

- Multitasking
- Homogeneous and heterogeneous multiprocessor systems
- Event-driven, priority based, preemptive scheduling
- Optional rate monotonic scheduling
- Intertask communication and synchronization
- Priority inheritance

- Responsive interrupt management
- Dynamic memory allocation
- High level of configurability

RTEMS is written to act as a bridge between the target hardware and the application dependent code by isolating the hardware dependencies in board support packages (BSPs) and providing an interface to the application code. This allows the developer to use functions from the interface without having to worry about the low-level implementation, as long as the correct BSP is used. The architecture of a RTEMS application can be visualized as seen in Figure 2.19.

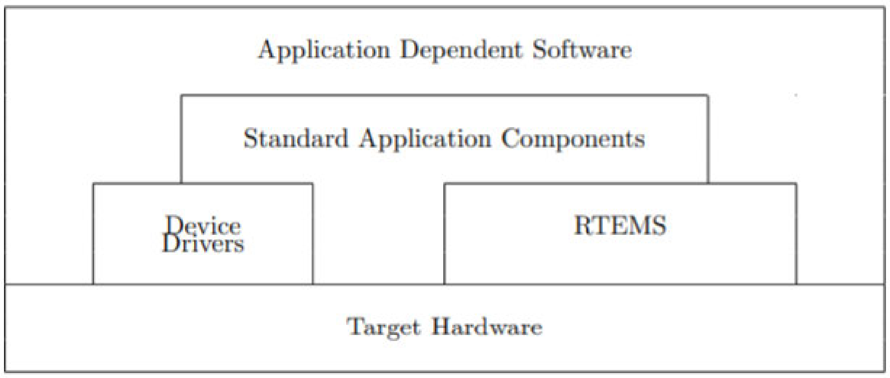

Application Dependent Software
Standard Application Components
Device Drivers
RTEMS
Target Hardware

Figure 2.19: Architecture of a RTEMS application [11].

The interface to the application code is formed by grouping logical sets of functions into what RTEMS calls *managers*. The application can then use requested functionality from the relevant managers, each of which is described in detail in the RTEMS C user guide. Functions which are used by several managers (such as functions dealing with e.g. scheduling) are grouped together in the *executive core* which is accessible to all managers[11]. The concept of managers and the core is displayed in Figure 2.20.

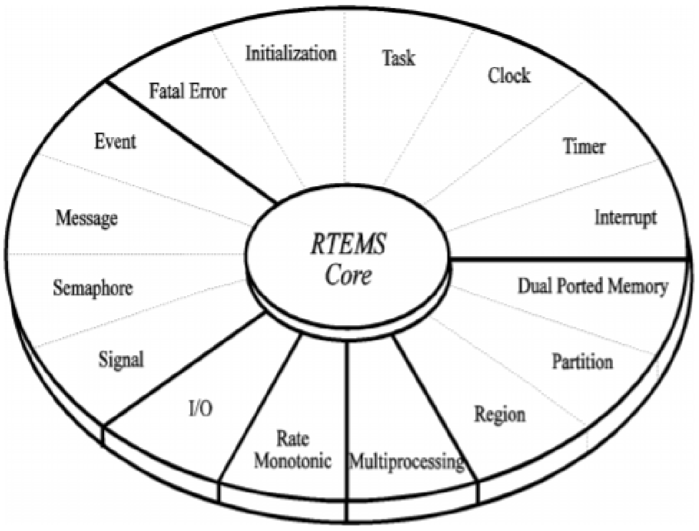


Figure 2.20: Internal architecture of RTEMS [11].

When navigating the RTEMS C user guide it is recommended to read chapters 1-5 for a detailed introduction to important RTEMS concepts, Chapter 22 for a guide on system configuration and Chapter 25 for an example application. A majority of the remaining chapters deal with each of the available managers (as seen in Figure 2.20) and can be read depending on the project. For example, if no semaphores are to be used Chapter 9 is of little importance.

2.4 RTEMS Cross Compiler

The RTEMS Cross Compiler (RCC) is a multi-platform development system. It is based on tools developed by the RTEMS community and Cobham Gaisler, as well as freely available tools from the GNU family of development tools. It consists of the following packages:

- GCC 7.2.0 or LLVM/Clang 7.0.0 C/C++ compiler
- GNU binary utilities 2.29
- RTEMS 5.0 C/C++ real-time kernel with precompiled BSPs for LEON2/3/4 and ERC32

- Newlib 2.5.0 standalone C library
- GDB 6.8 SPARC cross-debugger

RCC is the central software component used for the development of the project code. As such, the RCC user manual is a very important document for a proper understanding of how to perform this type of development. It is recommended to begin by reading chapters 1-4 which contains general instructions on usage. It also contains some of the installation instructions found in Chapter 3.2. Chapters 5 and 6 of the manual can be useful for gaining a deeper understanding of how hardware and bus drivers are handled by the so-called *driver manager*. In short, the driver manager maintains bus and device drivers in a tree structure where each bus has a linked list of devices present on that bus. The devices contain information about their respective drivers, and if a device is a bridge to a child-bus it can register this bus which in turn will contain a linked list on the devices on that bus [3]. These chapters are of great importance when developing custom hardware which shall run software developed using RCC. However, they are not vital to development on the GR-CPCI-GR740 development board since it uses the GR740 SoC for which there is a precompiled BSP included in RCC. The GR740 BSP contains the device drivers needed for operation and therefore no extra drivers need to be registered by the developer.

The remaining chapters of the user manual act as a reference on the drivers for different hardware devices. They contain descriptions on how to control and access the devices through the application code. For development using SpaceWire the central chapters are Chapter 18 and Chapter 20, and an overview of the information in these chapters is given in the following subsections. Please note that the SpaceWire driver described in Chapter 19 is planned to be deprecated and replaced by the packet driver described in Chapter 18 [3]. This is why the packet driver is chosen to control the SpaceWire device in this project.

It should also be noted that RCC already contains an RMAP driver which relies on RMAP hardware in the SpaceWire interface (see Section 2.5). Since the goal of this project is to implement a RMAP software driver all built-in RMAP handling capability is disabled. If the built-in RMAP driver is to be used, see Chapter 7 of the user manual.

2.4.1 GRSPW SpaceWire Packet Driver

The GRSPW SpaceWire packet driver provides the developer with an API to directly configure and control a SpaceWire interface. It replaces an older SpaceWire driver which used standard UNIX file procedures (such as open(), read(), ioctl() etc.) to receive or transmit a single packet at a time. The new driver uses *Direct Memory Access* (DMA) and supports the use of multiple DMA channels. It is divided up into two major parts, the device interface and the DMA channel interface. The DMA concept and the two driver parts are discussed in the following subsections.

DMA

DMA is used to facilitate the transfer of data between two sections of memory or, as in this case, between an I/O device and memory. The traditional approach to moving data is letting the processor copy each byte into its intended destination. For each such operation the processor must fetch the next instruction, decode it, read in data from the source (be it memory or an I/O device), fetch the next instruction, decode it as well and then store the read value before this cycle repeats for every byte to be transferred. DMA provides an alternative which allows for high-speed transfers of large data blocks by bypassing the processor. The processor is bypassed by one or several DMA channels, going directly from the I/O device to memory. The data is then written or read directly to or from memory under control of the DMA controller (DMAC). During transfers, the DMAC needs control over the data and address buses without the processor interfering. The simplest way to implement this is by letting the DMAC signal the processor to suspend its operation whilst the transfer is being done. This means that DMA does not necessarily allow the processor to continue operation in parallel to the data transfer, but the loss of overhead from the instruction fetching and decoding as well as avoiding the extra copying means that the total time of the transfer can be decreased.

There are four basic types of DMA:

- Standard block transfer
- Demand mode transfer
- Fly-by transfer
- Data-chaining transfer

The type that is relevant for this project is the last one, the data-chaining transfer. This transfer type works by giving the DMAC a linked list of so-called *descriptors*. Each descriptor contains (at least) a byte count, source address, destination address and a pointer to the next descriptor. The DMAC gets a pointer to the beginning of the list and starts transferring the data from the source address. When the byte count is reached the next descriptor is loaded and a new transfer starts. This continues until a NULL pointer is reached [14]. This is the basic functionality of the SpaceWire DMA interface used in this project.

Device interface

The SpaceWire device interface handles everything which is not related to the DMA transfers. This includes initialization during system start-up to get the device into a known state, link control to chose behavior on link errors, node address configuration, time code handling and more. To better understand the device interface it is useful to understand the SpaceWire hardware interface which is described in Section 2.5.

An overview of the functions from the interface available to the developer is given in Table 2.3. This is an convenient way to get an idea of what the device interface provides. Each of these functions are documented in detail in Chapter 18 of the RCC user manual.

Table 2.3: Device API functions

Function	**Description**
int grspw_dev_count(void)	Retrieve number of GRSPW devices registered to the driver.
void *grspw_open(int dev_no)	Opens a GRSPW device. The GRSPW device is identified by index. The returned value is used as input argument to all functions operating on the device.
int grspw_close(void *d)	Closes a previously opened device. All DMA channels must have been stopped and closed by the user prior to calling this function.
void grspw_hw_support(void *d, struct grspw_hw_sup *hw)	Read hardware capabilities of GRSPW device.
void grspw_link_ctrl(void *d, int *options, int *stscfg, int *clkdiv)	Read and configure link interface settings.
spw_link_state_t grspw_link_state(void *d)	Read and return current SpaceWire link status.
unsigned int grspw_link_status(void *d)	Reads and returns the current value of the GRSPW status register.
void grspw_link_status_clr(void *d, unsigned int mask)	Clear bits in the GRSPW status register.
void grspw_addr_ctrl(void *d, struct grspw_addr_config *cfg)	Always read and optionally set the node addresses configuration.
void grspw_tc_ctrl(void *d, int *options)	Always read and optionally set TimeCode settings of GRSPW device.
void grspw_tc_tx(void *d)	Generates a TimeCode Tick-In.
void grspw_tc_isr(void *d, void (*tcisr)(void *data, int tc), void *data)	Assigns a Interrupt Service Routine (ISR) to handle TimeCode interrupt events.
void grspw_tc_time(void *d, int *time)	Optionally writes and always reads the current TimeCode control flags and counter from hardware registers.
int grspw_port_count(void *d)	Reads and returns number of ports that hardware supports.
int grspw_port_ctrl(void *d, int *port)	Always read and optionally set port control settings of GRSPW device.
int grspw_port_active(void *d)	Reads and returns the currently actively used SpaceWire port.
int grspw_rmap_support(void *d, char *rmap, char *rmap_crc)	Reads the RMAP hardware support of a GRSPW device.
int grspw_rmap_ctrl(void *d, int *options, int *dstkey)	Read and optionally write RMAP configuration and SpaceWire destination key value.
void grspw_stats_read(void *d, struct grspw_core_stats *sts)	Reads the current driver statistics collected from earlier events by GRSPW device and driver usage.
void grspw_stats_clr(void *d)	Resets the driver GRSPW device statistical counters to zero.

DMA channel interface

As previously mentioned, the type of DMA transfer used with the GRSPW2 device is data-chaining DMA which uses linked lists of descriptors to move data to the correct destination. The descriptors used by the SpaceWire packet driver are represented by the C structure `grspw_pkt`. Typical usage is for the user to define an own structure with the same fields as `grspw_pkt` in the top and then custom fields at the bottom if needed. Each of these descriptors acts as metadata for exactly one SpaceWire packet. Every descriptor consists of the following fields:

- **next:** A pointer to the next descriptor. The DMA engine will start using the next descriptor as soon as en EOP/EEP marker was received or the received number of bytes matches the value in the **dlen** field.
- **flags:** Controls transmission behavior (e.g should finished transmission trigger interrupt) or indicates status of received packet (e.g. was there a link error during reception).
- **hlen:** Header length. Number of bytes in the header of the SpaceWire packet to be sent. For logical addressing this will be 1, since there is only a one byte address. Has no function for reception since all received data is put at where the **data** pointer points to.
- **dlen:** Number of bytes to send as data.
- **data:** Pointer to the data to be sent.
- **hdr:** Pointer to the header bytes.

Notice the division of header and data. The header field does not need to be used at all, one can put the entire packet including the destination address at the place pointed to by the data pointer and set hlen to 0. However, it can be convenient to deal with the address of a packet and the data it contains separately, especially if path addressing is used. This is what the two fields allow. For reception the header pointer and hlen have no use, all incoming data is put at the position pointed to by data and it is up to the user application to interpret the received bytes.

The linked lists of descriptors are referred to by the driver as queues. The organization of one such queue is visualized in Figure 2.21. Every queue maintains a counter of how many descriptors is currently in the list.

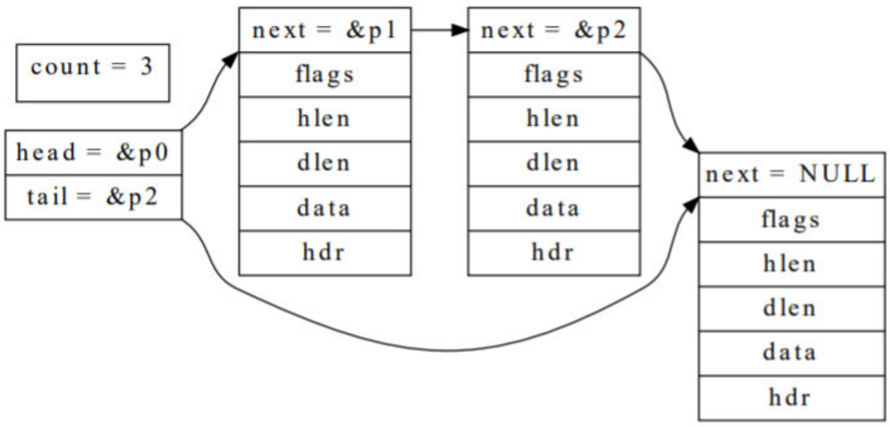


Figure 2.21: SpaceWire packet driver descriptor queue organization [3].

A very important concept to understand when using the SpaceWire packet driver is the use of several descriptor queues per DMA channel and direction. Every DMA channel has six different queues, three for transmission and three for reception:

- **RX READY:** List of descriptors that are pointing to free data buffers initialized by the user. This can be seen as the first step of preparing for reception, where the user has made sure there is some space for the incoming packets.
- **RX SCHED:** In order to receive packets, descriptors from **RX READY** are moved to this queue. Once in this queue, the descriptors will be used for any incoming data. The descriptors should not be directly accessed by the user here.
- **RX RECV:** List of descriptors pointing to received data.
- **TX SEND:** Transmission equivalent of **RX READY**. However, here the descriptors are not pointing to free space for incoming data, but rather to the data that the user application wants to send.
- **TX SCHED:** When ready to send, descriptors from **TX SEND** are moved to this queue. The packets will be sent when possible.
- **TX SENT:** List of descriptors pointing to sent packets. The flag field has been updated to indicate any detected errors that occurred during transmission.

In other words, the flow of descriptors is always the same: *USER* → *RX READY* → *RX SCHED* → *RX RECV* → *USER* for reception and *USER* → *TX SEND* → *TX SCHED* → *TX SENT* → *USER* for transmission.

The developer has a choice regarding the movement of descriptors between queues. The packet driver supports both blocking and polling modes. In polling mode, the developer is responsible for calling DMA reception and transmission routines at regular intervals. These routines are used to both retrieve or send new data, as well as trigger movement of descriptors between the queues. In blocking mode, the developer can use functions that wait for a condition to be fulfilled, e.g. wait until there are more (or less) descriptors than a certain threshold in one or more queues. In that case, the so-called *work-task* is responsible for all queue handling. The work-task is triggered by DMA interrupts and so it reacts to sent or received data and moves descriptors between queues accordingly. Every time the work-task completes the conditions in the blocking functions are evaluated. It is also possible to provide a time-out to the blocking functions. The developer can control the priority of the work-task by setting the attribute `int grspw_work_task_priority`. The default value is 100, and if polling mode is wished this value should be set to -1. This causes the work-task to never be created. The RCC user manual notes that care must be taken so that the priority of the work-task is high enough to not be constantly blocked by higher level tasks always being ready. This would make the transmission and reception appear to deadlock since no descriptors would be moved between the queues.

As it would be waste of resources - both time and memory - to constantly define new data buffers for reception and transmission, it is possible to re-use packet descriptors. For reception, this means that a descriptor in the RX RECV queue pointing to a received packet can be given back to the RX READY queue once the user application has either saved the received data or is finished using it. The next time this descriptor is used the same memory space will be used. The same can be done for transmission, where descriptors are moved from TX SENT to TX SEND. This is not done by the work-task since it does not know when the user application is finished with the data [3].

An overview of the available functions from this part of the API can be seen in Table 2.4. For this project, the blocking approach was used in order not to waste CPU cycles in polling mode.

Table 2.4: DMA API functions

Function	**Description**
void *grspw_dma_open(void *d, int chan_no)	Opens a DMA channel of a previously opened GRSPW device.
void grspw_dma_close(void *c)	Closes a previously opened DMA channel. Channel must be stopped first.
int grspw_dma_start(void *c)	Starts DMA operation.
void grspw_dma_stop(void *c)	Stops DMA operation.
int grspw_dma_rx_recv(void *c, int opts, struct grspw_list *pkts, int *count)	Get pointer to received RX packets.
int grspw_dma_rx_prepare(void *c, int opts, struct grspw_list *pkts, int count)	Add RX free packet buffers for reception.
void grspw_dma_rx_count(void *c, int *ready, int *sched, int *recv)	Get current number of packets in respective RX queue.
int grspw_dma_rx_wait(void *c, int recv_cnt, int op, int ready_cnt, int timeout)	Wait for queue conditions to be met. Blocking.
int grspw_dma_tx_send(void *c, int opts, struct grspw_list *pkts, int count)	Schedules a list of packets for transmission at some point in future.
int grspw_dma_tx_reclaim(void *c, int opts, struct grspw_list *pkts, int *count)	Reclaim TX packet buffers that have previously been sent or scheduled for transmission.
void grspw_dma_tx_count(void *c, int *send, int *sched, int *sent)	Get current number of packets in respective TX queue.
int grspw_dma_tx_wait(void *c, int send_cnt, int op, int sent_cnt, int timeout)	Wait for queue conditions to be met. Blocking.
int grspw_dma_config(void *c, struct grspw_dma_config *cfg)	Set the DMA channel configuration options as described by the input arguments.
void grspw_dma_config_read(void *c, struct grspw_dma_config *cfg)	Copies the DMA channel configuration to user defined memory area.
void grspw_dma_stats_read(void *c, struct grspw_dma_stats *sts)	Reads the current driver statistics collected from earlier events on a DMA channel.
void grspw_dma_stats_clr(void *c)	Resets one DMA channel's statistical counters.

2.4.2 GRSPW SpaceWire Router Driver

The other driver that is central for setting up communication using SpaceWire is the router driver. This driver is used for configuring the router hardware, setting up the routing table (see Section 2.1.3) and managing time codes. It can also enable or disable links, collect statistics and detect errors on different ports. The normal use case is to open the router, configure the settings and then set up the routing table. Just like in the case of the SpaceWire device interface, it is beneficial to have some understanding of the router hardware to better understand the driver. The router hardware is described in Section 2.5.1.

The driver provides getters and setters for many individual router configuration registers and therefore the list of functions available is quite long. For a quick overview of some of the most important parts of the driver, please see Table 2.5. For a more detailed documentation of the interface see Chapter 20 of the RCC user manual.

Table 2.5: Partial overview of the GRSPWROUTER driver interface

Function	**Description**
void * router_open(int dev_no)	Opens a router with the given dev_no index.
int router_close(void * router)	Closes a router.
int router_hwinfo_get(void * router, struct router_hw_info * hwinfo)	Get the hardware info of a given router.
int router_config_set(void * router, struct router_config * cfg)	Set the configuration of a given router.
int router_reset(void * router)	Reset a given router.
int router_routing_table_set(void * router, struct router_routing_table * rt)	Set the routing table of a given router.
int router_route_set(void * router, struct router_route * route)	Set a specific route of a given router.
int router_port_ioc(void * router, int port, struct router_port * cfg)	Configure a given port of a given router.
int router_port_enable(void * router, int port)	Enable data transfers to and from a given port of a given router.
int router_port_link_status(void * router, int port)	Get the link status from a given port of a given router.
int router_port_start(void * router, int port)	Start link interface FSM from a given port of a given router.
int router_tc_enable(void * router)	Enable time-codes in a given router.

2.5 GR-CPCI-GR740 Development Board

GR-CPCI-GR740 is the name of the development board used during this project for the validation of functionality of the developed code. This section contains an introduction to the board and the relevant on-board components.

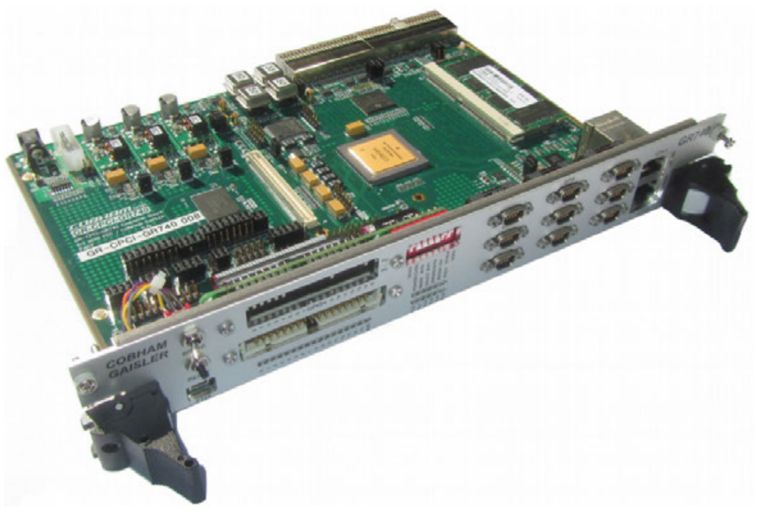


Figure 2.22: The GR-CPCI-GR740 development board [1].

The board can be seen in Figure 2.22. Some of the main components of the board is the GR740 SoC , a CPCI interface for connection to other hardware, configurable SDRAM (48 or 96 bit) and 64 Mbit parallel boot flash. In addition to this it also contains the required interface circuits for the connectors available on the front panel of the board, such as the FTDI to USB interface, the Ethernet interface and the eight port SpaceWire interface used in this project. The front panel also has a number of DIP switches for manual configuration. How to set these into a default state (and how to configure the board for SpaceWire communication) is described in Section 3.1.

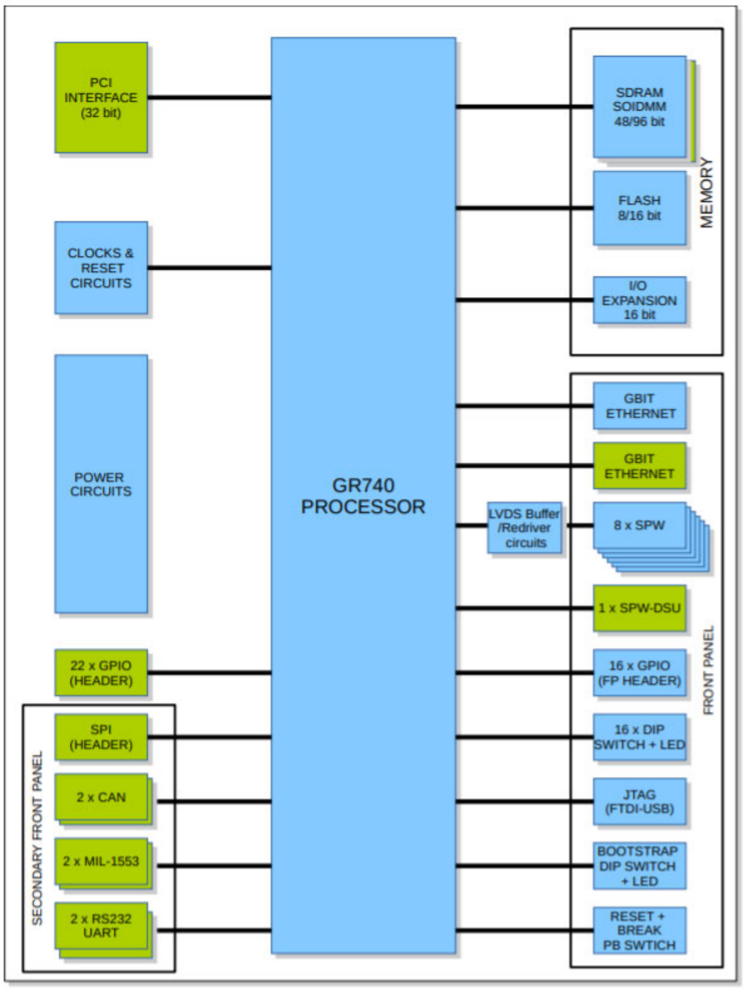


Figure 2.23: Block diagram of the GR-CPCI-GR740 [1].

Debugging is supported via JTAG, Ethernet or SpaceWire (port 1).

2.5.1 GR740

The most critical part of the development board is the GR740 SoC which contains the quad-core LEON4 SPARC V8 processor on which the embedded code is executed. It also contains the SpaceWire router and SpaceWire interface hardware developed by Cobham Gaisler. It is radiation-hard, which means that it is resistant to damage or malfunction caused by high levels of radiation. This is necessary due to its intended operational environment, i.e. space, where radiation can pose a big problem for electronic equipment. A block diagram of the SoC can be seen in Figure 2.24.

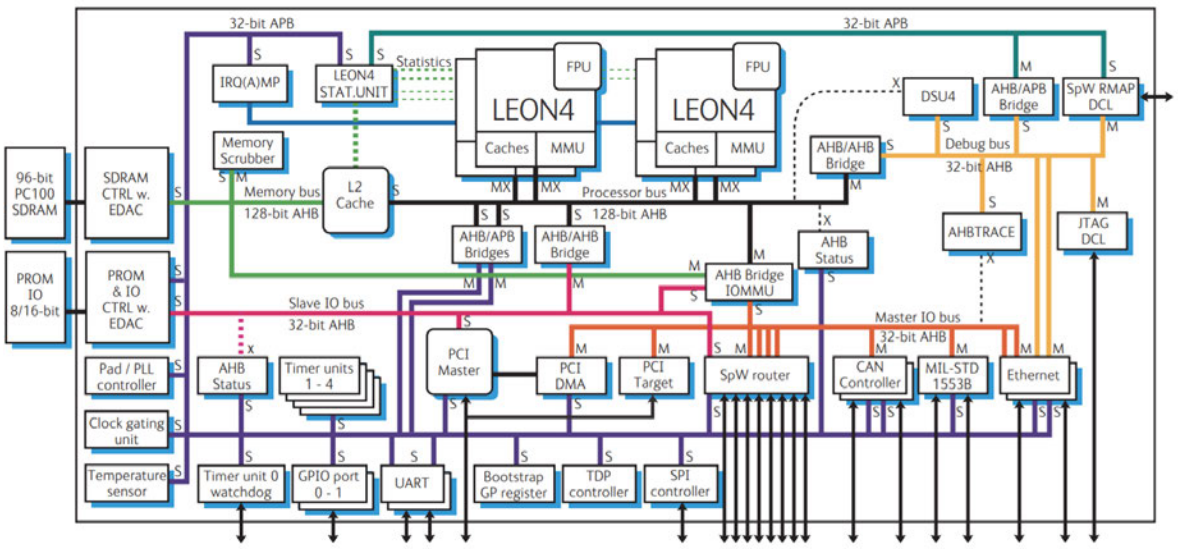


Figure 2.24: Block diagram of the GR740 SoC[1].

In addition to the LEON4 the GR740 contains a set of IP cores developed by Cobham Gaisler and documented in the GRLIB IP core user manual [6]. The IP cores are connected through a nested system of Advanced Microcontroller Bus Architecture (AMBA) buses. This connects back to the driver manager mentioned in Section 2.4, which allows for such a tree structure of buses. The IP cores most relevant for developing a SpaceWire-based application is the SpaceWire router core which goes under the name GRSPWROUTER and contains four SpaceWire devices which are called GRSPW2. The GRSPWROUTER and GRSPW2 cores are presented in the following subsections.

GRSPW2

A block diagram of the GRSPW2 core is seen in Figure 2.25.

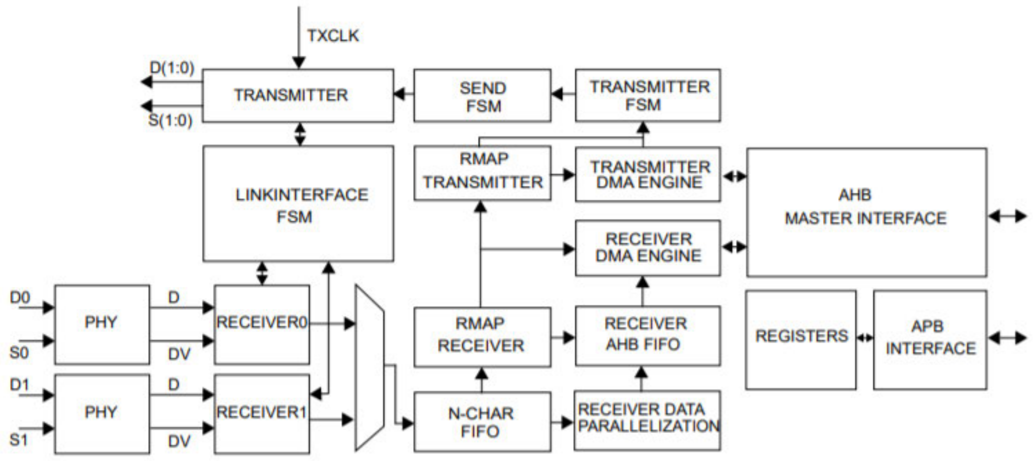


Figure 2.25: Block diagram of the GRSPW2 architecture [6].

The main parts of the core are the link interface, RMAP interface and AMBA interface. Starting from left to right, the core has support for two SpaceWire ports of which only one is active at a given time. The receiver interprets the received SpaceWire character and forwards some status signals to the finite state machine (FSM) which controls the link according to the SpaceWire standard (see Section 2.1.2) and the current configuration. Through the device interface described in Section 2.4.1 the user can configure the device to enable/disable the link, put it in auto-start mode, or choose error conditions which should disable the link, among other things. Received N-chars are stored in a FIFO. If RMAP functionality is enabled the RMAP receiver will formulate the appropriate reply and directly queue it for transmission. The received N-char is stored at the address pointed to in the current descriptor of the receiver DMA engine, via the AMBA interface on the right in Figure 2.25.

GRSPWROUTER

The SpaceWire router core is documented both in the GRLIB IP core user manual [6] and the GR740 user manual [2]. A block diagram of its design is shown in Figure 2.26.

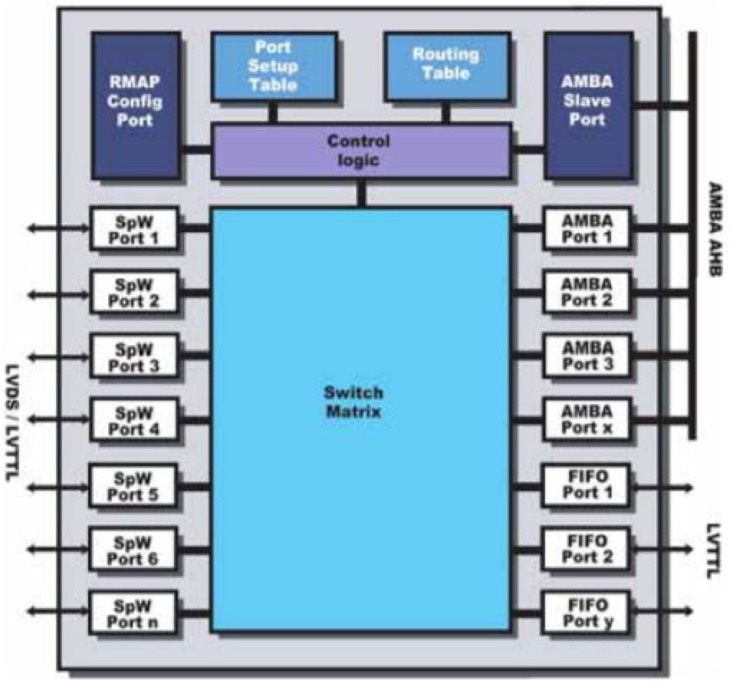


Figure 2.26: Block diagram of the GRSPWROUTER [2].

The routing table controls packet forwarding according to their address and is configured using the router interface provided by RCC. The port setup table controls individual port behavior. An important thing to understand about the router hardware is the port numbering. The configuration port is always port 0 and the SpaceWire ports follow in numbering starting from 1. The AMBA port numbering start where the SpaceWire port numbering end. In the case of the hardware used in this project, where there are eight SpaceWire ports available, this means that the first AMBA port is port 9 [2]. This is important for setting up the routing table, since the AMBA ports are what connects the router to the rest of the GR740. In order for the processor to access incoming SpaceWire communication the routing table needs to be set up to forward packets with the correct address to one of the AMBA ports. The process for doing this is found in Chapter 6.

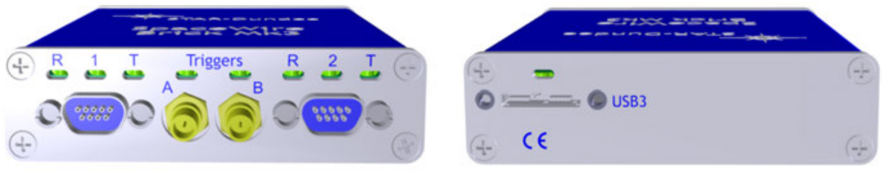
2.6 MK3 SpaceWire Brick

In order to verify the correct functionality of the protocol handling application it is helpful to test how the application reacts to valid RMAP commands. The MK3 SpaceWire brick developed by STAR-Dundee is a piece of additional hardware which can be used with a PC in order to send and receive SpaceWire packets. This is an important part of the debugging process and STAR-Dundee provides an API which can be used to write applications which automates the testing process. This enables the development of a test-bench running on the development PC. The design and implementation of the test-bench is discussed along with the embedded application in Chapter 5 and Chapter 6. A description of test-bench usage is found in Chapter 7.

The following subsections give an introduction to the SpaceWire MK3 brick.

2.6.1 Hardware Description

The SpaceWire MK3 brick can be seen in Figure 2.27. On the front panel there are two SpaceWire ports where SpaceWire cables can connect and form one end of the communication channel between the GR-CPCI-GR740 and the brick. It can be helpful to connect a cable between the ports and use one for input and one for output. This way it is possible to analyze exactly what is sent out by the brick and is especially useful when configuring the brick in routing mode to verify correct transmission. There are also LEDs for each port indicating status of reception and transmission.



(a) Front panel. (b) Rear panel.

Figure 2.27: Physical overview of the SpaceWire MK3 Brick [13].

The SpaceWire MK3 brick has two SpaceWire interfaces, available through port 1 and port 2. They are connected to an internal SpaceWire router which in turn is connected to three channels and a configuration port. The three channels are connected to a USB interface and all communications between the brick and the PC are sent via these channels. Channel 1 and 2 can be used in parallel by the two SpaceWire interfaces, so communication on one port will never block traffic on the other port. Channel 0 is a control channel which allows the user to access the control, configuration and status space of the brick regardless of the current data flow.

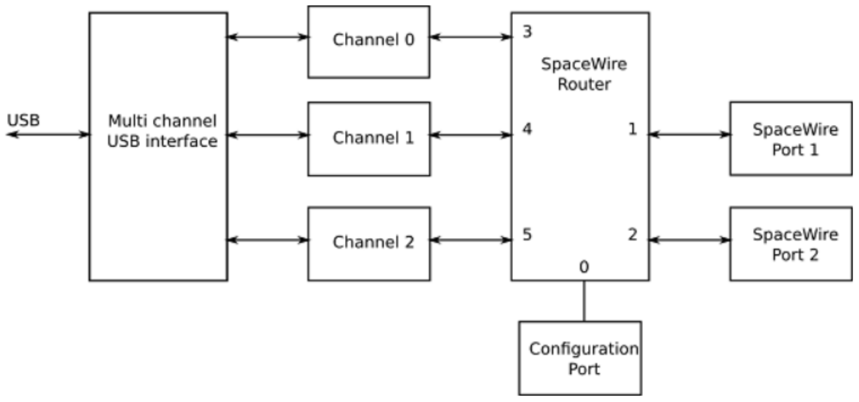


Figure 2.28: Block diagram of the SpaceWire MK3 Brick hardware [13].

The router can be used in two different modes, interface mode and router mode. In router mode, any incoming packet on a port can be routed to another port. In this mode the brick works like a normal SpaceWire router and needs to have a routing table configured for correct operation. Something to be taken into consideration is the optional header deletion. With header deletion enabled the first N-char of the SpaceWire packet is deleted before the packet is forwarded. This can be undesired if the application in the destination node expects the first N-char to be the logical address of the destination node, as is the case in logical addressing mode. The brick router can also be configured to be the time-master of a SpaceWire network and periodically send out time-codes to be distributed throughout the network.

In interface mode, the brick is set up to have a static connection between a SpaceWire port and either channel 1 or 2. All routing functionality is disabled and this type of connection allows for a point-to-point connection between the source and destination node. Therefore, a changing destination address will have no effect on whether the packet arrives at the destination node or not [13].

2.6.2 C++ API

The STAR-System API is provided as a set of libraries and header files intended for use with the C programming language. The C++ API is an extension of this API to provide an object-oriented interface to the STAR-System. The API can be used to configure the brick and e.g. set up the routing table, and send and receive SpaceWire packets or even RMAP commands in a convenient way from a development PC. The

STAR-System documentation contains many helpful example programs and since the test-bench is written using this API some specifics will be discussed in Chapter 6.

The STAR-System API is documented in detail in the API manual [13] which is available after installing the STAR-System software, see Section 3.5.2. Some simple GUI applications which can be used to configure and use the brick are also discussed in that section.

3 Development Environment Setup

As with any software project, it is necessary to set up the correct environment for development and debugging. This section contains instructions for a first time project setup, derived from the relevant documents and datasheets. The instructions found here are not as in-depth as the descriptions found in the referenced documents and the reader is encouraged to look into these in case of problems during the setup process. The instructions are given with SpaceWire based projects in mind but are applicable to development of any type of application on the GR-CPCI-GR740 development board. Also note that the instructions for the software setup are written for installation on a computer running Windows 10 as the operating system. For installation on other platforms please see the referenced documents.

Figure 3.1 gives an overview of the physical setup.

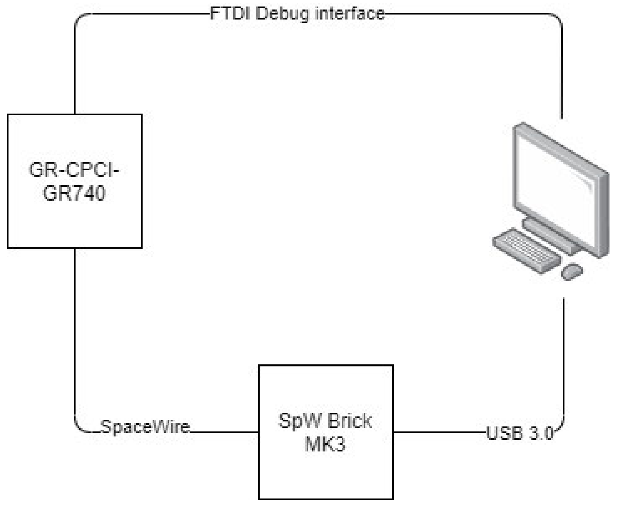


Figure 3.1: Overview of the connections between PC and development board.

3.1 GR-CPCI-GR740 Development Board

This section contains instructions on how to manually configure the settings on the development board in hardware, as well as warnings regarding usage.

3.1.1 Warnings

The development board contains components which can be damaged by electrostatic discharge (ESD). Therefore it is important to observe ESD safe practices when handling the board. When working with the board, use an ESD mat as the work surface. The mat should be connected to ground of a nearby electrical outlet, and you should be connected to conducting wristbands which are in turn connected to the ESD mat. Make sure to always wear the wristbands when touching the development board, as well as making sure the board is in contact with the mat. When the board is not in use it should be stored in a ESD safe container. When moving jumpers or connecting (or disconnecting) cables make sure the board is in an unpowered state. Also make sure the correct input voltage of 5V is used [1].

3.1.2 Settings

At the project outset it is advisable to start with all settings in their default positions. This increases the probability that the various guides and instructions will be helpful. Project specific settings are then set as required. This section contains instructions on how to set the development board's default settings as described in the GR-CPCI-GR740 user manual, and notes on changes needed for this specific project.

There are three parts to consider regarding the board settings; the jumpers, the switches and the plug-on board.

The jumpers Most of the jumpers can be seen in the top left area of the development board, as shown in Figure 3.4. By using them to connect or disconnect different pin pairs we are in effect configuring the hardware. The jumper numbers are printed on the PCB. In Table 3.1 we see the default positions of all jumpers, together with a comment about the effect of that configuration. To fully understand every jumper configuration option the reader is referred to the user manual of the GR-CPCI-GR740 development board, from which this table is taken [1]. During the project described in this document all the jumpers were set in the positions described in Table 3.1 and did not need any further changes.

Table 3.1: Default jumper positions for the GR-CPCI-GR740 [1]

Jumper	**Jumper Setting**	**Comment**
JP1	Connected to Front Panel	Connects front panel RESET and BREAK switch to JP1
JP2	Not installed (4x)	UART1 not connected to FTDI chip
JP3	Not installed (4x)	UART0 not connected to FTDI chip
JP4	1 2 3 4 5 6 7 8	Connects ASIC JTAG to FTDI chip
JP5	1 2 3 4	Connects I2C to FTDI chip
JP6	3 4 5 6 7 8 9 10 11 12 (1 2 open)	Configured for 8 bit Flash memory PROM
JP7	Not installed (4x)	GPIO[7..4]; For SPW DSU install 1 2 and 3 4
JP8	1 2	SPI OB; Install to enable on board SPI circuit
JP9	Installed 1 2	VC0 CLKSEL; XTAL generates 25MHZ clock
JP10	Not installed	VC0 PROG, do not install for parameters to be loaded via I2C
JP11	Jumpers 1,2,13 &14 in position C D Other jumpers in position A B (18x)	Install A B for PROM and C D for alternate I/F functions
JP12	Do not install	+3.3V
JP13	Do not install	+VIN
JP14	Install	I3V3asic
JP15	Install	I2V5
JP16	Install	I1V2
JP17	Open	TESTEN disabled
JP18	Open	PCI bus runs at 33MHz
JP19	Not installed	PCI_RSTN configuration options
JP20	Not installed	VC1 CLKSEL; SD_CLK generates SDCLK's
JP21	Not installed	VC1 PROG, do not install for parameters to be loaded via I2C
JP22	Install	BP CLK (Host Mode) all clocks present on BP
JP23	Install	M66EN; Force backplane to 33MHz
JP24	Not installed	Configuration options for Versaclock PLL ranges

The switches The switches are seen in the black and red areas on the board's front panel, as depicted in Figure 3.5. The eight most left-hand switches in the black area are denoted by the prefix FP-S1, the remaining right-hand eight switches are denoted by FP-S2, and the eight switches in the red area are denoted by FP-S3. The switches are open in the top position, and closed in the bottom position. Please see Figure 3.2 for reference. In Table 3.2 we see the default positions of the switches. As with the jumpers, the reader is referred to the development board's user manual for further explanations regarding the configuration options [1]. For the project described in this document only one switch needed to be changed from the default setting, namely FP-S2-4 which was changed from open to closed in order to enable the on-board SpaceWire router.

Table 3.2: Default switch positions for the GR-CPCI-GR740 [1]

Switch	**GPIO**	**Switch Setting**	**Comment**
FP S1 1 to 6	GPIO[5..0]	Open	GPIO[5..0] Mac address. Default Disable EDCL
FP S1 7 & 8	GPIO[7..6]	Open	SPW Router INT mode. Default Pulled up.
FP S2 1 & 2	GPIO[9..8]	Open	EDCL0 & EDCL 1 Link traffic : EDCL Enabled
FP S2 3	GPIO[10]	Closed	PROM Width 8 bit
FP S2 4	GPIO[11]	Open	'1' >Disables SPW Router
FP S2 5 & 6	GPIO[13..12]	Open	SPW Router ID '11'
FP S2 7	GPIO[14]	Closed	PROM EDAC disabled
FP S2 8	GPIO[15]	Closed	PROM I/O enabled
FP S3 1		Open	DSU Enabled
FP S3 2		Open	MEM CLK is taken from Xtal X2
FP S3 3		Closed	PLL for SYS_CLK enabled (close for 'bypassed')
FP S3 4		Closed	PLL for MEM_CLK enabled (close for 'bypassed')
FP S3 5		Closed	PLL for SPW_CLK enabled (close for 'bypassed')
FP S3 6		Open	Disable PLL lock
FP S3 7		Closed	Ethernet Clock: 100M Ethernet on ETH 0 and ETH1
FP S3 8		Open	WD output does not cause board reset

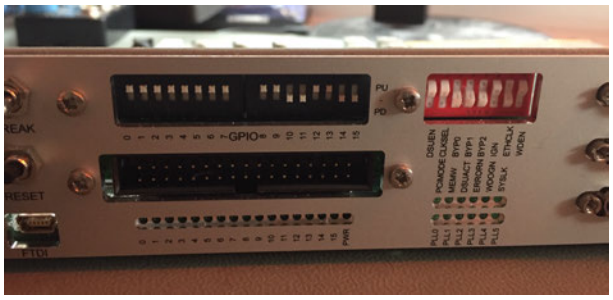


Figure 3.2: The front panel switch configuration used in this project.

The plug-on board The plug-on board can be found on the bottom of the development board and is shown in Figure 3.3. The plug-on board is used to configure the SDRAM memory interface. The GR-CPCI-GR740 board has two SODIMM modules which offer a 96 bit wide SDRAM data interface to the GR740 processor. However, the GR740 processor can operate in different memory modes including full-width (96 bit) and half-width (48 bit) operation. The upper 48 data bits of the SDRAM interface are multi-functional pins which are shared with the PCI and Ethernet_1 interfaces. This means that the user can choose to use the full 96 bit width if no PCI or Ethernet functionality is needed, or use the 48 bit half-width if it is. This is what the plug-on board is used to configure [1]. The settings are summarized in Table 3.3. Within the scope of this project neither PCI or Ethernet is used, so the plug-on board is installed on position J23 in order to use the full memory width.

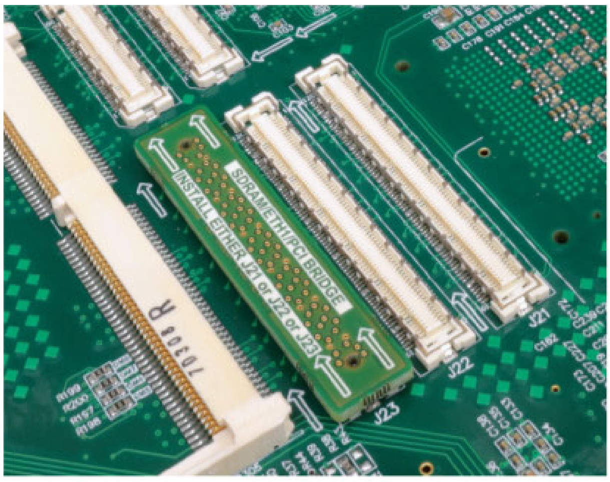


Figure 3.3: The plug-on board in of the three possible positions [1].

Table 3.3: Plug-on board configurations [1]

Plug Position	**Configuration**
J21	48 bit memory + PCI
J22	48 bit memory + ETH1
J23	96 bit memory

3.2 Installing The RTEMS Cross Compiler

The RTEMS Cross Compiler (RCC) is provided for two host platforms, Linux/x86_64 and Windows/x86_64. This section describes the installation of the latter version. RCC is available for download on the website of Cobham Gaisler at www.gaisler.com. Go to *Downloads -> Compilers -> RCC linux, cygwin and mingw binaries -> mingw* and download the file `sparc-rtems-5-gcc-7.2.0-1.3-rc6-mingw.zip` or a later version if release candidate (rc) 6 is no longer available. This is the version that was used during this thesis project and which this setup description is based upon. If any unexpected errors occur during the setup please beware of mentions in the RCC user manual regarding updates in later release candidates. The downloaded zip file must be

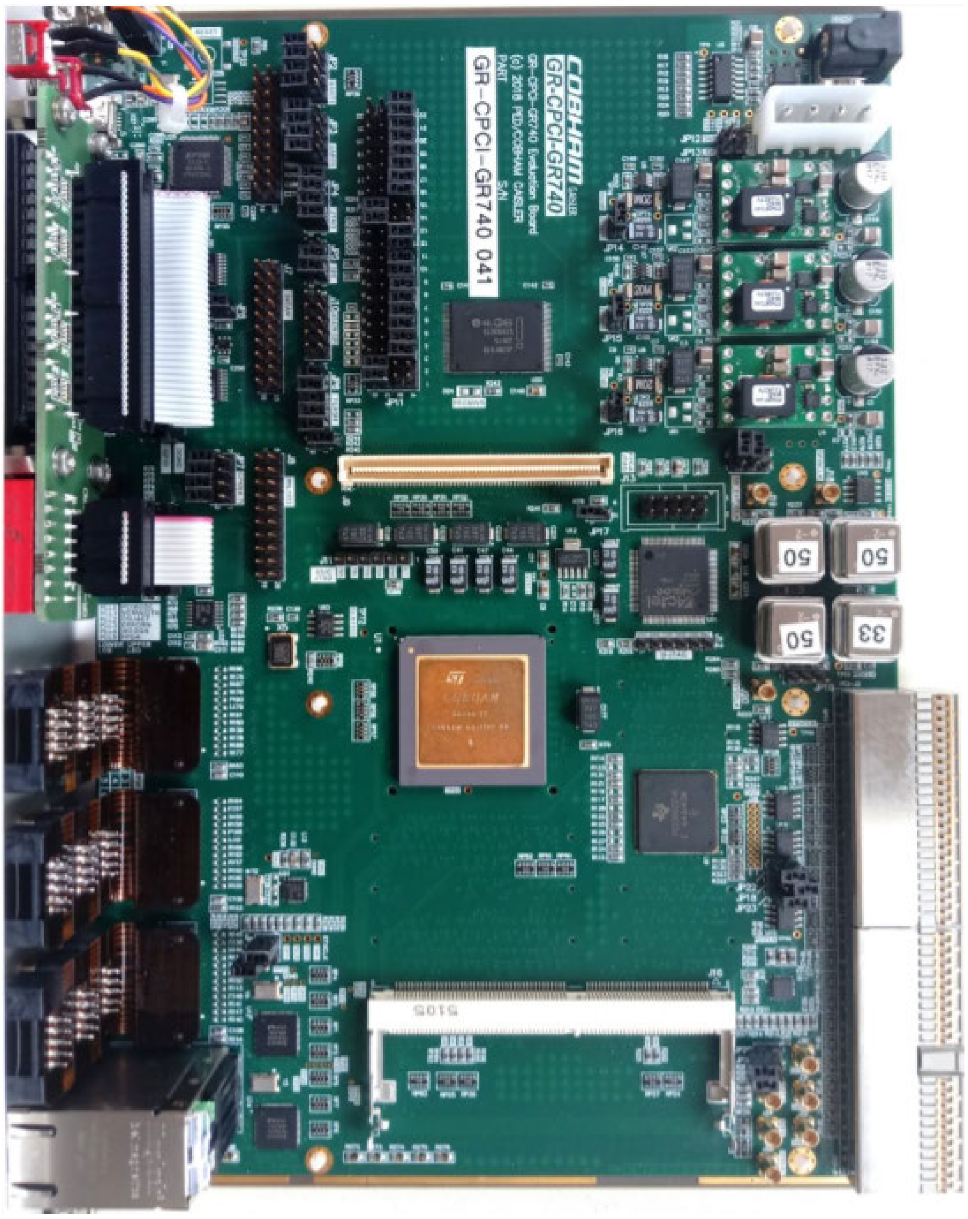


Figure 3.4: Top view of the GR-CPCI-GR740 development board [1].

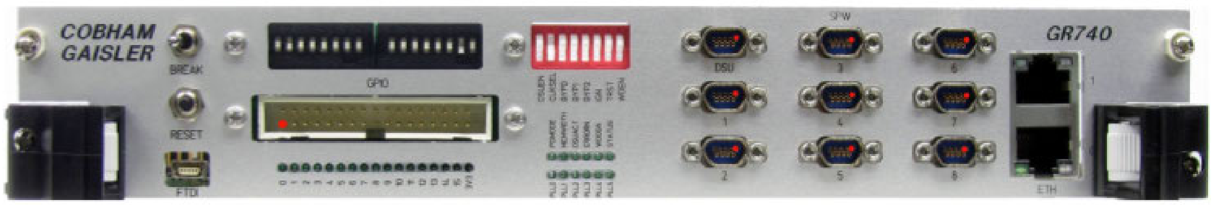


Figure 3.5: Front panel of the GR-CPCI-GR740 development board [1].

extracted to `C:\opt` and should thereby create the directory `C:\opt\rcc-1.3-rc6-gcc`. Now add the directory `C:\opt\rcc-1.3-rc6-gcc\bin` to your system path if you wish to be able to run the toolchain executables from the command line. In Windows 10 this can be done by right clicking on *This PC* in the file explorer and then clicking *Properties ->Advanced system settings->Environment variables.*

RCC needs some basic tools like those found on UNIX systems to function properly. In order to create a UNIX like development environment on Windows the MSYS package can be installed. It is available for download at `www.mingw.org`. The MSYS installation is handled through the MinGW installation manager. If MinGW is already installed simply open the installation manager. Otherwise, download the file `mingw-get-setup.exe` and run it with default settings to install MinGW and open the MinGW installation manager. The MSYS base version must be 2013.07.23 or later [7]. In addition to the MinGW and MSYS base packages (make sure to mark the packages mingw32-base-bin and msys-base-bin for installation), a few additional packages need to be installed.

- msys-findutils
- msys-m4
- msys-perl

In the installation manager, go on *All Packages -> MSYS* and make sure to mark these packages for installation. Only the package components ending in -bin are necessary. Afterwards, click on *Installation -> Apply Changes* to initiate the package installations. After the installation, add the paths `C:\MinGW\bin` and `C:\MinGW\msys\1.0\bin` to the system path.

The directory where the RCC toolchain is installed (`C:\opt\rcc-1.3-rc6-gcc`) must be found at `opt\rcc-1.3-rc6-gcc` from the MSYS environment. This is done by adding the line `C:/opt/rcc-1.3-rc6-gcc /opt/rcc-1.3-rc6-gcc` at the bottom of the file `fstab` found at `C:\MinGW\msys\1.0\etc`. The path to the toolchain binaries (`C:\opt\rcc-1.3-rc6-gcc\bin`) must be added to the MSYS PATH environment variable. This is done by running the MSYS shell (run `msys.bat`) and executing the following command: `export PATH=/opt/rcc-1.3-rc6-gcc/bin:$PATH`. At this point the toolchain installation can be tested by compiling the samples included in the toolchain. From within the MSYS shell, run the following two commands:

- `cd /opt/rcc-1.3-rc6-gcc/src/samples`

- `make`

Please note that the samples found in this directory are very helpful to understand the structure of applications written for RCC. They are great starting points for development.

As a final (although optional) step, the RCC user manual recommends installing the RTEMS kernel source code. It can be downloaded from the website of Cobham Gaisler, at *Downloads -> Compilers -> RTEMS source code*. The file name is `rtems-5-1.3-rc6-src.txz`. After downloading, extract the contents to `C:\opt\rcc-1.3-rc6-gcc\src` so that the resulting path becomes `C:\opt\rcc-1.3-rc6-gcc\src\rcc-1.3-rc6`.

For instructions on how to install in an Linux environment, please see the RCC user manual [3].

3.3 Installing GRMON

In order to load the compiled executable onto the development board it is possible to use the debug monitor software GRMON. Within the scope of this project the version GRMON3 Professional was used. GRMON relies on a few different components being available on the used system to fo function. This section describes these components, their installation and finially the installation of GRMON itself. Please note that in order to download some components from Cobham Gaisler's website you will need a password and a username. These login details are delivered together with the development board.

Java 8 Both GRMON (at least the GUI) and Eclipse need Java to work. Install Java version 8 or later on the system [4]. Java is freely available online at the address `https://www.java.com/en/download/`. Simply download the installer and follow the setup.

Sentinel LDK runtime GRMON is licensed using a Sentinel LDK USB hardware key. This means that in order to run GRMON a USB dongle with the correct license must be connected to the development PC. Within the scope of this project a node-locked (identified by its purple color) key was used. As stated in the GRMON user manual, it is necessary to install the Sentinel LDK runtime in order for the PC to validate the USB dongle. The latest LDK runtime is available for download as a zip file on the website of Cobham Gaisler, at *Downloads-> Debug Tools*. Extract the archive to a desired destination and find detailed installation instructions in the `README` file. However, ignore all instructions about installing `haspvlib_<vendorID>.so` and/or `haspvlib_x86_64_<vendorID>.so` [4].

D2XX When using Windows, GRMON will need the so-called D2XX interface developed by Future Technology Devices International Limited (FTDI) in order to communicate with the board via the USB interface. The D2XX driver is available for download at `https://www.ftdichip.com/Drivers/D2XX.htm`. Locate the Windows version and click the link *setup executable*. Extract the downloaded archive and run the file `CDM21228_Setup.exe` to complete the installation [7]. Make sure to restart the computer after this step.

GRMON3 Professional GRMON3 Professional is available for download as a zip file at Cobham Gaisler's website on the same page the the Sentinel LDK runtime. Version 3.0.16 or later is required in order to connect to the development board. After downloading the archive can be extracted to a desired destination, and it is recommended to add the path `<your_path>\grmon-pro\windows\bin64` to your system path variable [7].

3.4 Setting Up Eclipse For Development And Debugging

The Eclipse IDE is a popular development platform and was used as the primary IDE during this project. This section describes the Eclipse installation process and how to set up an Eclipse project for debugging a LEON target, such as with development for the GR-CPCI-GR740 development board.

The provided guide [5] states that it is written for Eclipse IDE for C/C++ Developers 2019-09 but is likely to work with other versions as well as long as all the necessary plugins are available. It is recommended to install the latest version of Eclipse IDE for C/C++ Developers. Before continuing make sure that the software components mentioned in the previous sections of this chapter are installed, especially Java (as mentioned in Section 3.3), since Eclipse needs Java to work.

3.4.1 Installation and project setup

The latest version of Eclipse is freely available for download at `https://www.eclipse.org/`. Run the installer and follow the instructions therein whilst making sure to install Eclipse for C/C++ Developers. After installation, create a new project by clicking *File -> New -> C/C++ Project*. Select the *Managed Build* template and click *Next*. Now enter the name of your project and under *Project type* select *Hello World ANSI C Project*. Under *Toolchains*, select *Cross GCC* and click *Next*. Now customize the input values as desired before clicking *Next*. Click *Advanced settings...*.

In the settings tree, select *C/C++ Build -> Tool Chain Editor*. Set *Configuration* to *Debug* and at *Current Builder* choose *CDT Internal Builder*. Now change *Configuration* to *Release* and choose the same internal builder.

Now go to *C/C++ Build -> Settings* and set *Configuration* to *All configurations*. Select *Cross GCC Compiler -> Miscellaneous* and add the compiler flags `-qbsp=gr740 -mcpu=leon3` to the field *Other flags*. These flags control compiler behaviour and the choices are described in detail in Chapter 2 of the RCC user manual[3]. Now select *Cross GCC Linker* and add the same flags in the field *Linker flags*. Click *Apply and close* and then *Next*.

In the field *Cross compiler prefix* enter the prefix of the toolchain, i.e. `sparc-gaisler-rtems5-`, including the trailing hyphen. In *Cross compiler path* enter the path to the `bin` folder of the toolchain, i.e. `C:\opt\rcc-1.3-rc6-gcc\bin`. Now press *Finish*. It should now be possible to build the project by clicking *Project -> Build Project* [5].

Finally, go to *Project -> Properties -> C/C++ General -> Paths and Symbols* and open the *Includes* tab. Click on *GNU C* and add the path `C:\opt\rcc-1.3-rc6-gcc\sparc-gaisler-rtems5\gr740\lib\include` to the *Include directories* list. This will allow eclipse to find the board-specific header files provided by Cobham Gaisler.

3.4.2 Debugging setup

In order to correctly load the compiled executable and debug the code as it is running on the target a hardware debug configuration must be made in Eclipse. Click on *Run -> Debug Configurations...*. Double click on *GDB Hardware Debugging* to create a new configuration. Now verify that the project name is correct and that the path in *C/C++*

Application is set to the application to be debugged. Go on the *Debugger* tab and add the path to the GDB executable at *GDB command*, i.e. `C:\opt\rcc-1.3-rc6-gcc\bin\sparc-gaisler-rtems5-gdb.exe`. Now set *JTAG Device* to *Generic TCP/IP* and enter `2222` in `Port number`. This number must match the port number opened by GRMON when starting the GDB server (see the following section).

Go to the *Startup* tab and add `monitor gdb reset` to the text-box at the top. Verify that *Load Image* and *Load Symbols* are both checked. Check the *Set breakpoint at* box and add `main` in the field. Now add `monitor gdb postload` to the second text-box. Please note that this command can lead to problems and the IDE setup guide from Cobham Gaisler states that this line can be excluded without any issues in the vast majority of cases. During this bachelor thesis project, the addition of this line led to problems during debugging and was therefore not used.

Finally, make sure *Resume* is checked [5].

3.4.3 Debugging with Eclipse

As a first step, make sure to have the development board in an unpowered state. Connect the development PC to the board via the provided USB cable, the port is visible in the lower left corner of Figure 3.5. Be aware that the connection when using the provided cable can be non-ideal since the casing around the cable connector prevents a tight fit into the port. One might consider using an alternative cable. Provided that all the hardware settings as described in Section 3.1 have been configured, it is now safe to power the board.

Before it is possible to start debugging, GRMON must be started. While it is possible to do this in the command line outside of Eclipse it is more convenient to fully control the debugging process from within one application, namely Eclipse. In order to achieve this a new terminal can be created by clicking the terminal icon in the Eclipse tool bar. The icon is highlighted in Figure 3.6.

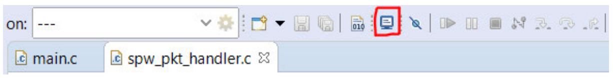
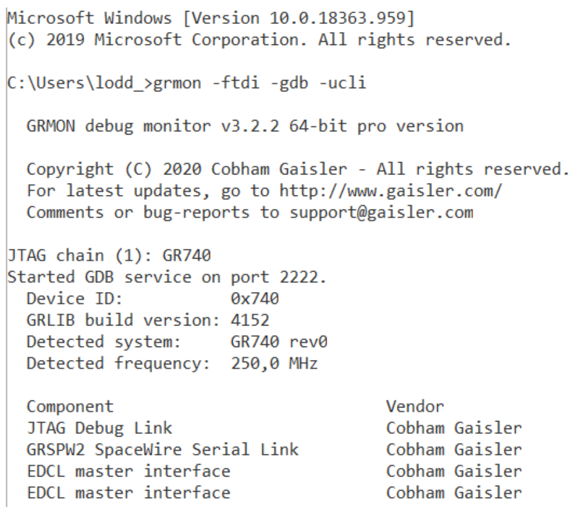


Figure 3.6: The terminal icon in the Eclipse tool bar.

Choose a local terminal with UTF-8 encoding in the pop-up dialog. Click *OK* and the terminal window should open at the bottom of Eclipse. To start GRMON and enable it to act as a GDB server (which is necessary for debugging the target), run the following command in the opened terminal: `grmon -ftdi -ucli -gdb`. Remember that the USB hardware key must be connected to the PC in order to run GRMON.

The `-ftdi` flag specifies the debug interface which is used. Possible options can be found in the GRMON user manual [4] Chapter 5. The `-ucli` flag makes sure that Eclipse prints the output sent from the board to PC to the GRMON terminal. This printing functionality is helpful during debugging and the usage of this flag is mentioned in the FAQ of the IDE setup guide [5] as a solution to non-working calls to `printf` in the embedded code. The `-gdb` flag starts the GDB server and can be specified with a port number as such `-gdb [port]`. This port number must match the number set in the debug configuration as described in Section 3.4.2.

If the setup is correct, a JTAG scan is performed and detected devices are printed to the terminal as seen in Figure 3.7. In case of errors, make sure that the USB cable has a proper connection to the board. The FAQ of the IDE setup guide [5] is a good resource for solving debug related problems.



```
Microsoft Windows [Version 10.0.18363.959]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\lodd_>grmon -ftdi -gdb -ucli

  GRMON debug monitor v3.2.2 64-bit pro version

  Copyright (C) 2020 Cobham Gaisler - All rights reserved.
  For latest updates, go to http://www.gaisler.com/
  Comments or bug-reports to support@gaisler.com

JTAG chain (1): GR740
Started GDB service on port 2222.
  Device ID:          0x740
  GRLIB build version: 4152
  Detected system:     GR740 rev0
  Detected frequency:  250,0 MHz

  Component                           Vendor
  JTAG Debug Link                     Cobham Gaisler
  GRSPW2 SpaceWire Serial Link        Cobham Gaisler
  EDCL master interface               Cobham Gaisler
  EDCL master interface               Cobham Gaisler
```

Figure 3.7: Running the command to start GRMON from within Eclipse.

At this point, assuming GRMON has started correctly and the project can be built without errors, debugging can begin. Click the arrow on the green *Debug As...* icon on the Eclipse tool bar and choose the debugging configuration which was created as described in Section 3.4.2. This should start the debugging process. If the board is to be restarted GRMON must be restarted as well. Write *q* in the terminal to exit GRMON.

3.5 The SpaceWire MK3 Brick

The previous sections of this chapter have explained the setup necessary for the top-most connection between the PC and development board in Figure 3.1, namely the debug interface. In this section, the bottom-most connection of the figure is described. The purpose of this connection is to communicate with the board via SpaceWire from the development PC. By loading an application onto the board and then using the SpaceWire MK3 brick to send a SpaceWire packet and receive any potential reply to the PC, the behavior of the embedded application can be evaluated. Therefore, it is an important part of debugging when developing an application which involves SpaceWire protocol handling. Whereas this section describes how to set up and use the SpaceWire MK3 brick, it is advisable to first read Section 2.6 for a more detailed understanding of the inner workings of the brick.

3.5.1 Physical Setup

The MK3 brick has two different SpaceWire interfaces available through port 1 and 2. It does not matter which one is used as long as the chosen port is noted and used later in the relevant software settings. The brick is connected to the PC via the USB 3.0 port available on its rear view panel, however this should be done first after the proper software and drivers have been installed so that the PC will correctly identify the brick. To connect to the development board, make sure it is in an unpowered state before attaching a SpaceWire cable to one of the eight SpaceWire ports (not counting port 0 which is the SpaceWire debug interface of the board). Then connect the other end of the cable to the brick via port 1 or 2 found on the front panel.

3.5.2 Software Installation

The necessary software and drivers are saved on a CD which should be found together with the SpaceWire MK3 brick. The components can be installed directly from the CD and after installation there should be a shortcut called *STAR-System - Shortcut* placed on the desktop. Now the brick can be connected to the PC. The desktop shortcut leads to a folder of shortcuts which in turn leads to documentation and some useful executables. The documentation (such as [13]) is given in the form of a collection of HTML documents.

The executables are small GUI programs intended for convenient control of the brick. Three very useful executables are:

- **Device Configuration:** Allows for easy configuration of the brick through the presented GUI. Here it is possible to set the internal SpaceWire router to either router or interface mode, as well as configure the router table and connections between the channels and SpaceWire ports. See Section 2.6 for more details.
- **Receive:** Listens for and displays incoming packets as raw bytes.
- **Transmit:** Allows user to define the raw bytes to be sent and then transmit them. Also has functionality for injecting EEP errors in packets.

There are other executables available in the shortcut folder, however only these three were used for development within the scope of this thesis. When many different packets are to be sent it quickly becomes cumbersome to use these simple GUI applications. Therefore, one can instead turn to the provided C++ API.

3.5.3 C++ API

The STAR-System C++ API allows the user to control all brick operations through custom applications. This is very useful when developing a test-bench which needs to send and/or receive packets via SpaceWire. The fastest way to get started with the brick's C++ API is by going through some of the provided examples, which can be found after installation at:
`C:\Program Files\STAR-Dundee\STAR-System\apis\cpp_api\examples`.
It is described in detail in [13].

4 Requirements

The requirements for both the embedded application and the test-bench developed during this project stem from a set of initial requirements defined at the project beginning that sequentially has been modified and added to. They are mainly a result of discussions between the student and the primary thesis supervisor, although the RMAP specification is also an important source as it defines large parts of the code behavior. The activity diagrams in Section 5.2.1 are tightly coupled to the requirements regarding the RMAP handling and are useful as a graphical overview of what here is described in text. The requirements are listed in the following subsections.

4.1 Embedded Application

1. The application is to be run on a LEON4 processor on a GR740 SoC.
2. The application must configure the hardware necessary for SpaceWire communication.
    a) The application must configure the on-board SpaceWire router to allow communication between the GR740 and a SpaceWire source node external to the development board.
    b) The application must set up a SpaceWire interface for communication.
3. The application must be able to receive incoming SpaceWire packets.
4. The application must support continuous operation without manual intervention (e.g. reset).
5. The application must make sure the RMAP handling is done in software and not in hardware.

6. The application must define a valid memory area from which RMAP commands can read or write to.
7. The application must define a destination key.
8. The application must define a logical address.
9. The application must react and reply to incoming RMAP communication according to the behavior of a destination node as specified in the RMAP standard.
    a) Incoming packets terminating with an EEP shall be discarded.
    b) Incoming truncated packets shall be discarded.
    c) Incoming packets with a protocol ID of a value not equal to 0x01 shall be discarded.
    d) Incoming RMAP packets with an incorrect header CRC shall be reported to a CRC error counter and then discarded.
    e) Incoming RMAP reply type packets shall be ignored.
    f) Incoming RMAP commands with an unexpected destination key and a set acknowledge bit shall cause a RMAP reply with a status code reporting "Invalid destination key". If the acknowledge bit is not set the command is discarded and ignored.
    g) The application must handle incoming RMAP Read commands.
        i. A read command trying to read outside the valid memory area and having a set acknowledge bit shall cause a RMAP reply with a status code reporting "RMAP command not implemented or authorized". If the acknowledge bit is not set the command is discarded and ignored.
        ii. A completely valid read command shall cause a reply containing the requested data and a status code reporting "Command executed successfully".
    h) The application must handle incoming RMAP Write commands.

i. A write command trying to write outside the valid memory area and having a set acknowledge bit shall cause a RMAP reply with a status code reporting "RMAP command not implemented or authorized". If the acknowledge bit is not set the command is discarded and ignored.

ii. If there are fewer data bytes than declared in the command header and the acknowledge bit is set, a reply with a status code indicating "Early EOP" shall be sent. If the acknowledge bit is not set, no reply is sent. It does not matter if some or all of the data has been written to memory before this is discovered, but afterwards the command execution stops.

iii. If there are more data bytes than declared in the command header and the acknowledge bit is set, a reply with a status code indicating "Cargo too large" shall be sent. If the acknowledge bit is not set, no reply is sent. It does not matter if some or all of the data has been written to memory before this is discovered, but afterwards the command execution stops.

iv. A write command with a set Validate Before Write bit shall cause a check if there is enough buffer space available for data CRC verification.

    A. If there is not enough space and the acknowledge bit is set a reply with a status code indicating "Verify Buffer Overrun" shall be sent. If the acknowledge bit is not set the command is discarded and ignored.

    B. If there is enough buffer space the data CRC shall be calculated and compared to the CRC value in the command. If not valid a reply with a status code of "Invalid Data CRC" shall be created and sent if the acknowledge bit is set. If the acknowledge bit is not set the command is discarded and ignored. If the data CRC is valid the command data can be written to memory.

v. A write command with a cleared Validate Before Write bit shall not perform the Verify Buffer Overrun and and Data CRC checks before data is written to memory.

vi. After a write command that successfully writes all its data to memory the data CRC of the written data shall be compared to the data CRC in the command. If valid and the acknowledge bit is set a reply with a status code indicating "Command executed successfully" shall be sent. If invalid and the acknowledge bit is set, a reply with a status code indicating "Invalid data CRC" shall be sent. If the acknowledge bit is cleared no reply is sent.

i) The application must handle incoming RMAP Read-Modify-Write commands.

   i. The declared and found data lengths must be identical and be one of the following values 0x00, 0x02, 0x04, 0x06, 0x08. If the value is not one of these values a reply with status code indicating "RMW data length error" shall be sent. If the data length was found invalid the command is then discarded and ignored.

   ii. If the command tries to perform the RMW action outside of valid memory area a reply with a status code indicating "RMAP command not implemented or authorized" shall be sent. The command is then discarded and ignored.

   iii. The data and mask CRC shall be calculated and compared to the CRC in the command. If invalid, a reply with a status code indicating "Invalid Data CRC" shall be sent. The command is then ignored and discarded.

   iv. A valid RMW command shall cause the data in the command to be written to memory according to an arbitrary scheme using the mask in the command. The unmodified data at the memory location before the write operation shall be sent in a reply.

j) All RMAP replies must contain a transaction ID which matches the command it responds to.

k) All replies must identify themselves as RMAP replies by using a protocol ID value of 0x01.

l) All reply types designed to contain data must have a data CRC value appended no matter if there is any data or not.

4.2 Test-bench

1. The test-bench shall be able to configure the SpaceWire MK3 brick.

2. The test-bench shall be able to use the SpaceWire MK3 brick to send the three different types of RMAP commands and wait for and receive the corresponding replies.
3. The test-bench shall be able to read instructions according to a defined format from user-defined script files.
4. The test-bench shall be able to send commands as defined in script file instructions.
5. The script files can contain multiple commands.
6. The test-bench shall be able to automatically perform instructions from several script files.
7. The test-bench shall create one output log file per input script file. The output log files shall contain information on the replies corresponding to the commands in the script file to which the output log file corresponds.
8. The test-bench shall be able to detect if no reply was received within a certain time-out.
9. The test-bench shall be able to detect and report incorrect instructions in script files.
10. The test-bench shall allow script files to contain a header which will be copied to the corresponding output log file.
11. The test-bench shall include some automatic general checks (e.g. making sure that incoming replies have the same transaction ID as the corresponding command) to assist the developer to detect such errors.
12. The output log files shall contain information on the raw bytes of the incoming reply.
13. When performing a set of script files, an error in one script file shall not interrupt the execution of following script files.
14. Errors due to script files or test-bench during testing shall be reported to the user.

5 Design

5.1 Overall system design

Figure 5.1 illustrates the original overall system design from the early stages of the project development. It is a control hierarchy diagram showing how the top-most modules have control over lower or same-level modules. The diagram includes hardware components and software modules in order to give a more complete overview of the whole system. To the right in the diagram are the modules which make up the test-bench and in the lower left corner the embedded application can be seen. The modules for each of these are discussed in the sections of this chapter. It is important to note that this is not the final system design since a few modifications were done during the project, and some parts were left out due to time constraints. The differences in the final design will be mentioned in the upcoming sections, but Figure 5.1 is still useful for understanding the system architecture.

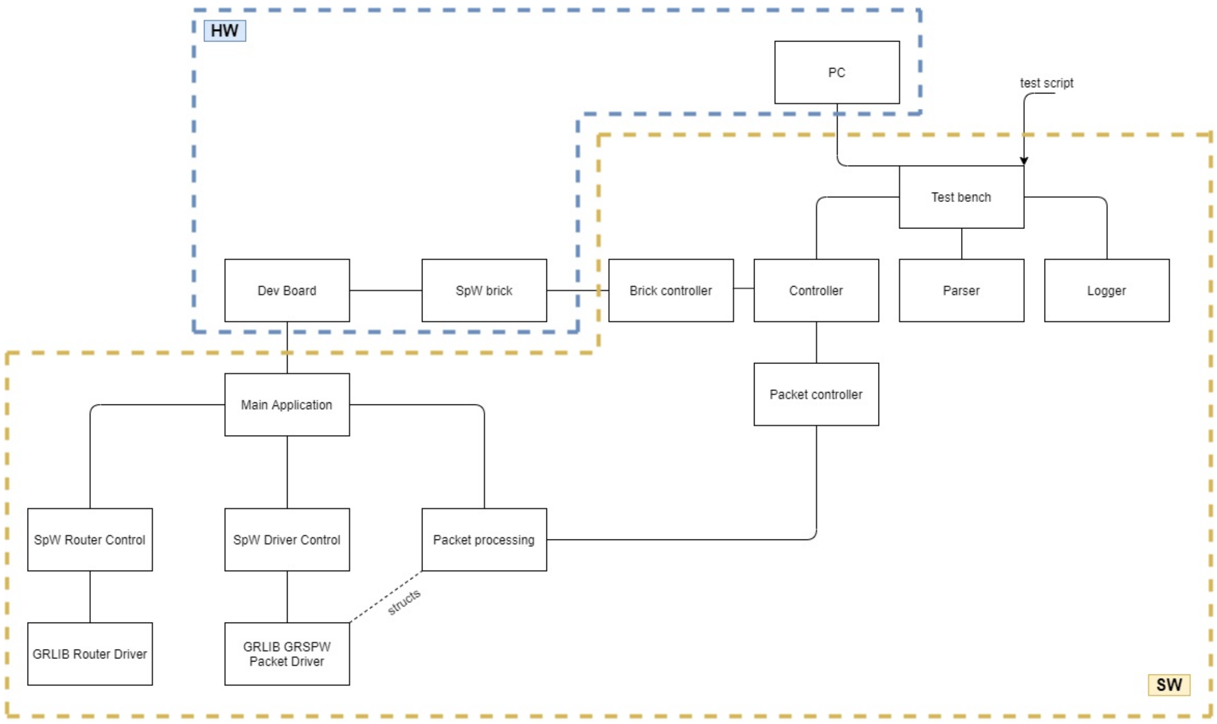


Figure 5.1: Diagram of the original system design including test-bench.

The overall design is mainly derived from dividing different responsibilities into logical blocks. For example, the module containing functions for configuring the SpaceWire router is separated from the module which contains packet handling functions. One part of the design is inspired by a pattern mentioned in the book *Making embedded systems* by Elecia White [14]. This is the well-known Model-View-Controller pattern but now in an embedded environment. The three parts can then be described as follows:

- **Model:** The model is the module which represents the actual algorithm or functionality that a developer wants to implement. It receives some data input and provides some data output.
- **View:** The view represents the interface to the user, for example this can be panels with push buttons or displays. More generally, the view is responsible for input and output of data at the borders of a system.
- **Controller:** The controller is a translator between the model and view. It receives input from the view and formats it in a way the model can understand. Similarly,

it receives the output from the model and formats it in a way compatible with the view.

What makes this pattern so useful is the interchangeability of the controller, which can help separating the functionality of an application from the hardware. This makes it possible to test the (overall) behavior of the model without needing the target hardware [14]. In this design the MVC pattern is not used as the overall pattern, but should rather be seen as a sub-pattern which is used for the planned connection between the packet controller and the packet processing modules. The test scripts can be seen as the view in this case, and the packet processing as the model. The packet processing contains functions for interpreting and reacting to incoming SpaceWire packets as well as formulating replies, all according to the RMAP specification. This is the main functionality of the embedded application. In the normal case, the SpaceWire packets handled by the packet processing module will have been received by the GR740 from the SpaceWire MK3 brick. This is the case when the brick controller has been used to send packets using the brick as instructed by the test scripts. By adding the packet controller, which can translate the instructions in the test scripts into the kind of data expected by the packet processing module, the hardware is in effect bypassed and the packet processing functions can be tested separately. This makes it convenient to test the application logic without access to the target hardware. However, this is not an ideal type of testing since concepts like timing will be different on different machines. In this case, where the RMAP handling in the model also requires reading and writing to certain memory areas it becomes a bit more difficult to separate the model from the hardware. Therefore, testing the model separate from the hardware could in this case be done by simply printing the actions which would have been taken as response to a certain test-script. This would at least allow for the RMAP handling logic to be tested separately.

The MVC pattern described here is mentioned in order to explain the original design. Due to time constraints the packet controller and therefore direct connection between the test-bench and embedded application could not be developed within the scope of this project, and is left as a suggestion for future improvements. Currently, the test-bench only controls the brick controller which in turn controls the SpaceWire MK3 brick. The packet controller and abstract controller module do not exists at the moment of writing.

5.2 Embedded Application

The design of the embedded application at the moment of writing is illustrated in Figure 5.2. It generally has the same structure as in the original design but with two additions, the two modules containing macro definitions. The design is based on the example found in the file `test.c` located at `C:\opt\rcc-1.3-rc6-gcc\src\samples\spw\grspw-pkt` after installing RCC.

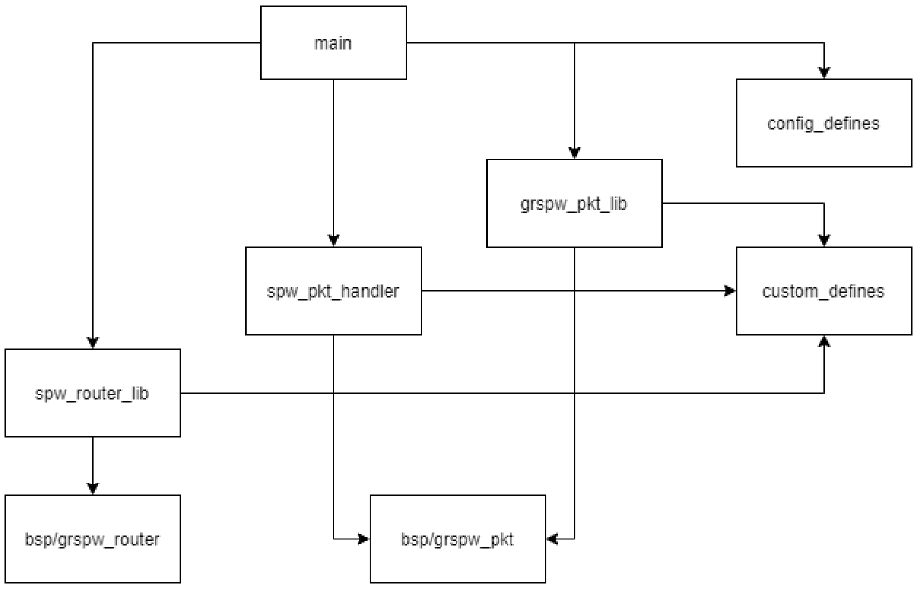


Figure 5.2: Architecture of the embedded application

- **config_defines:** RTEMS is mainly configured by defining values for certain macros. The definition of these macros is done in a separate module in order not to clutter the main module. The values are mainly derived from the example upon which the design is based.
- **custom_defines:** This module contains macro definitions for values which might be needed throughout the application, such as the logical address of the GR740 defined by the user.
- **grspw_pkt_lib:** This module is mainly used to configure and control a SpaceWire interface on the GR740. It was provided in RCC in the parent directory of the example upon which this design is based.

- **spw_pkt_handler:** This module contains the central part of the application, namely the logic implementing the RMAP handling. It is responsible for performing the received RMAP commands and creating RMAP replies.
- **spw_router_lib:** Is used for configuration of the SpaceWire router on the GR740. It is heavily based on the file `spwrouter_custom_config.c` found together with the example upon which the design is based.
- **bsp/grspw_pkt:** The SpaceWire packet driver (see Section 2.4.1).
- **bsp/grspw_router:** The SpaceWire router driver (see Section 2.4.2).

5.2.1 RMAP Handling

The activity diagrams presented in this section describe how the application handles RMAP commands as a result of the RMAP specification. It is almost completely compliant with the RMAP specification with a few differences stemming from the DMA implementation of the packet driver. This is mentioned in the subsection on the RMAP write command. Another difference is that the increment bit (see Section 2.2.1) of any RMAP command will not have any effect in the current implementation.

The first part after reception of a RMAP packet is to do some general checks which are relevant for any RMAP command. If all checks pass, the type of RMAP command is found and the packet is then delegated to the corresponding command handler.

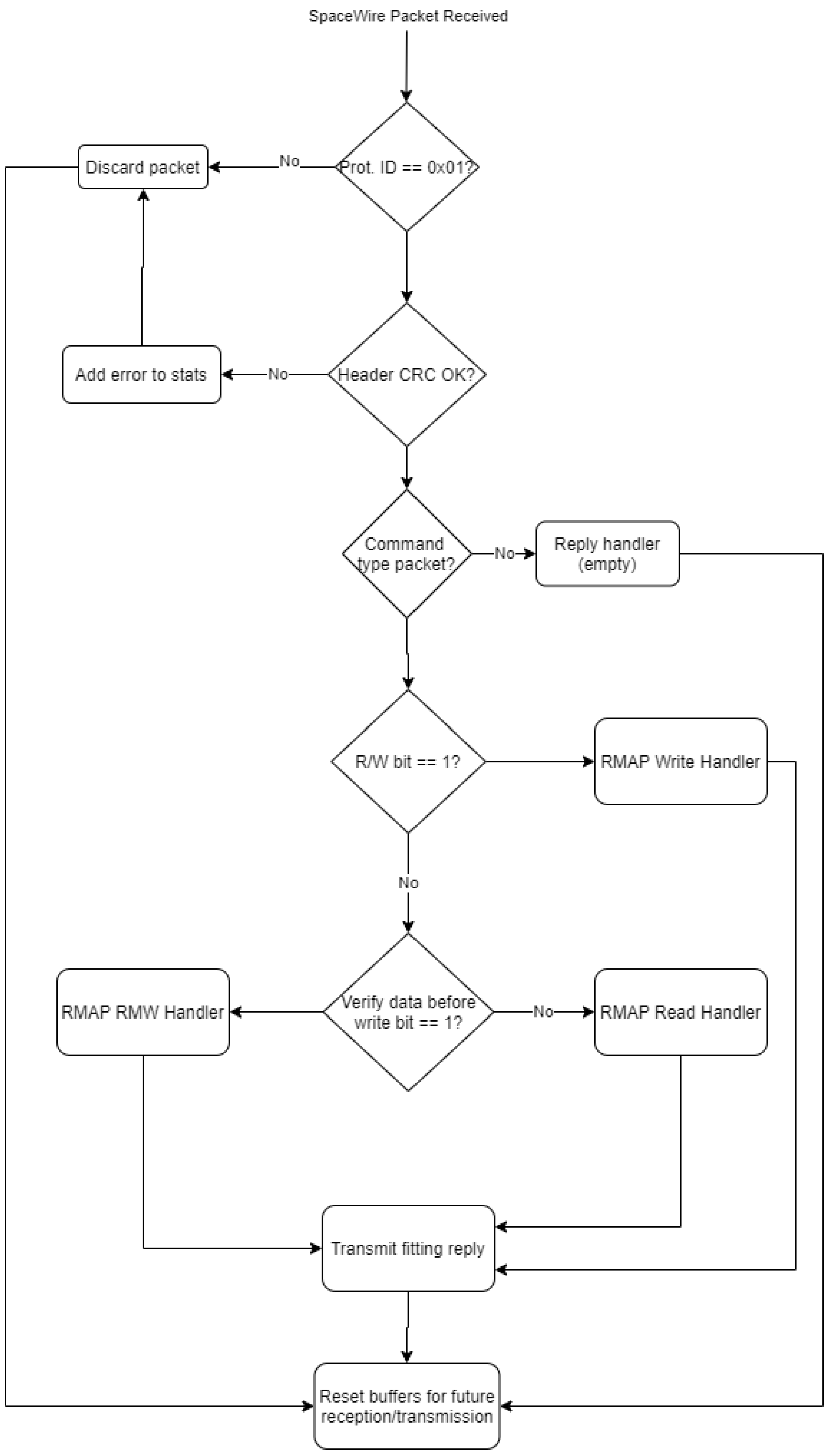


Figure 5.3: Activity diagram of first actions upon received packet.

Read Command Handling

Read command handling is the most straightforward, after a few checks pass the requested data is appended to a RMAP read reply. If the checks don't pass, the correct error code is set and given to the reply.

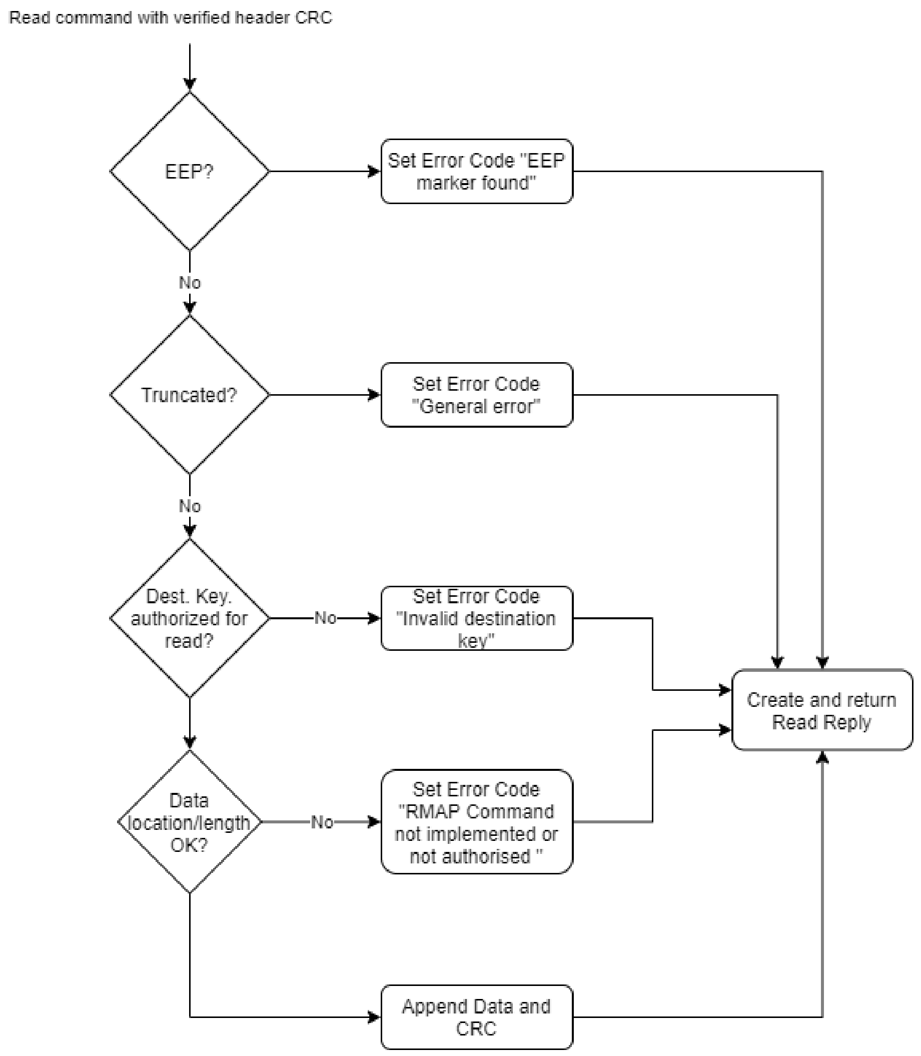

Read command with verified header CRC
EEP?
Set Error Code "EEP marker found"
No
Truncated?
Set Error Code "General error"
No
Dest. Key. authorized for read?
No
Set Error Code "Invalid destination key"
Create and return Read Reply
Data location/length OK?
No
Set Error Code "RMAP Command not implemented or not authorised "
Append Data and CRC

Figure 5.4: Activity diagram of the handling of a Read command.

Write Command Handling

The write command handler is the most complicated of the three command handlers. There are many checks involved and the implementation is not completely compatible with the RMAP specification. This is due to the use of DMA in the packet driver. The issue is that the RMAP specification implies that the incoming SpaceWire characters are interpreted one-by-one as they are received. Therefore, RMAP handling and checking are described as if the entire RMAP packet is not available at the start of handling it. In the DMA implementation, interrupts for incoming SpaceWire packets can not be generated on a character basis but only when a packet terminating EEP or EOP is found. Therefore, when the waiting embedded application is woken up by incoming communication, at least one entire packet will have been received. This makes e.g. the Verify Buffer Overrun check obsolete. According to the RMAP standard, after the header of a RMAP packet is received the data length field is checked to make sure the incoming data can fit in the available buffers before it is accepted. Since in this implementation the entire packet is already received at the point of this check, there is little point in performing it. Instead, in the beginning of the command handling a check is performed to see if the packet was truncated, meaning that the entire packet is too big to fit in the reception buffers. This is used as an alternative to the Verify Buffer Overrun check even though it in effect does not check the buffer overrun based on the length of the data, but on the length of the entire packet. This is a somewhat subtle difference.

Another difference from the RMAP specification due to the use of DMA is the order of checking the data length. According to the RMAP specification the application should look for an early EOP as the bytes are being written to memory. If the EOP is found before the declared number of bytes have been written, this is an error of insufficient data. Conversely, if the declared number of bytes have been written but no EOP is found, this is a data overflow error. At the time at which either of these errors are found, it is possible that data has already been written to the memory location. The RMAP specification allows this, even though the command is to be interrupted as soon as the error is found. In the case of this implementation, there is no need to wait until after writing starts to do these checks, since the entire packet with all the data is already received. Therefore the checks for insufficient data and data overflow happen before writing starts.

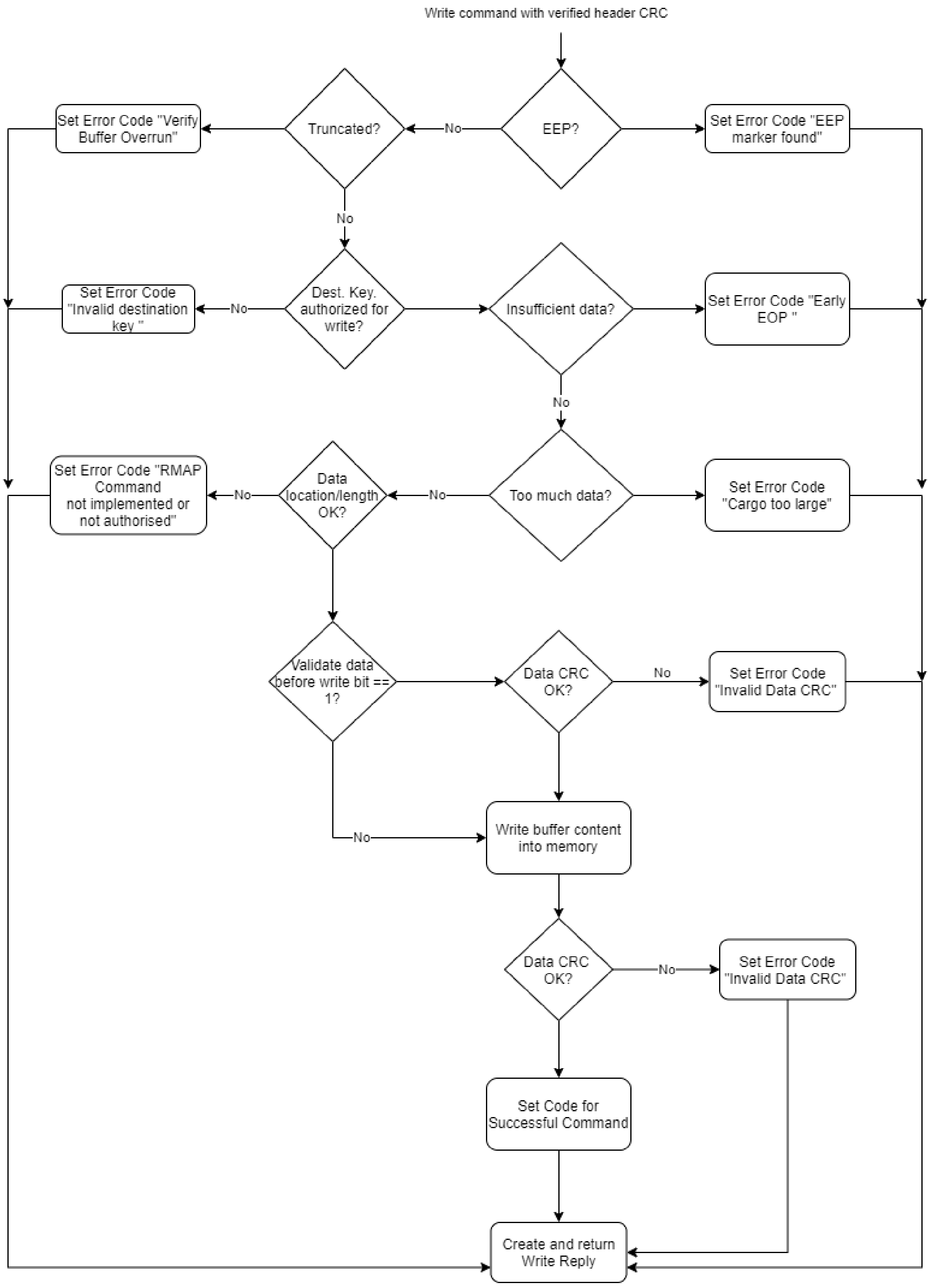
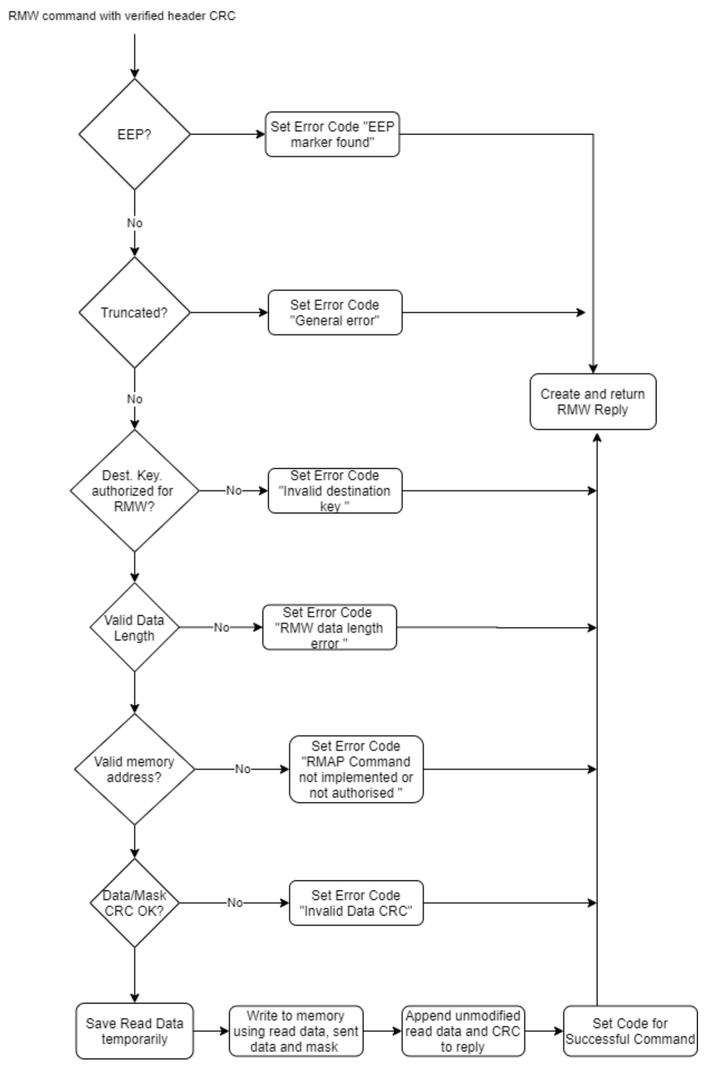

Write command with verified header CRC
EEP?
Set Error Code "EEP marker found"
No
Truncated?
Set Error Code "Verify Buffer Overrun"
No
Dest. Key. authorized for write?
No
Set Error Code "Invalid destination key "
Insufficient data?
Set Error Code "Early EOP "
No
Too much data?
Set Error Code "Cargo too large"
No
Data location/length OK?
No
Set Error Code "RMAP Command not implemented or not authorised"
Validate data before write bit == 1?
Data CRC OK?
No
Set Error Code "Invalid Data CRC"
No
Write buffer content into memory
Data CRC OK?
No
Set Error Code "Invalid Data CRC"
Set Code for Successful Command
Create and return Write Reply

Figure 5.5: Activity diagram of the handling of a Write command.

Also notice how a reply is always created no matter if the acknowledge bit is set or not. Whether the reply is sent or not is decided outside the write command handler.

Read-Modify-Write Command Handling

The RMW handler shares many elements with the other two handlers. One difference is the check for valid data lengths since the RMW command need the data length field to be one of a set of values. When passing all checks the RMW command is performed as described in Section 2.2.1.



RMW command with verified header CRC
EEP?
Set Error Code "EEP marker found"
No
Truncated?
Set Error Code "General error"
Create and return RMW Reply
No
Dest. Key. authorized for RMW?
No
Set Error Code "Invalid destination key "
Valid Data Length
No
Set Error Code "RMW data length error "
Valid memory address?
No
Set Error Code "RMAP Command not implemented or not authorised "
Data/Mask CRC OK?
No
Set Error Code "Invalid Data CRC"
Save Read Data temporarily
Write to memory using read data, sent data and mask
Append unmodified read data and CRC to reply
Set Code for Successful Command

Figure 5.6: Activity diagram of the handling of a RMW command.

Currently, the biggest flaw in the RMW handling is that there is no check that makes sure that the declared data length and the received data length are both the same, it is assumed the declared data length is correct.

5.3 Test-bench

The final design of the test-bench is very similar to the original design. The only difference is that the logger is only directly controlled by the rmap module instead of the test_bench module.

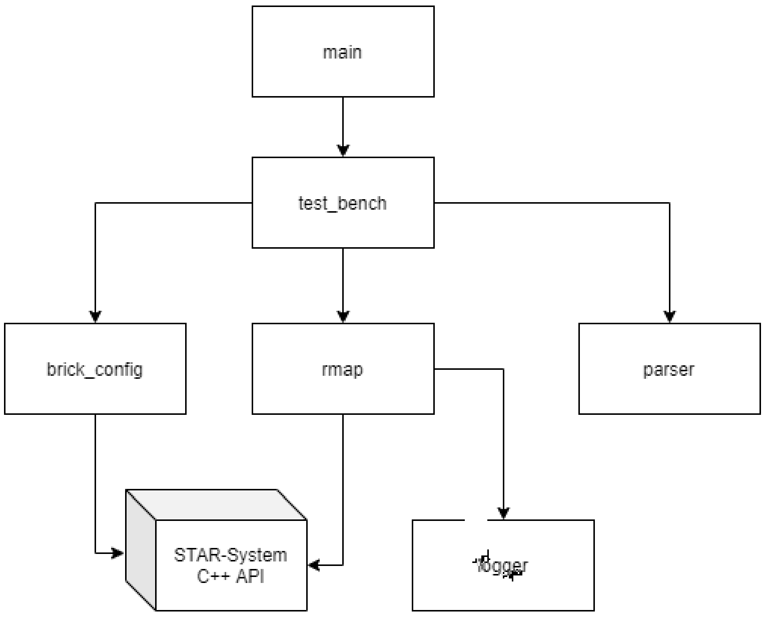


Figure 5.7: Architecture of the test-bench.

- **test_bench:** Responsible for performing the automatic tests and keeping track of errors occurring during the process.
- **brick_config:** Contains functions which can set up the SpaceWire MK3 brick for operation.
- **rmap:** This corresponds to the brick controller in the original design. It contains functions which uses the brick to send and receive RMAP packets.
- **parser:** This module is responsible for interpreting the instructions in the test scripts given by the user into data structures that the rmap module can use.

- **logger:** This module is responsible for creating the output log files as a response to each test script.
- **STAR-System C++ API:** The API provided by STAR-Dundee to control and configure the SpaceWire MK3 brick.

6 Implementation

The embedded application and the test-bench both consist of too much code to conveniently include within this document. Instead, the two projects will be found on a CD distributed together with this thesis. In this chapter, some important highlights of the code are shown and explained.

The starting points for both projects were based on examples provided in the RCC and STAR-System distributions. The embedded application is heavily based on the file `test.c` found at `C:\opt\rcc-1.3-rc6-gcc\src\samples\spw\grspw-pkt` after installing RCC. It contains the main application, but also uses another file in the same directory responsible for router configuration. This file is the basis for the created router handling module `spw_router_lib.c`. One directory above `test.c` one can find the file `grspw_pkt_lib.c` which was used in this project mainly for SpaceWire device configuration. Further files were created, and some modified, when dividing the actions taken in `test.c` into separate modules. For example, the section containing macro definitions for configuring RTEMS were put in an own module in order to make the main file more readable. As the different logical responsibilities were divided up into separate modules, corresponding header files were created as well. The files were all imported into an Eclipse project and the IDE environment set up as described in Chapter 3. The starting point for the test-bench was an example Visual Studio project called `rmap_examples.vcxproj` found at `C:\Program Files\STAR-Dundee\STAR-System\apis\cpp_api\examples\rmap` after installing STAR-System. By importing this file through Visual Studio a project is opened which contains examples of how to create RMAP commands using the STAR-System C++ API. This was used as inspiration for the module responsible for using the SpaceWire MK3 brick for transmitting and receiving RMAP commands. Another project in the same example directory contains concrete examples on how to use the API to configure the MK3 brick. This project can be opened by importing the project file `device_config_-tester.vcxproj` in Visual Studio. Within the scope of this project, Visual Studio Community 2017 version 15.9.16 was used for test-bench development.

If the developed project will be used as a starting point for further development, please be aware that the project will most likely need some changes to run on a new development PC. Specifically, all paths (such as the include paths) will need to be changed to match the new environment. Furthermore, there might be changes in directory structure if using another release candidate than RC6, which is used in this project, so header files might be found in other places than those expected by the `#include` statements in the code.

6.1 Embedded Application

As previously mentioned, RTEMS is mainly configured through defining a set of macros. The bulk of the configuration used in this project is seen in Listing 6.1.

Listing 6.1: RTEMS configuration through macro definitions

```
/* configuration information */
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER

/* The shared IRQ layer need one semaphore.
 */
#define CONFIGURE_MAXIMUM_TASKS                   8
#define CONFIGURE_MAXIMUM_SEMAPHORES              20
#define CONFIGURE_MAXIMUM_MESSAGE_QUEUES          20
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS 32
#define CONFIGURE_MAXIMUM_DRIVERS 32
#define CONFIGURE_MAXIMUM_PERIODS                   1

#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_INIT_TASK_ATTRIBUTES      RTEMS_DEFAULT_ATTRIBUTES | RTEMS_FLOATING_POINT
#define CONFIGURE_EXTRA_TASK_STACKS           (40 * RTEMS_MINIMUM_STACK_SIZE)
#define CONFIGURE_MICROSECONDS_PER_TICK       RTEMS_MILLISECONDS_TO_MICROSECONDS(2)

#include <rtems/confdefs.h>

/* Configure Driver manager */
#if defined(RTEMS_DRVMGR_STARTUP) && defined(LEON3) /* if --drvmgr was given to configure */
 /* Add Timer and UART Driver */
 #ifdef CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
  #define CONFIGURE_DRIVER_AMBAPP_GAISLER_GPTIMER
 #endif
 #ifdef CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
  #define CONFIGURE_DRIVER_AMBAPP_GAISLER_APBUART
 #endif
#endif

#define CONFIGURE_DRIVER_AMBAPP_GAISLER_SPW_ROUTER /* SpaceWire Router  */
#define CONFIGURE_DRIVER_AMBAPP_GAISLER_GRSPW2     /* SpaceWire Packet driver */

#include <drvmgr/drvmgr_confdefs.h>
```

The definition of the first two macros makes sure there is a clock tick driver and a console driver, the latter in order to enable printing which is useful for debugging. The following six macros define some maximum numbers of OS objects. This will cause RTEMS to allocate enough space for as many objects in the application image. The values currently used are directly derived from the example file and do not reflect the needs of the developed RMAP application. Therefore these values should be decreased in order to decrease the size of the final application.

The following three macros set some attributes for RTEMS tasks. The default attributes with the addition of floating point arithmetic are set, but the latter could most likely be removed for the RMAP application since no floating point arithmetic is needed. The CONFIGURE_EXTRA_TASK_STACKS parameter controls how many bytes should be added to the task size which is calculated automatically when using the `rtems/-confdefs.h` file. It is used to give a task some margin in case it should make use of more memory than calculated by the automatic mechanism. As seen in the listing, here we give the tasks lots of margin by adding an extra 40 times of the calculated task stack size. This is just a value from the example and can probably be set to 0, which is the default value for this macro. The tasks should have enough stack space without any addition, as is discussed a bit later when showing how a task is created.

The following section contain C preprocessing directives which controls the definition of some macros which in turn control the configuration of timer and UART drivers necessary for the clock and and console driver. The final two macros makes sure that RTEMS includes drivers for the SpaceWire router and the SpaceWire cores on the GR740 SoC. More details on these and all available macros can be found in the RTEMS documentation [11].

Listing 6.2 shows the entry point of the embedded application. RTEMS uses a function called *Init* as its entry point, as opposed to the common *main()*.

Listing 6.2: Initialization of RTEMS application

```
rtems_task Init(rtems_task_argument ignored)
{
  struct router_hw_info router_hw; //contains hw info of spw router
  grspw_work_task_priority   11;

#ifdef DEBUG
  /* Print device topology */
  drvmgr_print_topo();
  rtems_task_wake_after(4);
#endif

  /* Create task */
  rtems_task_create(
        rtems_build_name( 'R', 'M', 'A', 'P' ),
        12, RTEMS_MINIMUM_STACK_SIZE * 2, RTEMS_DEFAULT_MODES,
        RTEMS_FLOATING_POINT, &tid_rmap);

  /* Initialize the SpW router */
  PRINTF("Setting␣up␣SpaceWire␣router\n");
  if (router_setup_custom(&router_hw)) {
    PRINTF("Failed␣router␣initialization\n");
    exit(0);
  }

  /* Initialize the SpW device to use */
  memset(&spw_dev, 0, sizeof(struct grspw_dev_ext));
  if (grspw_dev_init(&spw_dev)) {
    PRINTF("Failed␣to␣initialize␣GRSPW0\n");
    exit(0);
  }

  /* Start DMA channel operations on device */
  if (grspw_start(&spw_dev)) {
    PRINTF("Failed␣to␣initialize␣GRSPW\n");
    exit(0);
  }

  PRINTF("Started␣Successfully\n");

  /* Start the task */
  rtems_task_start(tid_rmap, rmap_task, 0);
  rtems_task_suspend( RTEMS_SELF );
}
```

Init needs one argument, but as is implied by the parameter name it is not used in this application. The main actions of this function is to create the task that is the RMAP application, configuring and starting the SpaceWire core and router, and finally starting the RMAP task before suspending itself. The DEBUG macro is used to control whether debugging information should be printed to console. The macro is defined in `custom_-defines.h` and also controls whether the PRINTF statement is replaced by the normal printf function or by nothing during preprocessing.

There are some things to note in the task creation. All RTEMS tasks have a name associated with them, in this case that name is "RMAP". The following parameter is the task priority, which in this case is set somewhat arbitrarily to 12. In RTEMS, tasks with numerically lower priorities are more important. Therefore the priority of the work task is set to 11, which means it has priority over the main application. This is done in order to allow the work task to quickly process incoming communication and make the necessary moves between DMA queues without being blocked by any command handling (see Section 2.3.1). The following parameter controls how much stack size is allocated for the task. As mentioned, when using the `rtems/confdefs.h` module the macro RTEMS_MINIMUM_STACK_SIZE is automatically calculated by RTEMS. To give some margin for unexpected stack usage, this value is doubled in the function call to give the task twice as much space as expected.The two following parameters specify the attributes of the task, just like in the RTEMS configuration the floating point option could most likely be left out for the purposes of the RMAP application. The final parameter is the address of a variable in which an integer acting like a unique task ID will be saved.

Listing 6.3 shows the main loop of the RMAP task. It mainly handles the data buffers in terms of handing them to and retrieving them from the DMA drivers. The way it currently works is that the task waits for at least one received SpaceWire packet, retrieves it and hands it over to the command handling function which is the entry point of the packet handling. All the RMAP command actions are performed by the command handler and lower-layer functions which also are in charge of creating a reply which may or may not be sent, depending on the acknowledge bit in the RMAP command. In the current implementation, every incoming command is handled and responded to before the next command is handled. An alternative implementation would be to handle all received commands and save the replies, and then schedule the created replies for transmission after every command has been handled. In the case of many commands received at once, the source node would then have to wait for all the commands to be processed before receiving any feedback. The current implementation was chosen since it minimizes the time between the source node sends the command to that it receives a reply.

Listing 6.3: Main loop of RMAP application

```
while (1) {

    /* Schedule rx buffers for reception */
    grspw_dma_rx_prepare(dev->dma[0], 0, &rx_lst, rx_list_cnt);
    rx_list_cnt  0;
    grspw_list_clr(&rx_lst); // descriptors owned by driver, we shouldn't access during rx

    /* Wait until at least 1 packet in RECV queue */
    grspw_dma_rx_wait(dev->dma[0], 1, 0, 31, 0);

    /* Receive into list */
    rxcnt  -1;
    grspw_dma_rx_recv(dev->dma[0], 0, &temp_lst, &rxcnt);

    /* React accordingly to each received packet */
    for (pkt  temp_lst.head, reply  tx_lst.head;
        pkt && reply; pkt  pkt->next, reply reply->next) {

      /* Perform command and create replies */
      ack  command_handler(pkt, reply);

      /* Send reply if ack bit was set in command */
      if (ack > 0){
        txcnt  -1;

        /* Send reply, wait for completion, restore reply buffer */
        grspw_dma_tx_send(dev->dma[0], 0, &tx_lst, 1);
        grspw_dma_tx_wait(dev->dma[0], 0, 0, 0, 1);
        grspw_dma_tx_reclaim(dev->dma[0], 0, &tx_lst, &txcnt);
      }
    }

    /* Prepare rx lists for future transmissions */
    if (rxcnt > 0) {
      grspw_list_append_list(&rx_lst, &temp_lst); //reuse rx buffers
      rx_list_cnt +  rxcnt;
      grspw_list_clr(&temp_lst);
    }
  }
```

Another important thing to understand is the set up of the routing table of the SpaceWire router. This is mainly done through defining a struct later used in a configuration function. The beginning of this struct is seen in Listing 6.4.

Listing 6.4: Beginning of routing table configuring struct

```
static struct router_routing_table routing_table
{
  .flags    ROUTER_ROUTE_FLG_MAP | ROUTER_ROUTE_FLG_CTRL,
  .acontrol    {
        .control_logical    {
    /* 020..027 */ 0x5, 0x5, 0x5, 0x5, 0x5, 0x5, 0x5, 0x5,
    /* 020..02f */ 0x5, 0x5, 0x5, 0x5, 0x5, 0x5, 0x5, 0x5,
    /* 030..037 */ 0x4, 0x5, 0x5, 0x5, 0x5, 0x5, 0x5, 0x5,
    /* 030..03f */ 0x5, 0x5, 0x5, 0x5, 0x5, 0x5, 0x5, 0x5,

    ...
```

This struct allows the developer to configure address control and port routing through the corresponding *.acontrol* and *.portmap* members. Address control lets the developer choose whether packets with a certain address should have e.g. their header deleted or if they are of a higher priority than other packets. Like seen in the listing, each logical address (starting with 0x20   32) have a separate integer defining which of four bit flags should be set or cleared. In the current implementation, the GR740 is given a logical address of 0x68 and the SpaceWire brick acting as a source node is given an address of 0x30. As seen in the listing, at the index corresponding to 0x30 the value differs from the default value. This is because the rightmost bit is cleared in order to disable header deletion. This means that packets coming to the router will not have the first byte deleted before being passed on. The same thing is done at the index corresponding to address 0x68, as header deletion is not used in either direction.

Listing 6.5: Port routing section of routing table configuring struct

```
static struct router_routing_table routing_table
{
    ...

.pmap_logical    {
    /* 020..027 */ 0x00000002, 0x00000002, 0x00000002, 0x00000002, ...
    /* 028..02f */ 0x00000002, 0x00000002, 0x00000002, 0x00000002, ...
    /* 030..037 */ 0x00000002, 0x00000004, 0x00000004, 0x00000004, ...
    /* 038..03f */ 0x00000008, 0x00000008, 0x00000008, 0x00000008, ...
```

...

In Listing 6.5 a nested member of the *.portmap* member is shown. This member controls the port routing of the SpaceWire router. At each index there is a number representing a group of bit flags, where each bit sets the port or ports on which an incoming packet with an address matching that index should be routed to. For example, if one wants to send an outgoing packet on the first SpaceWire port of the router (as in the case in this project) one must set the second least significant bit, i.e. 0x00000002. As seen in the listing this is the value set for the index matching the address 0x30, which is the set address of the SpaceWire brick. Therefore, all packets arriving at the router with the logical address 0x30 will be routed out of the first SpaceWire port. The least significant bit corresponds to the configuration port of the router, which is why the first SpaceWire port is represented by the second bit. Multiple bits can be set if an incoming packet should be routed to several addresses. This is a practical example of the group adaptive routing mentioned in Section 2.1.3. Similarly, the value set at the index matching address 0x68 is set to a value of 0x00000200. This means that all incoming packets with address 0x68 will be routed to the ninth port of the router. Remember that the router on GR740 has eight SpaceWire ports and that the AMBA port numbering starts after the SpaceWire ports. That means that the first AMBA port is port 9, which is why the corresponding bit is set for this port. If the incoming packets would not be routed to an AMBA port, they are not available to the rest of the GR740 and it will appear as if there are no incoming packets.

As previously mentioned, it is not feasible to go through all of the developed code within the scope of this document. Hopefully the comments and variable names in the source code together with the theory given in this document are sufficient for future developers to understand and make use of the existing code. One final piece of code which could be useful to show however, is one of the RMAP command handlers. Specifically, the handler for the read command. The other command handlers are similar in structure, with the main difference being how they react to the command.

Listing 6.6: RMAP read handler

```
/* Handles incoming read commands and formulates a reply.
 * @pkt (struct *): Pointer to the packet descriptor containing
 * the read command.
 * @reply (struct *): Pointer to a descriptor which will hold the
 * reply to the command.
 * */
void rmap_read_handler(struct grspw_pkt *pkt, struct grspw_pkt *reply){

  struct rmap_pkt* p_pkt   (struct rmap_pkt*)pkt->data;
  unsigned int i, data_len, r_adr;

  data_len   arr_to_24bit(p_pkt->hdr.data_len);
  r_adr   arr_to_32bit(p_pkt->hdr.wr_addr);

  /* Reply has no data by default, just data crc (0) */
  *((U8*)reply->data)   0;
  reply->dlen   1;

  /* Check if HW detected errors in transmission */
  if (pkt->flags & (RXPKT_FLAG_EEOP | RXPKT_FLAG_TRUNK)) {
    if (pkt->flags & RXPKT_FLAG_TRUNK){
      PRINTF("Packet␣too␣large\n");
      make_read_reply(reply, pkt, RMAP_ERROR_GENERAL_ERROR);
      return;
    }
    else if (pkt->flags & RXPKT_FLAG_EEOP){
      PRINTF("EEP␣detected␣in␣packet\n");
      make_read_reply(reply, pkt, RMAP_ERROR_EEP);
      return;
    }
  }

  /* Verify the destination key */
  if (p_pkt->hdr.dst_key !  RMAP_DEST_KEY){
    PRINTF("Incorrect␣destination␣key:␣\%d\n", p_pkt->hdr.dst_key);
    make_read_reply(reply, pkt, RMAP_ERROR_INVALID_DEST_KEY);
    return;
  }

  /* Verify data location and length */
  if ((r_adr + data_len) >  MEM_END || r_adr < MEM_BEGIN){
    PRINTF("Cannot␣read␣from␣unauthorised␣memory␣area\n");
    make_read_reply(reply, pkt, RMAP_ERROR_COM_NOT_IMP_AUTH);
    return;
  }
  if(data_len > MAX_RMAP_DATA_SIZE){
    PRINTF("Too␣large␣chunk␣of␣data␣to␣read\n");
    make_read_reply(reply, pkt, RMAP_ERROR_COM_NOT_IMP_AUTH);
    return;
  }

  //All checks passed, now we perform the command

  /* Read the requested data into reply buffer */
  for (i   0; i < data_len; i++){
    *((U8*)reply->data + i)   *((U8*)r_adr + i);
  }

  /* Calculate and append data CRC to data */
  *((U8*)reply->data + data_len)   RMAP_CalculateCRC(reply->data, data_len);

  reply->dlen   data_len + 1; //+1 from data crc

  make_read_reply(reply, pkt, RMAP_COMMAND_SUCCESS);
  return;
}
```

All incoming data of a SpaceWire packet is put in the data field of the RX descriptor. As a first step, the content of this field is interpreted as a struct that has the same structure as a generic RMAP command. This is convenient since it allows the developer to reference a certain byte by naming the field instead of counting bytes. The data length and read address values in a read command is split over several bytes. Therefore, in the next step functions are called for conveniently interpreting these multi-byte values as single integers. Next, it is made sure that the data field of the reply points to a valid value. This is zero by default since a data CRC of zero is used in the reply even if no data could be read. The length of data to be sent is set accordingly.

As the next step, the function checks for errors detected by hardware. One of these is if the incoming packet could not fit in the given buffer and therefore was truncated. A read command using logical addressing has a non-variable size of 16 bytes and should be able to fit in the given data buffers. The developer must make sure of this when initializing the data buffers. If there for some reason is not enough space to accept the read command packet, this is treated as a general error (since it does not fit into the defined error categories of RMAP) and a read reply is created with a status code indicating this. If the DEBUG macro is defined, an error message will be printed to console. Then the handler returns, and the main loop will check whether the reply should be sent or not. This is the general series of events every time an error in the command is found. If an

EEP marker is found, something went wrong during transmission and a corresponding reply is formulated before the handler returns.

In the next step, it is verified that the command sent the same destination key as defined by the application. This functions as an extra verification that the source and destination node expected to communicate with each other, meaning that a source node with no information about the destination key can not get the destination node to perform any RMAP commands.

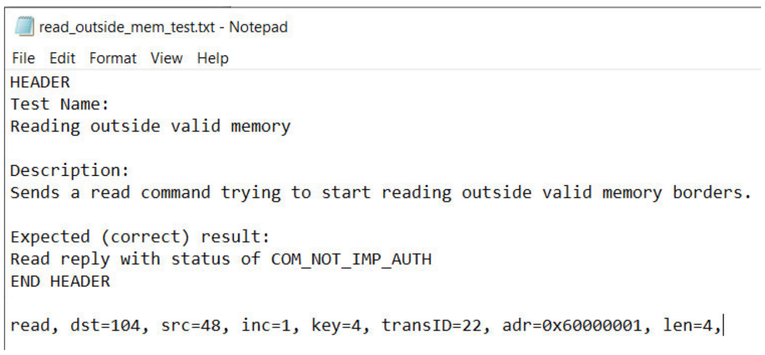
The next block verifies that the request of reading data will not lead to the source node accessing areas outside of the defined authorized memory. If that check passes the command is valid and the read can be performed. The data field of the reply is defined as a pointer to void and must first be cast to a pointer to an unsigned byte. The i variable is then used for indexing the bytes and values are copied from the desired memory location into the data buffer which will be used for transmission. After the data has been read and added, the data CRC is calculated and appended to the reply. Finally, the data length field is updated and a reply indicating successful command execution is created.

7 Test

In order to test the functionality of the developed embedded application a test-bench was developed. The test-bench is a useful tool for creating and performing automatic tests which results in log files with information about the command that was sent and the reply that was received as a response to the command. This section describes typical test-bench usage, and what the test-bench indicates regarding the current state of the embedded application.

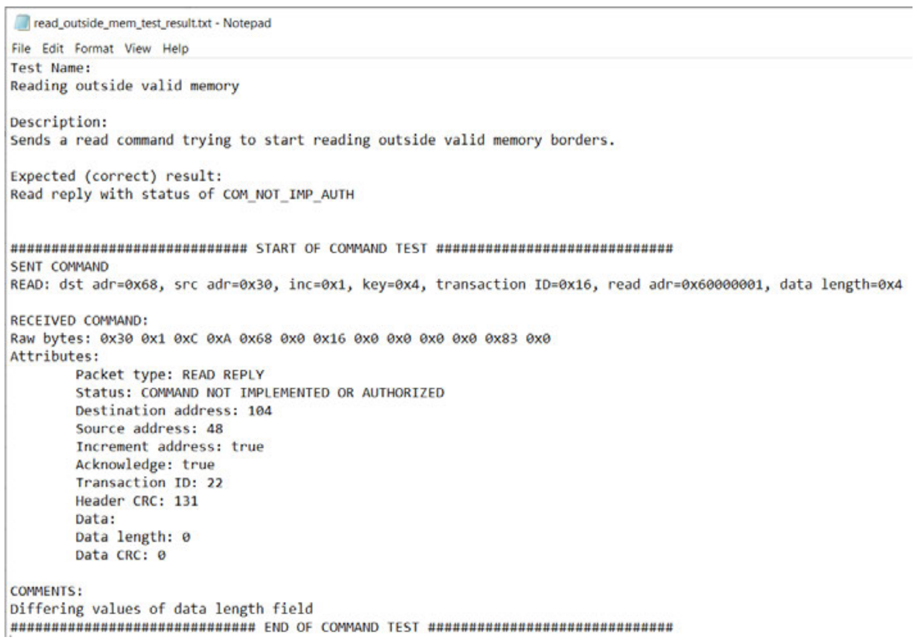
Usage of the test-bench assumes that all connections between the development PC, SpaceWire MK3 brick and GR-CPCI-GR740 are set up and that the embedded application and test-bench both agree on the logical addresses used. Other dependencies between the embedded code and the test-bench are the value of the destination key, and what section of memory is authorized for RMAP commands to use. A possible future simplification would be to have the two projects share one file containing the macro definitions that represent values to be shared between the embedded code and the test-bench, thus minimizing the manual maintenance needed by the developer.

The test-bench uses test scripts created by the user to create and send RMAP commands via the SpaceWire MK3 brick. The test scripts therefore need to contain information on what type of command should be sent as well as custom values for the relevant fields. Every test script can contain several commands. The test scripts also contain a header, marked by the *HEADER* and *END HEADER* tags. Everything that is between these two tags is simply treated as a string and copied to the output log file created as a response to the execution of the test script. This is convenient since it allows the test script author to define what is the purpose of the test and what is the expected result. This text will then be present in both the test script and the resulting log file. This is important since the test-bench is not as general as to detect any failing tests automatically. It simply logs the sent commands and resulting replies and it is then up to the tester to verify that was was received is correct. The test-bench does however include some general checks of values that should always behave a certain way. For example, the transaction ID of a reply should always match that of the sent command. This is a good candidate for a general check, and is implemented in the test-bench. If one or more of these general checks fail, it is reported to the user via the console and is logged under the header *COMMENTS* in the output log file. The output log files will be saved in a directory called `./output` within the same directory as the test script files. Within that directory, the log files will be saved in a nested directory whose name is derived directly from the system time at the start of the test. An example output log file directory would therefore be `./output/(28_-07_2020)_(08_07_39)` which is the date in day_month_year format and time in hour_minute_seconds format. An example test script and the resulting output log file can be seen in Figure 7.1 and Figure 7.2.


read_outside_mem_test.txt - Notepad
File Edit Format View Help

```
HEADER
Test Name:
Reading outside valid memory

Description:
Sends a read command trying to start reading outside valid memory borders.

Expected (correct) result:
Read reply with status of COM_NOT_IMP_AUTH
END HEADER

read, dst=104, src=48, inc=1, key=4, transID=22, adr=0x60000001, len=4,
```

Figure 7.1: An example test script for use with the test-bench.

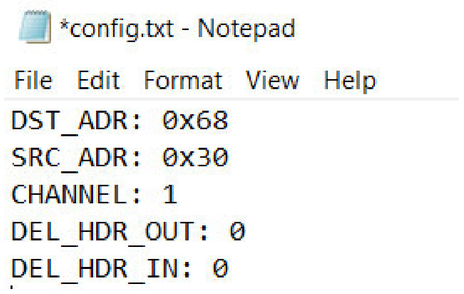


read_outside_mem_test_result.txt - Notepad
File Edit Format View Help

```
Test Name:
Reading outside valid memory

Description:
Sends a read command trying to start reading outside valid memory borders.

Expected (correct) result:
Read reply with status of COM_NOT_IMP_AUTH


############################# START OF COMMAND TEST #############################
SENT COMMAND
READ: dst adr=0x68, src adr=0x30, inc=0x1, key=0x4, transaction ID=0x16, read adr=0x60000001, data length=0x4

RECEIVED COMMAND:
Raw bytes: 0x30 0x1 0xC 0xA 0x68 0x0 0x16 0x0 0x0 0x0 0x0 0x83 0x0
Attributes:
        Packet type: READ REPLY
        Status: COMMAND NOT IMPLEMENTED OR AUTHORIZED
        Destination address: 104
        Source address: 48
        Increment address: true
        Acknowledge: true
        Transaction ID: 22
        Header CRC: 131
        Data:
        Data length: 0
        Data CRC: 0

COMMENTS:
Differing values of data length field
############################## END OF COMMAND TEST ##############################
```

Figure 7.2: The resulting output file after performing the script from Figure 7.1.

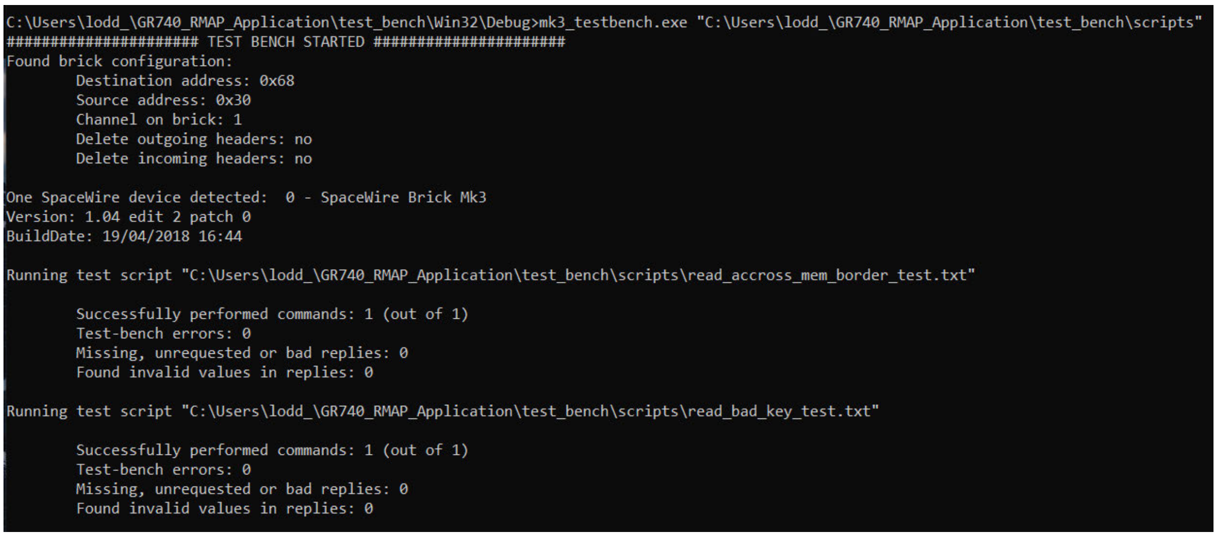
The test-bench will perform all the test scripts found in the given directory. It finds test scripts by assuming all text files whose name end in `*_test.txt` is a test script. It also needs to find a configuration file within the same directory called `config.txt`. This file contains information on the logical addresses to be used, as well which channel of SpaceWire MK3 brick should be used and if header deletion should be enabled or not. An example configuration file is seen in Figure 7.3.



```
DST_ADR: 0x68
SRC_ADR: 0x30
CHANNEL: 1
DEL_HDR_OUT: 0
DEL_HDR_IN: 0
```

Figure 7.3: A configuration file for a suite of test scripts.

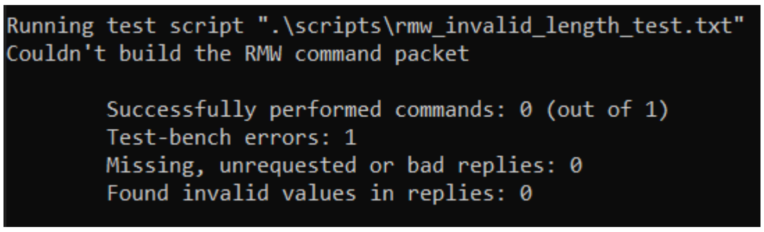
The test-bench is started through either compiling and running the code from within the Visual Studio project, or simply using the created executable. In either case, the test-bench needs to be started with one argument, namely the path to the directory containing the test scripts. An example of a successful start is seen in Figure 7.4.



```
C:\Users\lodd_\GR740_RMAP_Application\test_bench\Win32\Debug>mk3_testbench.exe "C:\Users\lodd_\GR740_RMAP_Application\test_bench\scripts"
##################### TEST BENCH STARTED #####################
Found brick configuration:
        Destination address: 0x68
        Source address: 0x30
        Channel on brick: 1
        Delete outgoing headers: no
        Delete incoming headers: no

One SpaceWire device detected:  0 - SpaceWire Brick Mk3
Version: 1.04 edit 2 patch 0
BuildDate: 19/04/2018 16:44

Running test script "C:\Users\lodd_\GR740_RMAP_Application\test_bench\scripts\read_accross_mem_border_test.txt"

        Successfully performed commands: 1 (out of 1)
        Test-bench errors: 0
        Missing, unrequested or bad replies: 0
        Found invalid values in replies: 0

Running test script "C:\Users\lodd_\GR740_RMAP_Application\test_bench\scripts\read_bad_key_test.txt"

        Successfully performed commands: 1 (out of 1)
        Test-bench errors: 0
        Missing, unrequested or bad replies: 0
        Found invalid values in replies: 0
```

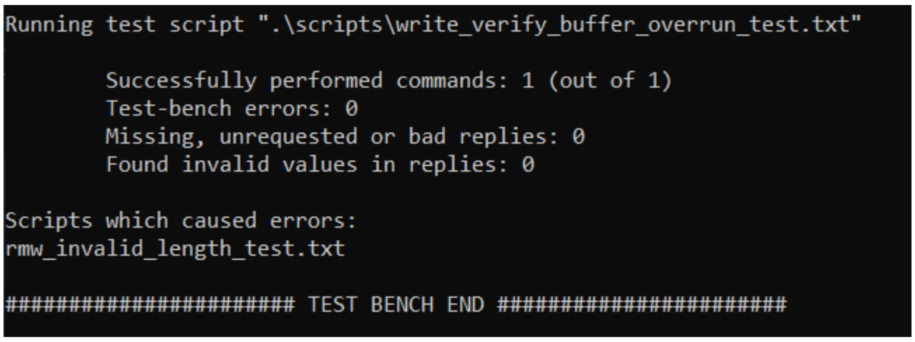
Figure 7.4: A successful start of the test-bench.

After a successful start, the test-bench will print the found configuration to the console, as well as the found SpaceWire device and the hardware version and build date. After this the found test-scripts are parsed and turned into RMAP commands one by one. The test-bench tracks different types of errors and present them to the user. The first category is test-bench errors. These are errors coming from the test-bench itself, such as unexpected code failures. They are mainly found by functions returning error codes when unsuccessful. The next error category is missing, unrequested or bad replies. These are errors such as the test-bench timing out whilst waiting for an expected reply, or conversely receiving a reply when non should be received. A ‘bad reply’ is within this context a received reply that for whatever reason could not be parsed into a valid RMAP packet. The last error category is the number of automatic checks that failed, such as the transaction ID check previously mentioned. Errors from the two last error categories are added to the relevant log file. Scripts which caused any of these errors will be logged at the end of the test execution, to highlight log files of extra importance to the tester. Figures 7.5 and 7.6 shows how such an error is reported to the user.



```
Running test script ".\scripts\rmw_invalid_length_test.txt"
Couldn't build the RMW command packet

        Successfully performed commands: 0 (out of 1)
        Test-bench errors: 1
        Missing, unrequested or bad replies: 0
        Found invalid values in replies: 0
```

Figure 7.5: Error in one of the test scripts.



```
Running test script ".\scripts\write_verify_buffer_overrun_test.txt"

        Successfully performed commands: 1 (out of 1)
        Test-bench errors: 0
        Missing, unrequested or bad replies: 0
        Found invalid values in replies: 0

Scripts which caused errors:
rmw_invalid_length_test.txt

####################### TEST BENCH END #######################
```

Figure 7.6: End of the test-bench output when errors were detected.

At the moment of writing, the existing test scripts that are provided together with the test-bench source code test the following aspects:

- Sends a read command that wants to start reading outside valid memory borders. Should cause a read reply with status code for *Command not implemented or authorized.*
- Sends a read command that wants to start reading within valid memory but with a data length that will cross valid memory borders. Should cause a read reply with status code for *Command not implemented or authorized.*
- Sends a read command with an invalid destination key. Should cause a read reply with a status code for *Invalid destination key.*
- Sends a write command (assumed to work correctly) to write a specific value at a place in memory. Followed by sending a read command at the same memory address, in order to make sure that the read data matches what was sent in the read command. Both write and read replies should have status codes for *Successful command execution.*

- Sends a otherwise valid write command with an incorrect data CRC value, with Verify Before Write bit set. Should cause a write reply with a status code indicating *Invalid data CRC.*
- Sends a otherwise valid write command with an incorrect data CRC value, with Verify Before Write bit cleared. Should cause a write reply with a status code indicating *Invalid data CRC*, but also the data should have been written. Therefore a read command follows to allow the tester to make sure the data was written, even if incorrect.
- Sends a write command with invalid destination key and Acknowledge bit set. Should cause a write reply with a status code indicating *Invalid destination key.*
- Sends a write command trying to write outside valid memory. Should cause a write reply with a status code indicating *Command not implemented or authorized.*
- Sends a valid write command with Acknowledge bit and Verify Before Write bit set. Should cause a write reply with status code indicating *Successful command execution.*
- Sends a valid write command with the Acknowledge bit cleared. No reply should be received (within defined timeout).
- Sends a write command with too much data for the data buffers (decided by macros in the embedded application). Should cause a write reply with a status code indicating *Verify buffer overrun.*
- Sends a otherwise valid RMW command with an invalid data CRC. Should cause a RMW reply with a status code indicating *Invalid data CRC.*
- Sends a RMW command with an incorrect destination key. Should cause a RMW reply with a status code indicating *Invalid destination key.*
- Sends a RMW command with an invalid data length. Should cause a RMW reply with a status code indicating *RMW data length error.*
- Sends a RMW command trying to perform the command outside valid memory area. Should cause a RMW reply with a status code indicating *Command not implemented or authorized.*

- Sends a completely valid RMW command. Should cause a reply containing some data and a status code indicating *Successful command execution.*
- A write command is sent to set certain data in memory. Then a valid RMW command is sent, which should cause a RMW reply containing the same data which was written by the previous command.

At the moment of writing, all the existing tests pass. This is determined by a manual inspection of the output log files.

8 Summary

At the conclusion of this project it can be stated that the main goals presented in Section 1.1 and Chapter 4 have been achieved. A software driver for the RMAP protocol running on the GR740 SoC acting as a destination node is developed and behaves according to the standard when communicating with a PC using the SpaceWire MK3 brick. Some compromises exist due to the usage of the DMA-based SpaceWire packet driver provided together with RCC, causing the order of certain checks to be slightly different than described in the RMAP specification. The difference is that in this implementation incorrect data will never have been written at the point where the error is detected and this is deemed as an acceptable deviation from the standard as it should have no effect on normal RMAP usage.

The developed test-bench allows convenient testing of the RMAP software driver by reading test scripts adhering to a certain format and using the SpaceWire MK3 brick to send the RMAP commands described in these test scripts. The results are logged in output log files and act as a great tool for a developer or tester to see how the RMAP driver responds to certain RMAP commands, thus helping in the validation of correct behavior.

8.1 Outlook

At the end point of the project there still exists several areas of improvement which could be the starting point for any further additions to the developed driver. They are listed here.

- There are dependencies between the test-bench and the RMAP driver such as logical addresses and valid memory area which need to be maintained manually by the developer when using the system. The maintenance could be simplified by having one shared file from which these values are taken by both the RMAP driver and the test-bench.

- The tester needs to go through the output log files manually to determine if any errors occurred in the RMAP driver. An addition to the test-bench could be to allow the test script author to define expected values in certain fields of the RMAP reply. This could help automatic detection of bugs and further simplify the testing process.
- Currently the RMAP driver does not respond to changes in the *Increment* bit. This is a deviation from the RMAP standard and should be added in the future.
- The application image size can likely be heavily decreased by modifying the values for the RTEMS configuration to differ from the example values and better reflect the needs of the RMAP driver.

Bibliography

[1] Cobham Gaisler AB: *GR-CPCI-GR740 Development Board User's Manual.* 1.9. URL https://www.gaisler.com/doc/gr740/GR-CPCI-GR740-UM.pdf, 2018

[2] Cobham Gaisler AB: *GR740 Quad Core LEON4 SPARC V8 Processor: Preliminary Data Sheet and User's Manual.* 2.0. URL https://www.gaisler.com/doc/gr740/GR740-UM-DS-1-10.pdf, 2018

[3] Cobham Gaisler AB: *RCC: RTEMS-5 Cross Compiler (RCC): User's Manual.* 1.3-rc6. URL https://www.gaisler.com/anonftp/rcc/doc/rcc-1.3.pdf, 2018

[4] Cobham Gaisler AB: *GRMON3: A debug monitor for LEON-based computer systems and SOC designs based on the GRLIB IP library: User's Manual.* 3.1.0. URL https://www.gaisler.com/doc/grmon3.pdf, 2019

[5] Cobham Gaisler AB: *Software IDE Quick Start Guide: User's Manual.* URL https://www.gaisler.com/eclipse/qsg_sw_ide.pdf, 2019

[6] Cobham Gaisler AB: *GRLIB IP Core User's Manual.* 2020.1. URL https://www.gaisler.com/products/grlib/grip.pdf, 2020

[7] Dischereit, Kai: *Studienarbeit Inbetriebnahme des Mikrocontrollerboards GR740.* 2020

[8] European Cooperation for Space Standardization: *Remote memory access protocol: ECSS-E-50-11 Draft F.* URL http://spacewire.esa.int/content/Standard/documents/SpaceWire%20RMAP%20Protocol%20Draft%20F%204th%20Dec%202006.pdf, 2006

[9] European Cooperation for Space Standardization: *Space Engineering: SpaceWire protocol identification: ECSS-E-ST-50-51C.* 2010

[10] European Cooperation for Space Standardization: *Space Engineering: SpaceWire: Links, nodes, routers and networks.* Rev. 1. 2019

[11] On-Line Applications Research Corporation: *RTEMS C User's Guide.* 4.10.1. URL https://docs.rtems.org/releases/rtems-docs-4.11.2/c-user/index.html, 2011

[12] Parkes, Steve: *SpaceWire User's Guide.* URL https://www.star-dundee.com/wp-content/star_uploads/2019/05/SpaceWire-Users-Guide.pdf, 2012. – ISBN 978-0-9573408-0-0

[13] STAR-Dundee Limited: *STAR-System C++ API.* 2018

[14] White, Elecia: *Making embedded systems.* 1. Beijing and Farnham : O'Reilly, 2011, 2012. – ISBN 978-1-449-30214-6

A FAQ

During the software driver development many issues were encountered and resolved. This appendix describes some of the problems and their solutions. Another important resource for resolving development issues is the FAQ of the software IDE quick start guide [5].

- **Problem:** `printf` is not putting any output on the console, even though GDB was started with the correct options.
  Solution: All strings to be printed must be terminated with newline character.

- **Problem:** Compiling gives the error message 'NULL undefined'.
  Solution: Make sure to include `stdlib.h` and `stdio.h` before any custom header files.

- **Problem:** When trying to debug, the error 'Failed to execute MI command: -file-exec-and-symbols' appears.
  Solution: Make sure to have MinGW installed directly under C:\(not in Program Files or another directory with spaces in the name) and add the bin path under `Project->Properties->C/C++ Build->Environment` in the PATH variable.

- **Problem:** When the test-bench received SpaceWire packets and tried interpreting it as RMAP, the function failed and simply said the status code was 'GENERAL_-ERROR' but didn't give more clues to what was wrong with what should have been a normal RMAP reply.
  Solution: In this case this was due to the header CRC being incorrectly calculated in the RMAP driver on the GR740.

- **Problem:** Starting GRMON gives: 'JTAG Instruction register length detection failed'.
  Solution: Make sure FTDI driver is installed and that the development PC has been restarted afterwards.

Declaration

I declare that this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used.

Hamburg	29/09/2020	[redacted]
City	Date	Signature