OULUN AMMATTIKORKEAKOULU

Kalle Perälä

CPRI IP CORE REGISTER ANALYSIS FOR DEBUGGING

CPRI IP CORE REGISTER ANALYSIS FOR DEBUGGING

Kalle Perälä
Bachelor's Thesis
Spring 2024
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Option of Software Development

---

Author(s): Kalle Perälä
Title of thesis: CPRI IP CORE REGISTER ANALYSIS FOR DEBUGGING
Supervisor(s): Ari-Pekka Taskila, Janne Kumpuoja
Term and year: Spring 2024
Number of pages: 42

The objective of this thesis was to develop an automated tool for analyzing Layer 1 Common Public Radio Interface (CPRI) IP core register dumps at Nokia. The aim of the tool was enhancing the efficiency and accuracy of base station diagnostics and debugging. Traditionally, understanding the state of base stations using the register dumps involved decoding the binaries and interpreting the output data manually, a process often hindered by its time-consuming nature and susceptibility to human error due to the vast amount of information and variety of data formats.

The Register Dump Analyzer was created to address these challenges by providing an automated solution for reading register binaries and producing an easily interpretable output. The tool was developed in Python as it is a commonly used language at Nokia. Python also allows a rapid iteration process in the development due to the interpreted nature of the language.

The tool's development involved integrating an existing binary decoder to handle initial parsing of the binary data, followed by the creation of a custom parsing algorithm to manage the non-standard text format of the data. This algorithm enabled the tool to efficiently process and analyze the binary decoders output and works as a solid base for further development. Additionally, the tool is designed to alert users about issues detected in the register values, further streamlining the debugging process.

Keywords: Base station, CPRI, IP core, Memory registers

PREFACE

My thanks go to Ari-Pekka Taskila, Markku Kulmunki and Janne Kukkonen for the great thesis topic, and the general support and guidance I received during the project. A special thanks to Juha Pirttikangas for the valuable professional advice on the CPRI related technical topics.

I am also grateful to Nokia Oyj and the CPRI team for offering me the opportunity to conduct this thesis work. The experience and knowledge I have gained during this process have provided an excellent foundation for the beginning of my professional career.

CONTENTS

TERMS AND ABBREVIATIONS

4G	Fourth generation of cellular network technology.
5G	Fifth generation of cellular network technology, successor to 4G.
AxC	Antenna carrier represents the digital baseband (IQ) user plane data necessary for the reception or transmission of a single carrier at an independent antenna element.
CPRI	Common Public Radio Interface is a standard interface for communication between a Radio Equipment Control (REC) and Radio Equipment (RE).
CRX	Receiving side of the CPRI link.
CTX	Transmitting side of the CPRI link.
DL	Downlink is the transmission path from the base station to the UE in a cellular network.
IP Core	Intellectual Property Core is a block of logic created to execute specific operations.
L1	Layer 1 is responsible for transceiving raw bit streams over a physical connection in network communications.
OSI	Open Systems Interconnection model is a seven-layer framework used to visualize and design a network system.
RE	Radio Equipment is the component in a cellular network that performs radio transmission and reception.

REC	Radio Equipment Control oversees controlling and managing the radio equipment in a cellular network.
Register dump	Register dump is a binary file that contains a memory structure and data, which was captured at some point from a base station.
UE	User Equipment refers to the devices used by consumers to communicate on a network. These include smartphones, tablets, and other mobile devices.
UL	Uplink is the transmission path from the UE to the base station in a cellular network.
VSB	Vendor Specific Bytes are typically used for proprietary features or functions that are unique to a vendor's hardware or software, allowing for customization and optimization beyond standard specifications.

1 INTRODUCTION

Understanding the state of a base station is important for effective problem-solving. However, this process can be challenging and time-consuming due to the vast amount of data and information contained in logs and snapshots. Dumping a segment of the base station's memory offers a snapshot of the current alarms and values, aiding in better understanding the status of the system. The issue here lies in the fact that this data is often encoded in various formats, necessitating the consultation of additional documentation for decoding. This is why manual interpretation of register values is often inefficient and prone to errors.

To address these challenges, the Register Dump Analyzer was developed. This tool automates the process of reading CPRI IP core register binaries, producing outputs that are easily interpretable. Python was chosen as the development language due to its widespread use at Nokia and its suitability for rapid, iterative development.

During this thesis, a test case is also created for the Register Dump Analyzer to make sure that the tool stays functional during further development. The test is a golden sample test where the test runs the Register Dump Analyzer and compares the generated output to a pre-verified one called the golden sample. This type of testing makes a developer more aware of their changes to the output of the tool.

This thesis was undertaken at Nokia. It is structured into three main parts: the first introduces the basics of base station layer 1 and CPRI; the second details the development of the Register Dump Analyzer, including its various components; and the final part discusses the results and conclusions drawn from this work.

2 BASE STATION LAYER 1 AND THE OSI MODEL

A mobile telecommunication system is a communication system managed by network operators like Vodafone or AT&T, officially referred to as a Public Land Mobile Network (PLMN). It consists of four key components: the Core Network (CN), Radio Access Network (RAN), management system, and the user's device, formally known as User Equipment (UE) as seen in Figure 1. The Core Network facilitates data transfer between the mobile and external networks like the Public Switched Telephone Network (PSTN) and the Internet. It also manages the mobile's external communications and stores subscriber data for the network operator. (Cox 2020, Chapter 1.1.1.)

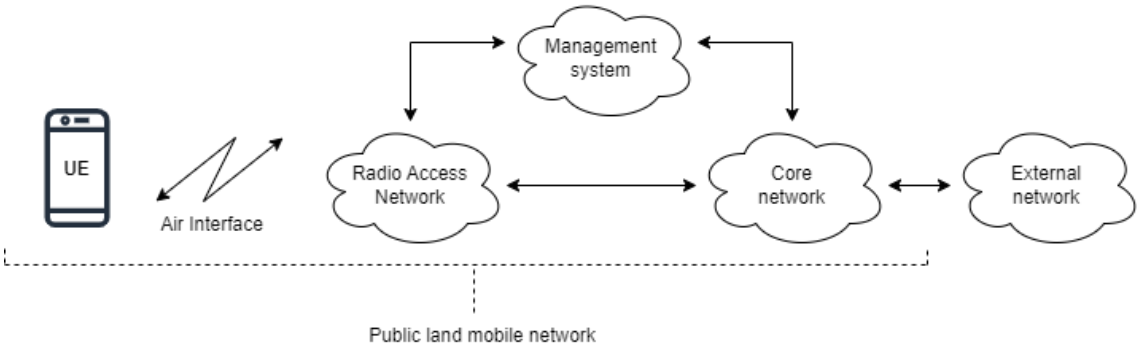

FIGURE 1. Architecture of a mobile telecommunication system (adapted from Cox 2020, Chapter 1.1.1)

RAN manages radio communications within the network, operating via two main interfaces. It connects to the Core Network through the backhaul interface, while interacting directly with mobile devices over the air interface, also known as the radio interface. The direction from the network to the UE is known as the downlink (DL) or forward link, the route from the UE to the network is called the uplink (UL) or reverse link. (Cox 2020, Chapter 1.1.1.)

The base station is the most important element of the radio access network. Base stations transmit and receive data via one or more carrier frequencies on the air interface, with each carrier having its designated bandwidth. For instance, a 5G base station might utilize a 3500 MHz carrier frequency with a 40 MHz bandwidth, spanning from 3480 to 3520 MHz. These base stations oversee one or more cells, each involving radio transmissions with specific carrier frequencies and bandwidths, covering distinct areas. On a single radio frequency, a base station can manage multiple cells, also known as sectors, by transmitting in different directions, typically with three sectors per

base station, each spanning a 120° arc. A base station can also manage multiple cells in the same direction using different radio frequencies to prevent signal interference. (Cox 2020, Chapter 1.1.3.)

Open Systems Interconnection (OSI) model, depicted in Figure 2, is a framework for creating an open communication system standard. The OSI model comprises seven layers, represented as a vertical stack. These layers are integral to communication standards in connected computer systems, with only the physical layer directly facilitating communication between the systems. Each of the layers uses the services of the layer below it, with data passed as packages containing headers and data. The model itself is not a complete standard but serves as an abstract plan, relying on protocols to implement computer communication. (Costa 1998.)

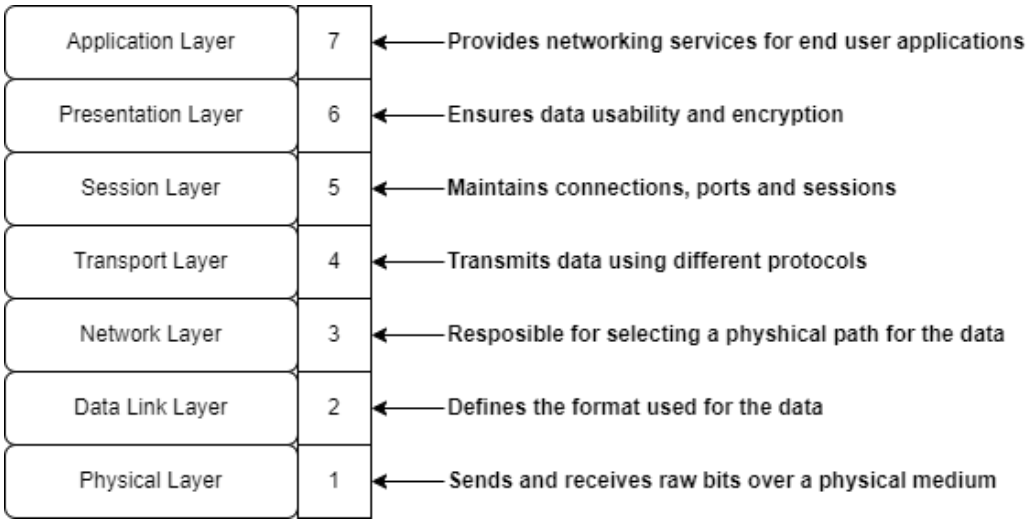

FIGURE 2. Seven layers detailed in the OSI model (adapted from Cloudflare n.d.)

This thesis focuses on CPRI (Common Public Radio Interface) register analysis on the physical layer, also known as layer 1. Costa (1998) describes that layer 1 defines the mechanical, electrical, procedural, and functional aspects of the communication link, including hardware specifications, signal transmission methods, and cable types. It manages the activation and maintenance of physical connections, preparing the transmission medium for synchronous and asynchronous two-way communication.

3 COMMON PUBLIC RADIO INTERFACE

CPRI (n.d.) website explains that the Common Public Radio Interface (CPRI) is an industry collaboration with contributions from Ericsson, Huawei, NEC, and Nokia. CPRI intends to provide an open and accessible specification for the internal interface in radio base stations. This interface facilitates the connection between the Radio Equipment Control (REC) and Radio Equipment (RE), as illustrated in Figure 3. The interface, also known as fronthaul, is digital and serialized, and it supports both single-hop and multi-hop network topologies.

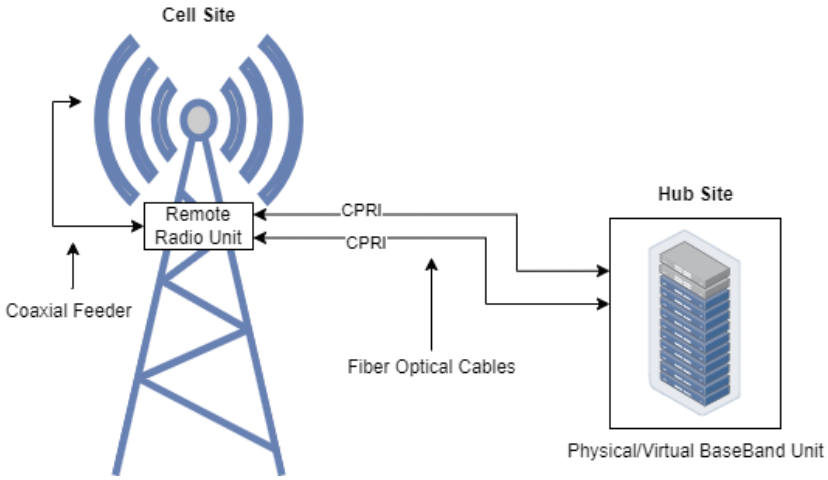

FIGURE 3. CPRI in a radio access network (adapted from Mukherjee 2021)

The radio base station system comprises two primary subsystems: REC and RE, also referred to as "nodes". A functional radio base station system must have at least two nodes, one of each type. (CPRI 2011, 6.)

The term "link" in the CPRI specification indicates the bidirectional interface between two directly connected ports, typically between an REC and an RE. Each link consists of two ports with asymmetrical functions: a master and a slave. The REC typically houses the master port, while the RE holds the slave port. This configuration plays an essential role in synchronization, C&M (Control & Management) channel negotiations during the startup phase, reset indications, and the overall startup sequence. In terms of directionality, "downlink" refers to the direction from REC to RE for a logical connection, and "uplink" means the direction from RE to REC for the same. Figure 4 depicts these basic CPRI definitions. (CPRI 2011, 7-9.)

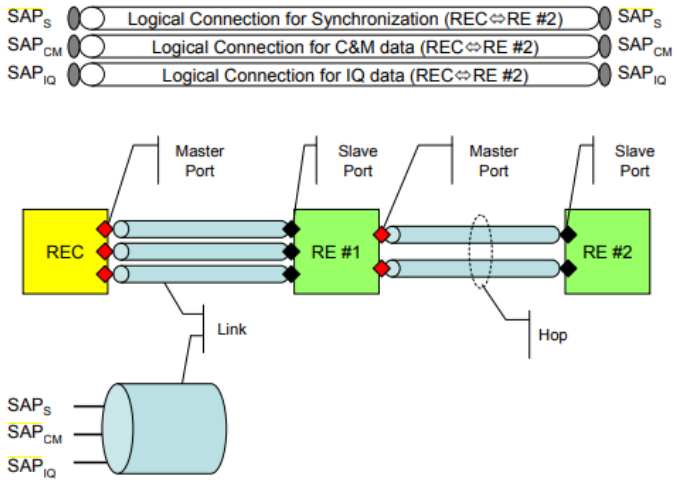

FIGURE 4. Illustration of basic CPRI definitions (CPRI 2011)

3.1 CPRI line bit rates

The CPRI (2011) specification defines several line bit rates to ensure flexibility and cost efficiency in communications. As outlined in Table 1, these bit rates range from 614.4 Mbit/s to 9830.4 Mbit/s. The rates are chosen such that the basic Universal Mobile Telecommunications System (UMTS) chip rate of 3.84 Mbit/s can be efficiently derived, considering the 8B/10B line coding specified in clause 36 of IEEE 802.3-2005. For instance, the 1228.8 Mbit/s rate, represented as Option 2 in Table 1, when processed through an 8B/10B encoder, corresponds to an encoder rate of 122.88 MHz. A subsequent frequency division by 32 yields the basic UMTS chip rate of 3.84 megachips per second. (CPRI 2011, 29.)

TABLE 1. Different CPRI line rates (adapted from CPRI 2011)

Option	Rate (Mbit/s)	Multiplier of 614.4 Mbit/s
1	614.4	1x
2	1228.8	2x
3	2457.6	4x
4	3072.0	5x
5	4915.2	8x
6	6144.0	10x

7	9830.4	16x

3.2 CPRI frame structure

The CPRI (2011, 6-7) specification identifies different data flows or "planes". The control plane manages the data flow used in call processing. The management plane handles the management information essential for the operation, administration, and maintenance of the CPRI link and its nodes. The user plane facilitates data transfer between the radio base station and the mobile station. Additionally, there is the synchronization data flow, which ensures the transfer of synchronization and timing details between nodes. Data within the user plane is conveyed as In-phase/Quadrature (IQ) data. Each IQ data flow represents the data of one antenna for one carrier, known as the antenna-carrier (AxC). An AxC represents the digital baseband (IQ) U-plane data necessary for the reception or transmission of a single carrier at an independent antenna element.

CPRI communication is structured around frames to efficiently transmit data. One CPRI frame encapsulates data over a duration of 10 ms, comprising 150 hyper frames. Each hyper frame contains 256 basic frames, with every basic frame consisting of 16 words. Figure 6 illustrates this hierarchy between the different frame types. (CPRI 2011, 46.)

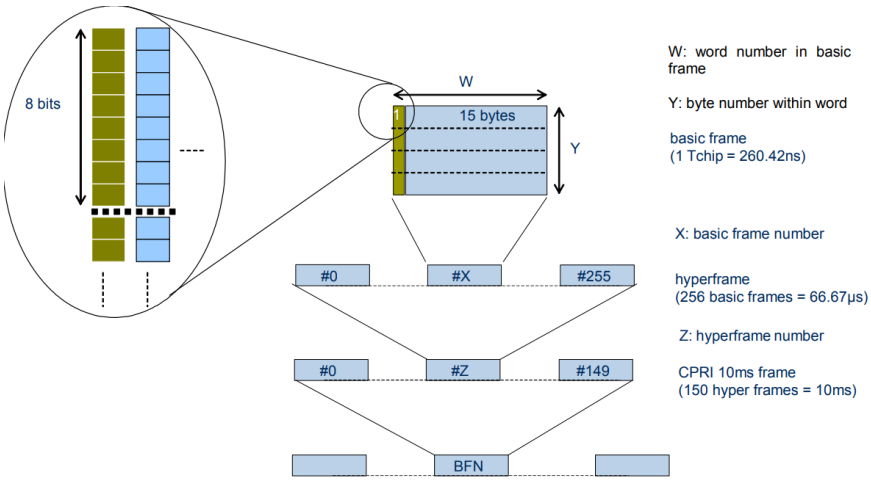

FIGURE 6. CPRI frame hierarchy (CPRI 2011)

In a basic frame, one of the words manages control data, while the remaining 15 words relay user plane IQ data, as seen in Figure 7. The frame structure's integrity is maintained with a temporal

length of 260.416667 ns for each basic frame. When the CPRI line rate increases, the frame structure and duration stay the same and only the word size increases. (CPRI 2011, 31., RF Wireless World n.d.).

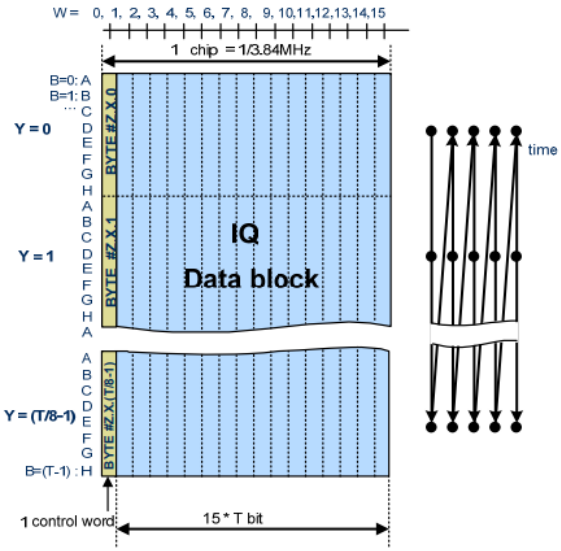

FIGURE 7. Generic basic frame structure for different CPRI line rates (CPRI 2011, 36)

The hyperframe structure contains the 256 control words derived from the basic frames. These control words are organized into 64 subchannels, with each subchannel containing 4 control words, as illustrated in Figure 8. (CPRI 2011, 46-47.) While these subchannels serve various functions, this thesis will mostly focus on the Synchronization, Vendor-specific, and Fast Control & Management subchannels.

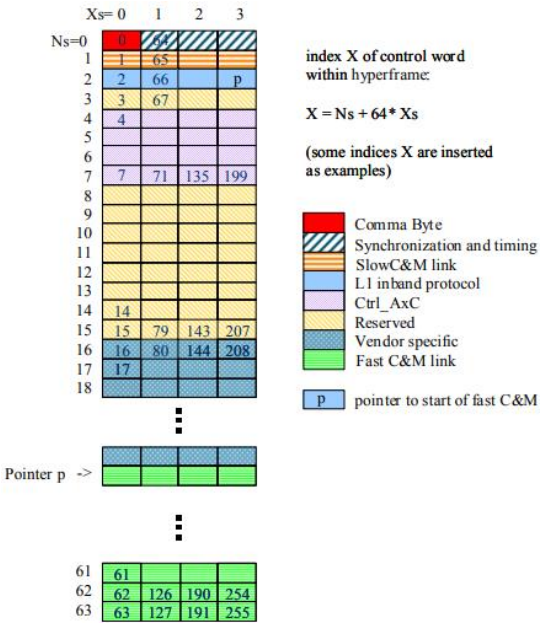

FIGURE 8. Subchannels within one hyperframe (CPRI 2011, 47)

The CPRI specification also introduces a coordinate system represented as Z.X.Y, which is used to navigate the CPRI frame structure. In this system, "Z" represents the hyperframe number, "X" indicates a specific basic frame within the hyperframe, and "Y" points to a particular byte within a word of the basic frame. This approach ensures accurate referencing of different data within the CPRI frame structure. (CPRI 2011, 46, 107.)

3.3 AxCs with different line bit rates

CPRI line bit rate and the carrier type define the number of AxCs a basic frame can hold. Table 2 presents how many specific LTE (Long Term Evolution) AxCs are supported by each line rate, assuming that we are using 15-bit IQ samples. Different LTE carriers have varying sampling frequencies that impact the number of carriers that can be used. (Anritsu 2016, 5, 9-10.)

The required sample count can be calculated by: $f \times t$, where f is the sampling frequency of a carrier and t is the temporal length of the basic frame. Taking LTE 10 MHz carrier as an example, it has a 15.36 MHz sampling frequency. This means with the basic frame length of 260.416667 the required sample count can be calculated to be 4. As seen in Figure 6, IQ data in one basic frame is 15 times the word length in bits, which in case of 10x line bit rate is 80. This means bits of IQ

data fit in one basic frame when using the 10x line bit rate. Now it is possible to calculate the number of possible 15-bit IQ samples by dividing 1200 with the 30 bits of combined I and Q data which results in 40 samples. This means one basic frame can contain 10 LTE 10 MHz carriers when using the 10x line bit rate. It is also possible to use compression to fit more carriers in to a specific line bit rate. Compression utilizes advanced algorithms to decrease data volume and size of the carriers, while maintaining signal integrity. (Anritsu 2016, 5-11.)

TABLE 2. Number of supported LTE carriers using different CPRI line bit rates (adapted from Anritsu 2016)

Line rate	LTE 5 MHz	LTE 10 MHz	LTE 15 MHz	LTE 20 MHz
1x	2	1	0	0
2x	4	2	1	1
4x	8	4	2	2
5x	10	5	3	2
8x	16	8	5	4
10x	20	10	6	5
16x	32	16	10	8

3.4 CPRI link synchronization and maintenance

As shown in chapter 3.2, each CPRI hyperframe contains four "synchronization and timing" control words. These control words are used to ensure the CPRI link synchronization between the REC and RE and keep the communication timing correct. The first synchronization word is located at #Z.0.0, which is the very first control word of each hyperframe. It houses the special sync byte called K28.5 and it indicates the start of the hyperframe. The second subchannel is at #Z.64.0 and it contains the hyperframe number known as the HFN. This value ranges between 0 and 149, resetting with every new CPRI frame. The last two synchronization subchannels, #Z.128.0 and #Z.192.0, are reserved for the CPRI frame number. (CPRI 2011, 47-48).

The synchronization and link maintenance process in CPRI includes a few main states, Loss of Signal (LOS), Loss of Frame (LOF), and HFNSYNC. As can be seen in Figure 9, the process can be represented as a state machine diagram which starts at a state where both LOS and LOF alarms

are on, indicating that the communication is not established. The synchronization process checks that the passing hyperframes contain the correct K28.5 synchronization byte exists at the correct position. When the first check succeeds, the LOS alarm is turned off, if the second check is also successful, the LOF alarm gets turned off. The process will reach the HFNSYNC state, which indicates the link is synchronized correctly. If the K28.5 byte does not exist or is incorrect, the state machine will go back up one step and turn the alarms back on if necessary. (CPRI 2011, 24-25, 49, 64).

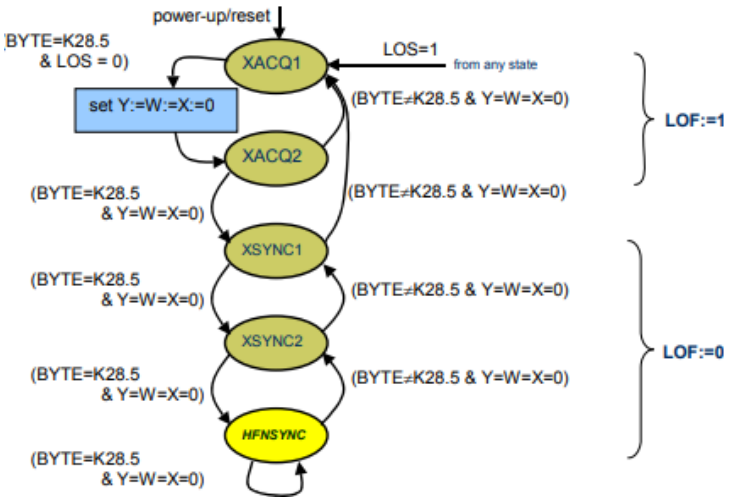

FIGURE 9. CPRI synchronization state machine (CPRI 2011, 64)

The synchronization state machine shown earlier is part of the larger start-up sequence illustrated in Figure 10. After HFNSYNC is reached and line bit rate is determined in state B, the system moves to the state C, also called as the protocol setup. During this phase, the REC and RE will determine the used CPRI protocol version, either 1 or 2. The different protocol versions have different capabilities, but this thesis will not go into further detail about them. If the protocol version can be agreed on between the REC and the RE, the start-up sequence will move to the state D or C&M plane setup. In state D, the master and slave ports will negotiate what control and management channel bit rate to use. If the negotiation is successful, the process proceeds to the state E. If the master port does not propose any C&M channel, or the slave port refuses the proposed channel, the system will enter state G. The state G is a passive link state, meaning that the interface is not carrying the C&M plane. If the system enters state E, the master and slave ports negotiate the specifics of CPRI usage. They exchange information about capabilities and limitations, resulting in a preferred configuration of the CPRI, which can include vendor-specific elements. The final state is the operation state F where the C&M channel is established and the CPRI link is operational. (CPRI 2011, 72-79.)

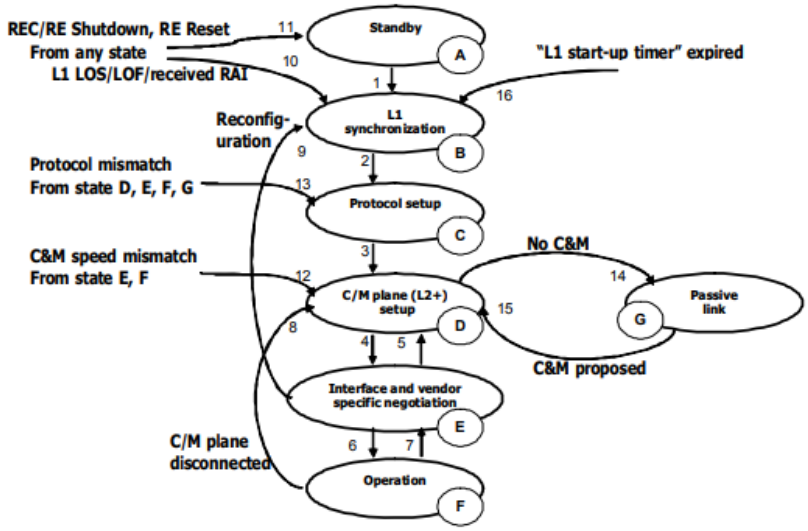

FIGURE 10. Start-up sequence state machine (CPRI 2011, 72)

The layer 1 start-up timer plays a key role in this sequence. It is activated during various transitions in the start-up process. If this timer expires before the completion of certain stages, it can lead to the restart of the start-up procedure. The expiration time of this timer is vendor-specific, but its primary function is to ensure that the start-up process does not get stuck indefinitely due to faults or mismatches in layer 1 protocol, C&M channel bit rate, or C&M type. (CPRI 2011, 72.)

4 CPRI INTELLECTUAL PROPERTY CORE AND REGISTER DUMPS

This thesis is about analyzing the memory registers of an CPRI intellectual property core (IP core). IP cores are reusable blocks of logic that implement specific functions or operations on a semiconductor chip (Awati 2022). Using IP cores as part of the semiconductor chip design streamlines workflows and accelerate design and development (Butler 2023).

In the context of this thesis, the CPRI IP core serves as a hardware accelerator block on the base station SoC (System on Chip), executing CPRI-specific features and operations at the hardware level to enhance their speed and efficiency. The specific IP core in this thesis is intellectual property of Nokia, albeit other manufacturers such as Intel and AMD also provide IP core solutions tailored for CPRI use cases (AMD n.d., Intel n.d.). The CPRI IP core not only implements numerous vendor-specific features but also covers the functionalities lined in the CPRI specification. This thesis will concentrate on the IP core register values and states related to the CPRI specification, rather than vendor specific ones.

The focus of this thesis is to analyze register dumps of the base station memory, which are created as part of a snapshot process that captures the current state of the base station. These dumps contain the register values from the IP Core block in binary form and can be converted into human-readable format with internal tools. Examples of these register values include, but are not limited to, the current LOS or LOF status bits, state of the CPRI synchronization state machine, and AxC status. (Nokia 2023.)

By analyzing the register, it is possible to figure out for instance, whether the link configuration aligns with the expected setup, the types of AxCs in use, and which are active, or are any error states present. This information provides the user of the analysis tool with a comprehensive view of the base station's internal operations at the time the snapshot was taken. This aids the debugging process and facilitates the identification of potential problem areas. (Nokia 2023.)

5 DEVELOPING THE REGISTER DUMP ANALYZER

This chapter covers the development of an automated tool to parse and analyze CPRI IP core register dumps, called the Register Dump Analyzer. The register dumps are one of multiple tools used when diagnosing the state of a base station, and an automated program for this purpose would help to analyze these dumps further. As the register dumps are captured and stored in a binary format, the implementation of this tool is based on an existing internal binary decoder tool, which reads the actual register dump binaries of the CPRI IP core instances and creates human readable output. After generating a readable output, the data could be parsed into this analysis tool easily.

5.1 Design and structure of the data

The analysis tool was designed to parse and assess a text file generated by the binary decoder tool. This file contains the memory structures and values from each IP core instance on the SoC. After the analysis tool had parsed this data into an internal data structure, it needed to analyze and compare various values from the memory registers. Figure 11 is a high-level sequence diagram of the designed work logic behind the analysis tool and its relationship with binary decoder. As seen in the illustration, the tool is accessible via a shell interface.

When a user initiates the tool with a specific command and desired parameters, the Register Dump Analyzer will count the number of register dump binaries in the user-specified path. With this information, the tool can know which IP core version is in use. This information, along with the path specified by the user is then relayed to binary decoder, which proceeds to convert the binary files into a text format. After the binary decoder has completed the decoding, it outputs a temporary output file which the analysis tool will now read and parse the data into an internal data structure. When the tool finishes the parsing process, it will delete the temporary output file and move into an analysis phase.

During the analysis phase, the tool reads and compares different values in the registers. Comparing the values to example data, the tool can interpret different states of the IP Core instances and for example, provide the user an analysis of the current state of the CPRI links or AxCs.

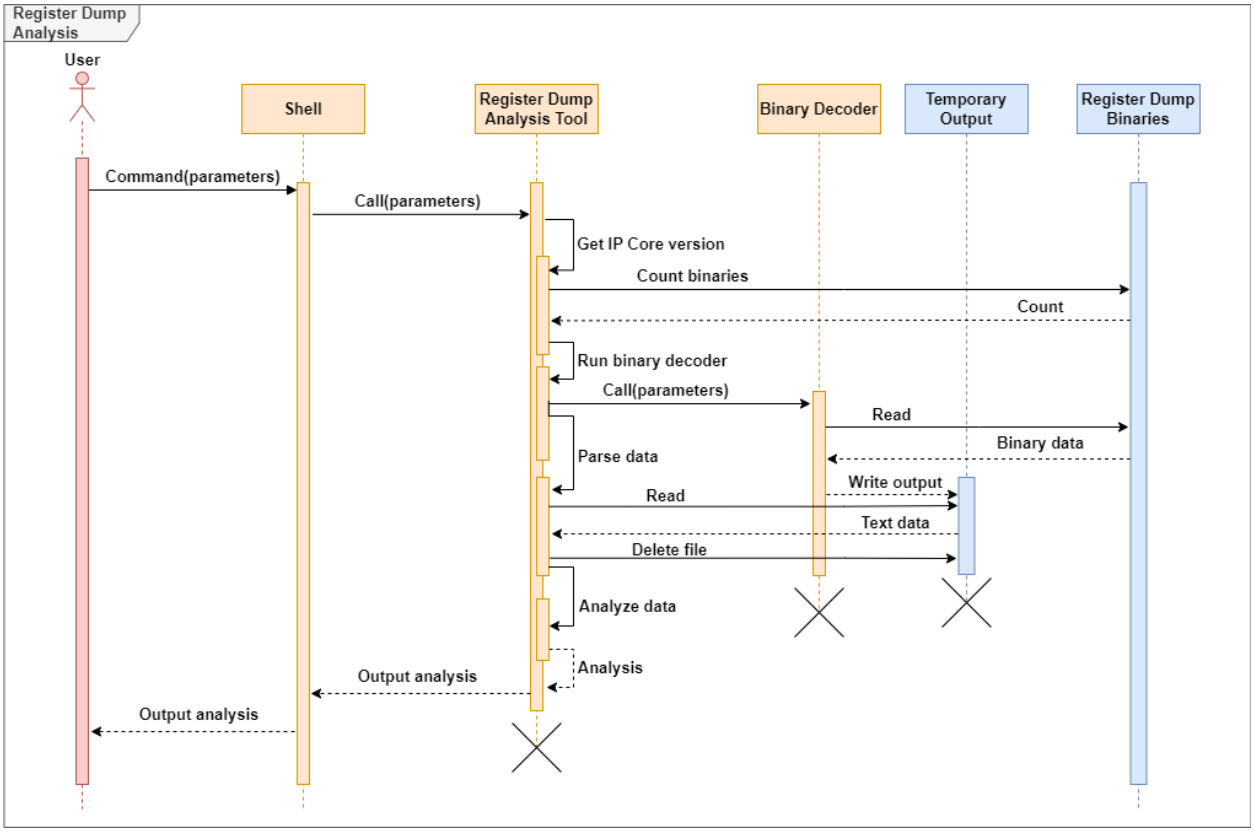

FIGURE 11. Register dump analysis sequence diagram.

In the beginning of the design work, significant effort was devoted to understanding the internal memory structure of the CPRI IP cores and the output produced by binary decoder. The temporary file contains an output that closely resembles a JSON-like format. The analysis tool must be capable of dynamically interpreting this format, as there are hundreds of thousands of lines in the temporary file. Figure 12 is a screenshot of sample data structured in an equivalent way to the binary decoder output. As seen in the screenshot, there is a list of object-like structures for each IP core instance. Within these instance structures, lists, objects and key-value pairs comprising decimal, hexadecimal, and actual values can be observed. Given this complexity, the tool needed to automatically create new lists, objects, and key-value pairs into an internal data structure during the parsing process. We also wanted to store the actual names of lists, objects, and values, as they provide straightforward access to them within the tool's code. This approach ensures that every bit of information in the registers is parsed dynamically, which also allows us to easily expand the tool in the future.

```
= (Instances *) 0x00000000 = 0
{   structure
 ipCoreInstance[0] =
 {   structure
  ID = 0x0 = 0
  reserved = 0x0 = 0
  CTRL =
  {   structure
   example0 = 0 = 0x0 = Normal
   example1 = 0 = 0x0 = Normal
   reserved0 = 0x0 = 0
  }
  STATUS =
  {   structure
   close0 = 0 = 0x0 = Error
   close1 = 0 = 0x0 = Error
   reserved0 = 0x0 = 0
  }
  RESERVED_AREA0 =
  {   structure
   reserved[0] = 0x0 = 0
   reserved[1] = 0x0 = 0
   reserved[2] = 0x0 = 0
  }
  LINK0 =
  {   structure
   reserved0 = 0x0 = 0
   RATE = 4 = -4 = 0x4 = CPRI_9830_4Mbps_16x
   MODE = 1 = -1 = 0x1 = CPRI
   reserved1 = 0x0 = 0
  }
```

FIGURE 12. Sample data of the binary decoder output.

5.2 Implementation

The implementation of the analysis tool was split into two parts, parsing, and analyzing. In the parsing phase, a Python program was created to read the binary decoder output and store the data into a Python data structure. Python was chosen for the development of the tool because it is already a widely used language at Nokia. The analysis phase was about reading this data and comparing it to known values and figuring out which ones were expected, and OK, and which ones indicated error states. After the analysis was completed, the tool would output this information as a text to the user's shell output or store it into a file.

To create the analysis tool, a few prerequisites were necessary. Firstly, the tool needed a way to determine the IP Core version to run the binary decoder, and it needed to parse command line arguments from the user to find the binaries along with other useful information. The analysis tool also needed to output the analyzed information to the shell or a log file. With this information, the creation of the tool, and the parsing functionality was started.

5.2.1 Entry point of the analysis tool

An entry point was created for the script as shown in Figure 13. This part parses the command line arguments provided by the user by calling the read_arguments() function. Then it stores the returned arguments into a dictionary called “flags” and passes them to the main function. Lastly, a logger is generated by using the Python logging library and the register path is validated in case of a faulty path provided by the user.

```
if __name__ == '__main__':
    parsed_args = read_arguments()
    logger = logging.getLogger(__name__)
    if not parsed_args.register_path.exists():
        logging.error('Register path "%s" is not valid!', str(parsed_args.register_path))
        sys.exit()
    if parsed_args.output:
        logging.basicConfig(
            format='%(message)s',
            filename=str(parsed_args.output),
            filemode='w',
            level=logging.DEBUG,
        )
    else:
        logging.basicConfig(
            handlers=[logging.StreamHandler(sys.stdout)],
            format='%(message)s',
            level=logging.DEBUG,
        )
    if not parsed_args.tx and not parsed_args.rx:
        parsed_args.tx = parsed_args.rx = True

    flags = {
        'print_vsb': parsed_args.vsb,
        'print_axc': parsed_args.axc,
        'print_tx': parsed_args.tx,
        'print_rx': parsed_args.rx,
        'print_all_axcs': parsed_args.print_all_axcs,
    }
    sys.exit(
        main(
            {
                'register_path': parsed_args.register_path,
                'decoder': parsed_args.decoder,
                'output': parsed_args.output,
                'links': parsed_args.links,
                'log_flags': flags,
            },
        ),
    )
```

FIGURE 13. Entry point for the analysis tool.

As shown in Figure 14, the arguments contain options such as the path to the register dumps, path to binary decoder executable (optional), output path (optional) and the specific CPRI links to be

analyzed (optional). Several arguments were also added for analyzing and logging specific information, like AxCs and Vendor Specific Bytes (VSB).

```
def read_arguments():
    parser = argparse.ArgumentParser(description='Register dump analyzer')
    parser.add_argument('register_path', type=Path)
    parser.add_argument('--vsb', action='store_true')
    parser.add_argument('--axc', action='store_true')
    parser.add_argument(
        '-d',
        '--decoder',
        default=DEFAULT_DECODER_PATH,
        help='Path to the decoder executable',
    )
    parser.add_argument(
        '-o',
        '--output',
        default=None,
        help='Output file, if not set output goes to terminal',
    )
    parser.add_argument(
        '-l',
        '--links',
        default=None,
        help='Specific linkIds to be analyzed separated with ","',
        type=lambda s: [int(id) for id in s.split(',')],
    )
    parser.add_argument(
        '--tx',
        action='store_true',
    )
    parser.add_argument(
        '--rx',
        action='store_true',
    )
    parser.add_argument(
        '--print-all-axcs',
        action='store_true',
    )
    return parser.parse_args()
```

FIGURE 14. Function for reading the command line arguments provided by the user.

Figure 15 shows the actual main function that firstly creates an instance of the RegDumpAnalyzer class, where the actual analysis will be done and generated. Then the program calls the get_ip-core_version() method inside that class. The get_ip_core_version() method contains the logic to count the register binaries and it stores that information to a member variable known as ipcore_version. Then the tool will make sure that IP Core version was found and proceed to the parsing part. With the version information along with arguments provided by the user, the main function will

create an instance of the BinaryReader class and pass its output, which is the parsed data, to the analyzer class.

```
def main(args):
    analyzer = RegDumpAnalyzer(args['register_path'], args['links'], args['log_flags'])
    analyzer.get_ipcore_version()

    if analyzer.ipcore_version != IpCoreVersion.UNKNOWN:
        try:
            reader = BinaryReader(
                args['decoder'],
                args['register_path'],
                analyzer.ipcore_version,
            )
            analyzer.analyze(reader.read())
        except (KeyError, ValueError, FileNotFoundError) as err:
            logging.error(err)
```

FIGURE 15. The main function of the analysis tool.

The BinaryReader class which contains the parsing logic is depicted in Figure 16. During the initialization of the class, it assigns the provided arguments to member variables and creates some extra variables needed for the actual parsing process. The class also validates the path to the binary decoder executable to make sure that it exists.

The read() method of the BinaryReader class will run the binary decoder using the run_decoder() method. Inside that method, the binary decoder is ran using the subprocess library provided by Python and the output is directed to a temp file which is named using the TEMP_FILE_NAME variable. After binary decoder has been run successfully, the read() method will call the parse_temp_file() method that executes the actual parsing logic and stores the parsed data into the result[] member variable of the BinaryReader class. The result[] variable will be returned to the caller of the read() method upon successful execution. The read() method was also designed to contain some error handling logic to help with debugging of the tool.

```python
TEMP_FILE_NAME = 'temp_file.h'

...
class IpCoreVersion(str, Enum):
    ExampleVersion1 = '1'
    ExampleVersion2 = '2'
    UNKNOWN = 'unknown'

...
class BinaryReader:
    def __init__(self, decoder_path, binary_directory, ipcore_version):
        self.decoder_path = Path(decoder_path)
        self.binary_directory = Path(binary_directory)
        self.ipcore_versiopn = IpCoreVersion(ipcore_version)
        self.stack = []
        self.result = []

        if not os.path.isfile(self.decoder_path):
            raise FileNotFoundError(f'Path to binary decoder is not valid: "{self.decoder_path}"')

    def read(self):
        try:
            logging.info('Running binary decoder...')
            self.run_decoder()
            logging.info('Parsing output...')
            self.parse_temp_file()
        except subprocess.CalledProcessError as err:
            raise OSError(f'Error while running binary decoder: {err}') from err
        except OSError as err:
            raise OSError(f'Error while reading binary decoder output: {err}') from err
        except KeyError as err:
            raise KeyError(f'Error while parsing the data: {err}') from err

        return self.result
```

FIGURE 16. BinaryReader class and the read() method.

5.2.2 Parsing data from the temporary file

When the BinaryReader class calls the parse_temp_file() method (Figure 17), the tool opens the binary decoder output using the global TEMP_FILE_NAME variable declared earlier. The tool then reads the content of the temporary file line by line and checks if it can find matching patterns on that line. For instance, if the line contains the word “ipCoreInstance”, the analysis tool knows that this is the beginning of the data structure and will change the start_parsing boolean to True.

```
def parse_temp_file(self):
    start_parsing = False
    skip_structure = False

    with open(TEMP_FILE_NAME, 'r', encoding='ascii') as file:
        for line in file:
            line = line.strip()

            if 'ipCoreInstance' in line:
                start_parsing = True
            if 'reserved' in line.lower():
                if line.endswith('='):
                    skip_structure = True
                continue
            if line == '}' and skip_structure:
                skip_structure = False
                continue
            if not start_parsing or skip_structure:
                continue

            self.parse_line(line)

    os.remove(TEMP_FILE_NAME)
```

FIGURE 17. parse_temp_file() method

However, if the line contains the word "reserved", the tool will switch the skip_structure variable to True. Then it uses the continue keyword to skip the rest of the current loop iteration and move to the next line. This behavior was implemented to reduce the size of the internal data structure as the tool does not need the data contained inside the reserved structures or key-value pairs produced by the binary decoder.

If the line does not contain the "reserved" word, the analysis tool will check for a pattern "}". If this pattern is found, it means that some object has ended. If this happens and the skip_structure flag variable is currently "True", it indicates that the reserved structure has ended, and the program can start parsing the lines again.

Lastly, when the flag start_parsing is "True" and skip_structures is "False", the code will move to the parse_line member method (Figure 18). In this method, the tool will first check if there is a new structure, which is the case if the current line ends with the "=" symbol. If this pattern is found, the tool adds a new empty object to the current "stack" and names it accordingly using the first word of the line, which is the structure name in the binary decoder output. The "stack" is a temporary list for handling nested structures and arrays from the binary decoder output.

If the current line contains only the symbol "}" and the "stack" is not empty, the program will know that the current object has ended, and it moves to the parse_end_of_object() method.

```
def parse_line(self, line):
    # Handle new structure
    if line.endswith('='):
        current_obj = {}
        self.stack.append((line[:-1].strip(), current_obj))
    # Handle end of object
    elif line == '}':
        if self.stack:
            self.parse_end_of_object()
    # Handle key-value pairs
    elif '=' in line:
        self.parse_key_value_pair(line)
```

FIGURE 18. parse_line() method.

First, the parse_end_of_object() method (Figure 19), will pop the last item from the stack and check if there is still something inside the stack. If there is, the tool can assume that it is currently handling a nested structure, or an array and it will handle those accordingly. If the stack is empty, the object will be appended to the result list.

```
def parse_end_of_object(self):
    obj_key, obj = self.stack.pop()
    if self.stack:
        _parent_key, parent_obj = self.stack[-1]
        # Handle arrays of object structures
        if '[' in obj_key and ']' in obj_key:
            array_key = obj_key.split('[')[0]
            if array_key not in parent_obj:
                parent_obj[array_key] = []
            parent_obj[array_key].append(obj)
        else:
            if obj:
                parent_obj[obj_key] = obj
    else:
        if obj:
            self.result.append(obj)
```

FIGURE 19. parse_end_of_object() method.

If there is no new structure or an end of an object is found during the execution of the parse_line() method, the tool will check if the symbol "=" can be found inside the current line. If it is found, the tool will move to the parse_key_value_pair() method (Figure 20). In this method, the tool extracts the key and value from the line. Then it will check if it can find the symbols "["and "]" in the line. These symbols indicate that the values are part of an array structure and must be stored as a list. If the pattern is not found, the value will be saved as a key-value pair to the current object.

```
def parse_key_value_pair(self, line):
    key, value = line.split('=', 1)
    value = value.split('=')[-1].strip()
    key = key.strip()
    if self.stack:
        current_obj = self.stack[-1][1]
        #Handle array like keys
        if '[' in key and ']' in key:
            array_key = key.split('[')[0]
            if array_key not in current_obj:
                current_obj[array_key] = []
            current_obj[array_key].append(value)
        else:
            current_obj[key] = value
```

FIGURE 20. parse_key_value_pair() method.

When the tool finishes parsing the line, it will iterate to the next line and so on until the file runs out of lines. After the whole file is parsed, the BinaryReader class deletes the temporary file and returns the result variable which contains all the parsed data.

5.2.3 Analyzing and logging the parsed data

As seen previously in Picture 4, the parsed data returned by the binary reader is then passed to the RegDumpAnalyzer class using the analyze() method (Figure 21). The analyzer will firstly log the IP Core version that is being analyzed and then check if the user provided any specific links to be analyzed. If the user did not provide any links, it will use a default set of link ids according to the IP Core version.

When the requested_link_ids variable is set, the analyzer will enter a loop that goes through each IP Core instance in the parsed data. During each of these loops, the tool will call the add_links() method, which will then add the data from each link in the current IP Core instance as a separate entry to a list variable called “links”. Lastly, the analyze() method will loop through each link in the list and use the log() method to print out the analysis if the current link’s link_id is present in the requested_link_ids list.

```
class RegDumpAnalyzer:
    def __init__(self, path, link_ids, log_flags):
        self.register_path = path
        self.ipcore_version = IpCoreVersion.UNKNOWN
        self.requested_link_ids = link_ids
        self.log_flags = log_flags
        self.current_link_id = 0
        self.links = []

    def analyze(self, parsed_data):
        logging.info('Analyzing IP Core Version: %s', self.ipcore_version.value)
        if not self.requested_link_ids:
            if self.ipcore_version == IpCoreVersion.ExampleVersion1:
                self.requested_link_ids = [0, 1, 2]
            else:
                self.requested_link_ids = [0, 1, 2, 3, 4, 5, 6]
        for instance in parsed_data:
            self.add_links(instance)
        for link in self.links:
            if link.link_id in self.requested_link_ids:
                link.log()
```

FIGURE 21. The RegDumpAnalyzer class and the analyze() method.

Each link will be added to the "links" list as an object of a class "Link" (Figure 22). It contains several member variables such as "rate" and "mode" which will be logged when the log() method is called. The Link class also creates a new object of the CpriRegisters class and passes CPRI related register data to it. Also, an if statement was added to the log method that will check the current mode of the link and only call the log() method in the CpriRegisters class if the link is in CPRI mode.

```
class Link:
    def __init__(self, regs, log_flags, link_id):
        self.link_id = link_id
        self.cpri_registers = self.create_cpri_registers(regs, log_flags)
        self.rate = regs['link']['RATE']
        self.mode = regs['link']['MODE']

    def create_cpri_registers(self, regs, log_flags):
        return CpriRegisters(regs['cpri'], regs['stream'], log_flags)

    def log(self):
        logging.info('\nlinkId: %s', self.link_id)
        logging.info('Rate: %s', self.rate)
        logging.info('Mode: %s', self.mode)
        if self.mode == 'CPRI':
            self.cpri_registers.log()
```

FIGURE 22. Links class.

The tools internal hierarchy and structure were designed to contain a class for every IP core register structure that is going to be analyzed. Using this implementation, the tool can be expanded easily by creating new classes and including them in the hierarchy. The whole internal variable structure of the analysis tool is visualized as an object diagram in Figure 23.

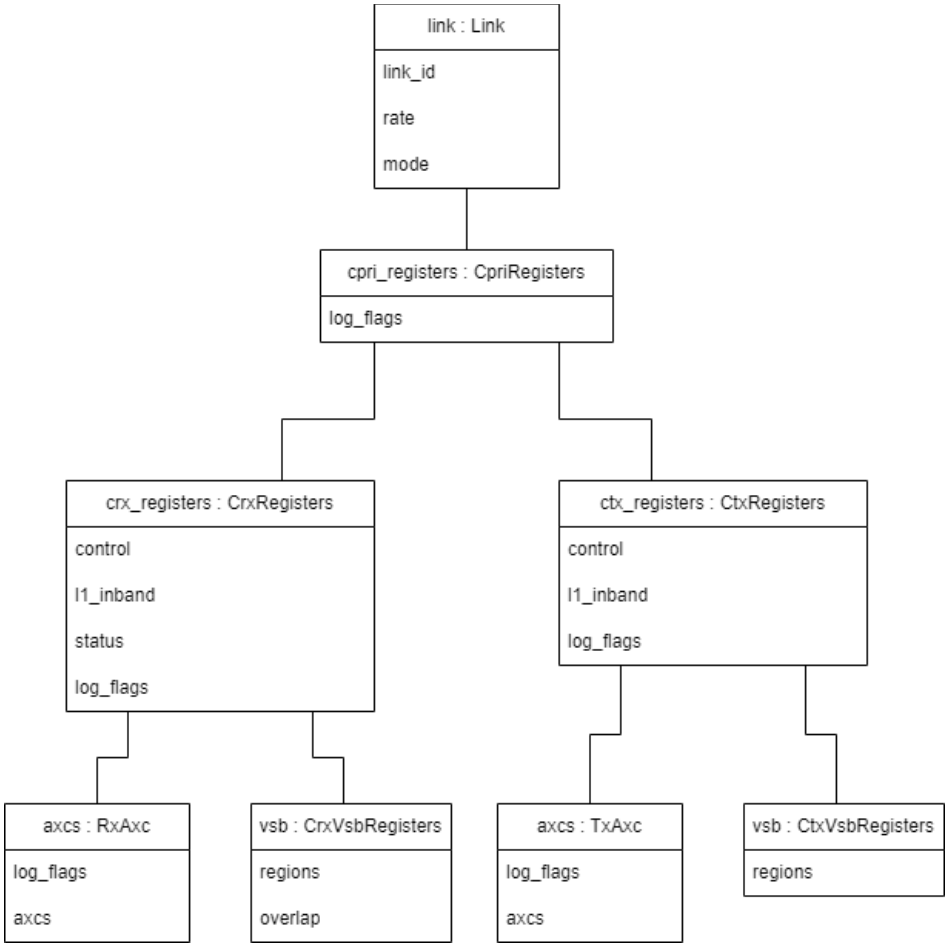

FIGURE 23. Object diagram detailing the internal variable structure of the Register Dump Analyzer.

The CPRI register class contains two objects for CPRI Transmit (CTX) and CPRI Receive (CRX). CTX contains the transmission side (REC to RE) register data of the CPRI link and CRX has the reception side (RE to REC) register data. This class also has a log() method that call the CTX and CRX log() methods if those are enabled, and the user has enabled the –rx or –tx logging flags with command line arguments as shown in Figure 24.

```
class CpriRegisters:
    def __init__(self, cpri_regs, stream_regs, log_flags):
        self.crx_registers = CrxRegisters(cpri_regs['RX_Registers'], log_flags)
        self.ctx_registers = CtxRegisters(
            cpri_regs['TX_Registers'],
            stream_regs,
            log_flags,
        )
        self.log_flags = log_flags

    def crx_enabled(self):
        return self.crx_registers.enabled()

    def ctx_enabled(self):
        return self.ctx_registers.enabled()

    def log(self):
        if self.crx_enabled() and self.log_flags['print_rx']:
            self.crx_registers.log()
        else:
            logging.info('CPRI RX Disabled!')
        if self.ctx_enabled() and self.log_flags['print_tx']:
            self.ctx_registers.log()
        else:
            logging.info('CPRI TX Disabled!')
```

FIGURE 24. CpriRegisters class.

The internal memory structure of the IP Core is not identical for the CTX and CRX sides. This is why separate classes were created for them as seen in Figure 25. However, both do contain an object for the VSB and AxC data. Both classes also have a log() method to print out the analysis and an enabled() method to check if the CRX or CTX side is enabled in the current link.

```
class CtxRegisters:
    def __init__(self, ctx_regs, stream_regs, log_flags):
        self.control = ctx_regs['control']
        self.l1_inband = ctx_regs['l1']
        self.log_flags = log_flags
        self.vsb = CtxVsbRegisters(ctx_regs)
        self.axcs = TxAxc(ctx_regs['AxC_Registers'], stream_regs, log_flags)


...
class CrxRegisters:
    def __init__(self, crx_regs, log_flags):
        self.control = crx_regs['control']
        self.l1_inband = crx_regs['l1_status']
        self.status = crx_regs['status']
        self.log_flags = log_flags
        self.vsb = CrxVsbRegisters(crx_regs)
        self.axcs = RxAxc(crx_regs['AxC_Registers'], log_flags)
```

FIGURE 25. CtxRegisters and CrxRegisters class definitions.

The log() method is where the actual analysis mostly happens. These methods contain checks and calls to decoding methods to see if specific bits are set in the registers. With this information, the tool can then print information to the user about the status of the CPRI link. The log() method of the CrxRegisters class is shown in Figure 26. As can be seen from the code, the prints out values from the parsed registers and checks if values like LOS or LOF are on. If the tool finds values that can indicate a problem, it will print out a warning for the user.

```
def log(self):
    logging.info('CPRI RX:')
    logging.info(' Protocol Version: %s', self.check_protocol_versions())
    logging.info(' Ethernet: %s', decode_eth_pointer(int(self.l1_inband['ethernet'])))
    logging.info(' CRX Status: %s', self.decode_state_byte(self.status['state']))
    if self.status['lof'] == '1' or self.status['los'] == '1':
        logging.info(' Warning, CRX LOS or LOF bit enabled:')
        logging.info(' LOF: %s LOS: %s', self.status['lof'], self.status['los'])
    enabled_notifications = self.check_received_l1_notifications()
    if enabled_notifications:
        logging.info(' Warning, Received Enabled L1 Notifications: %s', enabled_notifications)
    if self.log_flags['print_vsb']:
        self.vsb.log()
    if self.log_flags['print_axc']:
        self.axcs.log()
```

FIGURE 26. The log() method inside the CrxRegisters class.

The logging methods for both CRX and CTX call several decoding and parsing methods during the logging process. Those methods were created to read larger or multiple register values and return a string to be logged. For example, as can be seen in Figure 26, there is a call to the decode_state_byte() method, which is a decoding method that reads all the bits from a one-byte sized value and checks if they match a predetermined map of “states”. The state byte represents the current state of the CPRI synchronization state machine which was explained in chapter 3.4. The decode_state_byte method is shown In Figure 27. These logging methods also initiate the log() methods inside the AxC and VSB classes.

```
def decode_state_byte(self, state_byte):
    byte_to_state = {
        0b00000000: 'XACQ1',
        0b00000001: 'XACQ2',
        0b00000010: 'XSYNC1',
        0b00000100: 'XSYNC2',
        0b00001100: 'LOF',
        0b00001111: 'LOF',
    }
    state_str = ''
    for byte, state in byte_to_state.items():
        if int(state_byte) == byte:
            state_str = state
    return state_str
```

FIGURE 27. The decode_state_byte() method of the CrxRegisters class.

The AxC and VSB classes are also different for the CRX and CTX sides. The AxC classes (Figure 28) both have a list of AxCs that are filled by the parse_axcs() methods. These parsing methods take the AxC related registers as a parameter and combine all needed values from them as a single list of dictionaries. It was implemented this way to split some of the functionality from the logging methods and to make the code more readable.

```
class RxAxc:
    def __init__(self, regs, log_flags):
        self.log_flags = log_flags
        self.axcs = self.parse_axcs(regs)


class TxAxc:
    def __init__(self, axc_regs, stream_regs, log_flags):
        self.log_flags = log_flags
        self.axcs = self.parse_axcs(axc_regs, stream_regs)
```

FIGURE 28. Definitions of the RxAxc and TxAxc classes.

The AxC parsing method of the RxAxc class is demonstrated in Figure 29. The method starts with creating a loop of the AxC registers provided by the CrxRegisters class and during each loop it first checks if the AxC is enabled, or if the user has used the –print-all-axcs argument to log also the disabled AxCs using. Then the tool creates a new AxC object with all the necessary information

and appends it to the parsed_axcs list. The parsed_axcs list is returned after the method has gone through all the registers.

```
def parse_axcs(self, axcs):
    parsed_axcs = []
    for index, axc in enumerate(axcs):
        ctrl_reg = axc['CTRL']
        container_reg = axc['CONTAINER']
        axc_enabled = ctrl_reg['ENABLE'] == 'enabled'
        if axc_enabled or self.log_flags['print_all_axcs']:
            parsed_axcs.append(
                {
                    'id': index,
                    'enable': axc_enabled,
                    'decompress': ctrl_reg['DECOMPRESS'],
                    'interface': ctrl_reg['INTERFACE'],
                    'type': ctrl_reg['TYPE'],
                    'address': ctrl_reg['ADDR'],
                },
            )
    return parsed_axcs
```

FIGURE 29. The parse_axcs() method of the RxAxc class.

In the AxC classes, separate methods were created for logging and generating the output string, called generate_log_string() and log() (Figure 30). This way the intended output format is more readable in the code. If there are any AxCs in the list, the log() method will loop through them and call the generate_log_string() method using the current AxC as a parameter. In the generate_log_string() method the tool checks if there could be any disabled AxCs in the list by checking the log flags. If the —print-all-axcs parameter was used, the method assigns either "(Enabled)" or "(Disabled)" string to the axc_status_str variable, depending on the value of axc['enable']. Finally, all the information from the axc variable is added to a string which is returned and printed out in the log() method.

```
def generate_log_string(self, axc):
    axc_status_str = ''
    if self.log_flags['print_all_axcs']:
        axc_status_str = '(Enabled)' if axc['enable'] else '(Disabled)'
    log_string = (
        f'  AxC {axc["id"]}{axc_status_str}:{NL}'
        f'    Decompression: {axc["decompress"]}{NL}'
        f'    Interface: {axc["interface"]}{NL}'
        f'    Type: {axc["type"]}{NL}'
        f'    Address: {axc["address"]}{NL}'
    )
    return log_string

def log(self):
    logging.info(' RX AxCs:')
    if self.axcs:
        for axc in self.axcs:
            logging.info(self.generate_log_string(axc))
    else:
        logging.info('  All RX AxCs Disabled!')
```

FIGURE 30. Logging related methods of the RxAxc class.

The registers that are analyzed in the VSB classes contain status and control values for the VSB functionality and information flow of the CPRI IP Core. However, this thesis primarily focuses on functionality related to the CPRI specification rather than on vendor-specific functionalities, so those classes are not displayed here.

5.2.4 Usage of the Register Dump Analyzer

While developing the Register Dump Analyzer, the focus was on ensuring the output was as readable as possible, while also containing a significant amount of information. Figure 31 represents an example output from the tool when not using any of the optional command line parameters. The screenshot shows that the IP core version is detected correctly. The example output shows three links from which two are not in use. The third link has a linkId of 2 and is using the 16x CPRI line

bit rate, which is the highest rate as detailed in chapter 3.3. The output is also indented to make it easier for the user to understand which information belongs to which link.

```
Running binary decoder...
Parsing output...
Analyzing Connectivity Version: ExampleVersion1

linkId: 0
Rate: RATE_Not_used_0
Mode: RP3

linkId: 1
Rate: RATE_Not_used_0
Mode: RP3

linkId: 2
Rate: CPRI_9830_4Mbps_16x
Mode: CPRI
CPRI RX:
 Protocol Version: 2
 Ethernet: Enabled, pointer p: 40
 CRX Status: XSYNC2
 RX AxCs:
  All RX AxCs Disabled!
CPRI TX:
 Protocol Version: 2
 Ethernet: Enabled, pointer p: 40
 CTX Framing: CPRI TX Framing has started
```

FIGURE 31. Example output of the Register Dump Analyzer.

The user of the tool can enable AxC analysis by using the –axc command line parameter. Figure 32 shows an example output of the tool when using the following command "python3 reg_dump_analyzer.py --rx --axc /path/to/binaries". The log shows information about the first active AxC on the CRX side of the fourth link.

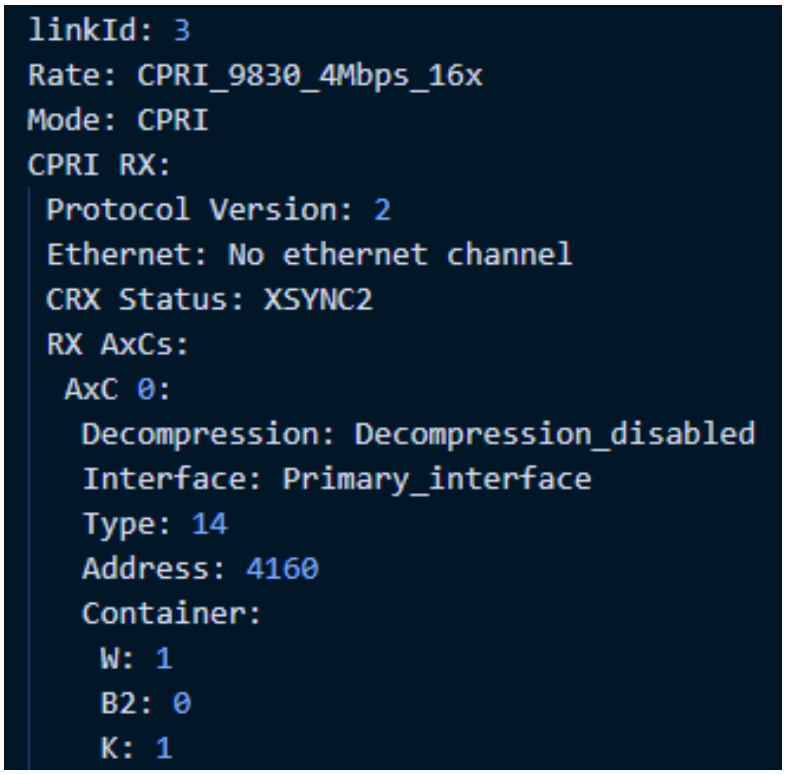

FIGURE 32. Example output of the tool when using the –axc parameter.

5.3 Testing

The Register Dump Analyzer is intended to be deployed to the development environment of our team at Nokia. This means that the tool should include some tests to help with further development and maintaining its functionality. A small "golden sample" test case was created, which uses a set of register dump binaries to run the tool and compare the output to an "golden sample" which is a pre-generated output using the same parameters and binaries as the test. The test is simple and fast to run but it has a downside that the sample needs to be updated if changes are made to the output format. However, this process should make the developer doing it more aware of their changes to the tool.

The golden sample test case that was created, is shown in Figure 33. The test case is ran using the pytest framework and start by setting up the python logging library. This is done because the test skips the entry point of the Register Dump Analyzer, and it just runs the actual main function with a set of arguments. When the Register Dump Analyzer has been run, the test case will run the compare_logs() function which compares the pre-generated golden sample with the newly created output. If all the content in the files match, the test passes. The test will fail if there is any difference between the two files.

```
def compare_logs(golden_log, test_output_log):
    with open(golden_log, 'rt', encoding='utf8') as golden, open(
        test_output_log,
        'rt',
        encoding='utf8',
    ) as test:
        assert golden.readlines()[2:] == test.readlines()[2:]


def test_reg_dump_analyzer():
    reg_dump_analyzer.logger = logging.getLogger()
    reg_dump_analyzer.logging.basicConfig(
        format='%(message)s',
        filename=OUTPUT_FILE,
        filemode='w',
        level=logging.DEBUG,
        force=True,
    )
    reg_dump_analyzer.main(TEST_ARGS)

    compare_logs(GOLDEN_LOG_FILE, OUTPUT_FILE)
```

FIGURE 33. The golden sample test case.

6 RESULTS

During this thesis, a tool was developed to analyze CPRI IP core register dumps. The tool was built in Python, and it utilized an existing binary decoder to decode the register dump binaries. The Register Dump Analyzer then read and parsed the output of the decoder and produced an analysis of the current state of the base station. Different alarms and register values are checked during the analysis and the tool informs the user with a warning if any concerning values are found. The analysis also includes general information about the current state of the CPRI links which can help the user to determine the state even further.

The tool was proven functional, and it effectively parsed and analyzed the register dump binaries. The Register Dump Analyzer currently stores all the register values from the binary decoder output. This scalable design allows for future enhancements and development, such as adding more registers to be checked during the analysis.

One of the primary challenges was the binary decoder's output, which was a non-standard text format that closely resembled JSON. A custom parsing algorithm was created for that purpose to allow effective conversion of the text data to the tool's memory for analysis. The development of this algorithm was essential for enabling the tool to handle the complex data format and perform efficient analysis.

The development of the tool also covered creating a test case and documentation for the tool. The test case that was created is a golden sample type test case that runs the Register Dump Analyzer using a set of register dump binaries. After the analyzer has been run, the test case validates the tool's output by comparing it against a pre-verified output, which is the golden sample. If the contents of the files do not match, the test case fails. This test case prevents a developer from unknowingly modifying the tools output and makes them more aware of their changes. If the output is modified, the golden sample must be updated manually by the developer.

7 CONCLUSIONS

The primary objective of this thesis was to create a tool for analyzing register dumps from the CPRI IP core, to help interpret the register dumps and the current state of the base station during a snapshot process. This goal was successfully achieved with the creation of the Register Dump Analyzer.

The integration with the existing binary decoder significantly accelerated the development of the Register Dump Analyzer by handling the initial parsing of binary files, thus eliminating the need for the Register Dump Analyzer to perform this task. However, the binary decoder's output in a non-standard format presented a notable challenge. To address this, a custom parsing functionality was developed. This feature not only overcame the initial challenge, but also enhanced the tool's up-gradability, allowing for easier adaptations and improvements in the future.

While the main objectives were met, there are areas for potential improvement. The test case, though effective, could be developed further to provide a more validation of the tool's capabilities. Also, the tool currently saves only the last values from the binary decoder output (either a string or an integer). This could be expanded to include the storing of hexadecimal values, offering a more comprehensive analysis. Another subject of improvement could be the execution time, while the tool currently takes one to two seconds to produce an output, it is possible that the parsing functionality could be optimized for performance, as it was not a main concern during this project.

This thesis provides valuable insights into base stations and CPRI technology, as the publicly available research and literature about these topics is limited or complicated. Additionally, within the L1 organization at Nokia, it serves as an introduction to the IP core register dumps and the Register Dump Analyzer, offering a practical reference for both new and existing team members.

During my thesis, I significantly expanded my understanding of CPRI and base station hardware. While my year-long traineeship at Nokia had already provided me with a solid grounding in the telecommunications and software engineering fields, it was this thesis project that allowed me to dive deep in the technical aspects and functionalities of the actual hardware. The knowledge and practical experience I have gained in CPRI and base station hardware through this thesis work have laid a solid foundation for my future career in software engineering.

In summary, this thesis has not only achieved its primary goal of developing a functional analysis tool for CPRI IP core register dumps but has also laid a great foundation for future enhancements of the tool. I believe that the Register Dump Analyzer will be taken into use by the CPRI team and will hopefully be a very useful tool when debugging the state of base stations. Throughout this process, I learned a lot, significantly enhancing my understanding and skills in this field of work.

REFERENCES

AMD. n.d. CPRI. Search date: 25.10.2023. https://www.xilinx.com/products/intellectual-property/do-di-cpri.html

Anritsu. 2016. Improving Wireless Network Flexibility Using CPRI Technology. Search date: 11.10.2023. https://dl.cdn-anritsu.com/en-us/test-measurement/files/Technical-Notes/White-Paper/11410-00910A.pdf

Cloudflare. n.d. What is the OSI Model? Search date: 2.10.2023. https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/

Costa, L. 1998. Open Systems Interconnect (OSI) Model. Search date: 2.10.2023. https://doi.org/10.1177/221106829800300108

Cox, C. 2020. An Introduction to 5G: The New Radio, 5G Network and Beyond. Search date: 30.9.2023. Available at internal library.

CPRI Info n.d. Common Public Radio Interface. Search date: 30.9.2023. http://www.cpri.info/

CPRI Specification V5.0 2011. Common Public Radio Interface. Search date: 30.9.2023. http://www.cpri.info/downloads/CPRI_v_5_0_2011-09-21.pdf

Intel. n.d. CPRI Intel® FPGA IP. Search date: 25.10.2023. https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/cpri-ip.html

Mukherjee, S. 2021. Packetizing Mobile Fronthaul with Cisco NCS540-FH. Search date: 5.10.2023. https://xrdocs.io/packet-fronthaul/blogs/PacketizingFH/

Butler, S. 2023. What Is Intellectual Property Core (IP Core)? Managing Semiconductor IP. Search date: 27.10.2023. https://www.perforce.com/blog/mdx/what-is-ip-core

RF Wireless World n.d. CPRI frame structure. Search date: 7.10.2023. https://www.rfwireless-world.com/Articles/CPRI-RRH-frame-structure.html

Awati, R. 2022. Intellectual property core definition. Search date: 27.10.2023. https://www.tech-target.com/whatis/definition/IP-core-intellectual-property-core