Design and User Guide for the Single Chip Mote Digital System

by Sahar M. Mesri

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Kristofer S. J. Pister
Research Advisor

Date

* * * * * *

Professor Ali M. Niknejad
Second Reader

Date

Design and User Guide for the Single Chip Mote Digital System

Sahar M. Mesri

29 April 2016

Abstract

In order to create a low-power and lightweight wireless sensor node for the control of MEMS microrobots, the Single Chip Mote project aspires to integrate a fully-functioning microprocessor, radio, sensors, and solar cells onto a single die, while also eliminating the need for external components through careful architectural design. This report presents the past two years of work on the design of the Single Chip Mote digital system, complete with an ARM Cortex-M0 microprocessor, control logic for an IEEE 802.15.4 radio, special-purpose radio timers, and ADC interface. This includes details on the design and contents of the Verilog code used to describe the hardware, and the software written to run and test the Single Chip Mote digital system. The required tools and testing procedures are also explained, along with the details required to convert this FPGA-based design to an ASIC design ready for tapeout. The intention behind this report is to pass on the knowledge acquired throughout the course of this project to those who are working to improve and iterate on this design. This report also presents preliminary power, area, and timing characteristics for the ASIC version Single Chip Mote digital system.

Acknowledgments

First and foremost, I would like to thank my adviser, Professor Kristofer S. J. Pister, for his overwhelming support and mentorship throughout my undergraduate and graduate studies at UC Berkeley. His door was always open when I needed academic, research, and career advice, and his vast breadth of knowledge never ceases to amaze (and occasionally intimidate) me. I am honored to be one of his students.

I would also like to thank the Single Chip Mote team, Dr. Osama Khan, David Burnett, Brad Wheeler, and Filip Maksimovic, for their friendship, guidance, and inspiration during the course of this project and my graduate studies. I could not ask to work with a better group of engineers, and I hope that we can continue to collaborate on projects in the future.

I would also like to express my gratitude towards all of the students, postdocs, and visiting scholars both past and present within Professor Pister's research group, for their friendship, encouragement, and support, as well as the students and staff in UC Berkeley's Ubiquitous Swarm Lab.

Dedication

To Mike Mesri, Zohreh Mesri, and Chris Case for their continued love, support, and encouragement throughout my undergraduate and graduate studies.

Contents

Chapter 1

Introduction

The term "Smart Dust" was originally coined by Professor Kris Pister to describe low-cost, low-maintenance, and unobtrusive wireless sensor nodes on a micro scale. These motes form interconnected mesh networks to communicate with one another and transmit sensor information. Sprinkling this dust over a field, on a road, or inside an office could provide valuable sensor data for a variety of applications... until the dust is swept away or thrown in the trash. On the other hand, mobile dust, in the form of autonomous microrobots, would be built with special MEMS devices that enable them to crawl or even fly. A swarm of these microrobots could space themselves out within a target area for optimal coverage, reposition themselves to areas with better wireless connectivity or illumination for their solar cells, or even mount a rescue for their brethren trapped within the depths of a Roomba.

While these ideas may appear to be more suited for science-fiction than reality, graduate students in Professor Pister's research group at UC Berkeley are actively designing MEMS structures for the purposes of microrobot movement and flight. However, before these robots can take their first tiny steps or fly off the surface of a lab bench, they will need a small but fully-functioning wireless sensor node for control and communication. The Single Chip Mote project aims to design an autonomous wireless sensor node with all external components integrated onto a single IC, without sacrificing the functionality needed for controlling swarms of microrobots.

Wireless Sensor Nodes and the Internet of Things

The recent rise in the popularity of the "Internet of Things" (IoT) has fueled the demand for consumer-quality wireless sensor node devices. While the availability and variety of these nodes continues to grow, the increasing popularity of IoT devices brings forth challenges in low-power communication and interoperability. The wireless standards used in laptops and cell phones such as WiFi and LTE are too power-hungry to be used on small wireless sensor nodes. Bluetooth Low Energy is appealing due to its compatibility with laptops and cell phones, at the cost of the associated licensing fees. IEEE Standard 802.15.4, entitled Low-Rate Wireless Personal Area Networks, is also a popular choice since it defines the PHY and MAC layers underlying many other protocols commonly found on commercial motes. This standard is designed specifically for short-range, low-power, and low-data-rate application, and does not require licensing.

The OpenWSN project [26] aims to create an open-source implementation of the

complete protocol stack for IoT wireless sensor nodes with 802.15.4 radios. OpenWSN is compatible with a variety of software and hardware platforms, allowing different motes to communicate with one another and form mesh networks. Many of the top contributors to this project are former students and visiting scholars from Professor Pister's research group, and Professor Pister himself considers OpenWSN the ideal software platform for controlling and communicating with swarms of autonomous microrobots.

When Small is Not Small Enough

A quick overview of commercially-available wireless sensor nodes shows that many of these general-purpose motes have 8, 16, or 32-bit microprocessors running at frequencies on the order of 1-100MHz, 802.15.4 compliant radios, and a variety of inputs and outputs for analog and digital sensors. With these specifications, the motes are certainly capable of running OpenWSN and other applications within a mesh network. Their power consumption tends to be on the order of 1mW-1W while awake, requiring a battery or USB power. While these motes are small, able to fit within the palm of a hand, the weight of the battery and PCB itself makes them too large and heavy to be used for microrobotic control and communication. Without the benefit of energy harvesting, these motes also need to have their batteries replaced every few days or perhaps weeks. Examples of these motes include the TelosB and OpenMote-CC2538, both with OpenWSN support [14].

Research projects involving low-power motes tend to focus more on cramming commercial hardware onto tiny PCBs than in new embedded architectures for low-power applications. The designs presented in [27], [28], and [7] are coin-sized, low-power, and unobtrusive motes optimized for infant observation, energy sensing, and transportation monitoring. And yet, all three are remarkably similar: a small 8-bit microprocessor, one or more PCBs stacked on top of one another, a coin-cell battery, and radio duty-cycling for energy savings. These motes perform the same main function as well: sample, transmit, sleep, and repeat. Given the simplicity of their microprocessors, these motes are not able to implement a complex protocol stack for mesh networks. [27] and [7] also require their own base stations to communicate with the motes, whereas motes using IEEE 802.15.4 radios can communicate with any other mote or base station with an 802.15.4 transceiver. While these designs succeed in lowering energy consumption, they still require batteries that may only last for a few weeks, and the combination of the PCB and battery is still too heavy for a microrobot.

The authors of [18] claim to have developed the world's smallest wireless sensor node by designing a custom IC for their signal processing and data transmission. The custom IC die is directly bonded to a MEMS die containing all of the required sensors. This mote still requires an external antenna, which is fabricated using a thin flexible substrate instead of a PCB. The mote uses solar cells in combination with a rechargeable battery for longer battery life; it can also run without a battery as long as there is sufficient illumination. This mote is small, lightweight, and has minimal external components. However, the major downside of the design in [18] is the lack of a general-purpose microprocessor, as it is designed for the sole purpose of sampling and transmitting data.

Perhaps the best attempt thus far towards full integration is the Michigan Micro Mote [20], a series of thin layers stacked like LEGOs in order to form a complete

wireless sensor node. With as many as eight different layers containing the microprocessor, radio, sensors, and other components, the mote measures at just $2\times4\times4mm^3$, and has an incredibly low standby current of $2nA$. The mote can be powered completely via ambient light through its solar cells, and contains a battery layer to store any excess harvested energy. The only potential downside to this design is the increased complexity when manufacturing, aligning, and bonding eight different dies. This problem would only get worse when integrating microrobots into the design, as this would require a ninth layer for the robot body.

Finally, the 24/60GHz passive radio designed here at Berkeley [31] proves that a low-power radio relying entirely on energy harvesting is possible on a single die. Unfortunately, the chip relies on energy scavenging from a high power RF source, and this asymmetric communication link means that the chip behaves more like an RFID tag instead of an autonomous computer. As a result, two of these radios cannot directly communicate with one another, and are not well-suited for forming a network of microrobots. Also, both the 24GHz receiver and 60GHz transmitter are not compliant with any current IoT standards.

Single Chip Mote to the Rescue

The Single Chip Mote project intends to combine the generality and processing capability of these not-so-small commercial sensor nodes with the lightweight form factor and low-power techniques used in the tiny, specialized wireless sensor nodes. The high-level block diagram in Figure 1.1 shows the various subsystems that must be integrated onto a single die in order for this project to succeed. While this project is still a work in progress, the final version of the Single Chip Mote will contain a fully-functional 32-bit ARM Cortex-M0 microprocessor, a low-power 2.4GHz IEEE 802.15.4 compatible radio, energy-harvesting solar cells, and an on-chip oscillator to create an autonomous wireless sensor node on a single CMOS die (or possibly a CMOS die bonded to a MEMS die). Without the need for any external components, a battery, or a PCB, the Single Chip Mote is the ideal microcontroller for the future swarms of autonomous microrobots, each with a lightweight yet fully-capable brain for performing actions beyond the simple observe and report. The addition of the OpenWSN protocol stack allows for these robots to create an extensive and adaptive mesh network, and communicate with a variety of IoT hardware platforms and sensors supporting OpenWSN.

Single Chip Mote Digital System

The work presented in this report lays out the foundation for the digital components of the Single Chip Mote. A tested and functioning FPGA prototype of the Single Chip Mote digital system complete with an ARM Cortex-M0 microprocessor, radio controller, custom radio timer, and ADC interface is presented, along with the tools and procedures for designing hardware, writing software, and verifying functionality. A high-level block diagram of the Single Chip Mote digital system is shown in Figure 1.2. This is far from the final iteration of the Single Chip Mote digital system; the design lacks support for integrated sensors, periodic sensing without intervention from the microprocessor, power management and low-power modes, and other potential hardware accelerators to handle repetitive and energy-consuming tasks normally executed in software. As an example of hardware acceleration, OpenWSN

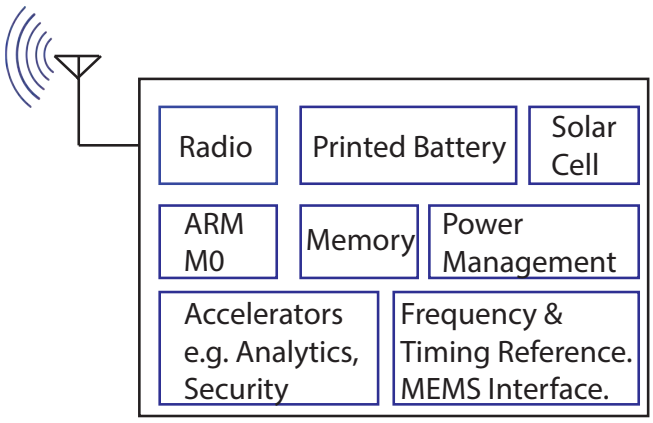


Figure 1.1: High-level block diagram of the Single Chip Mote and its subsystems

uses hardware timers to wake up the microprocessor for each step involved in sending a packet over the radio (such as copying packet data, turning on the radio, telling the radio to send, listening for a response). The current Single Chip Mote digital system has custom timers designed to automatically trigger the actions required to send a packet without waking up the microprocessor, and the overall process requires less energy than a traditional microcontroller. The continuation and success of this project depends on the collaboration between digital designers and embedded systems developers in identifying potential processes that can be more efficiently carried out in hardware.

Preliminary Results

Preliminary results already show the potential for improvement when using the Single Chip Mote in place of existing wireless sensor nodes. Using the Verilog code described in this report, the Single Chip Mote digital system was synthesized, placed, and routed using Synopsys Design Compiler and Synopsys IC Compiler to obtain estimates for the area, power, and timing characteristics. Figure 1.3 shows an image of the complete design. The technology used for this design is TSMC $65nm$ LP, with a clock frequency of $5MHz$ and an operating voltage of $1.2V$. The synthesis scripts are constrained to use only high threshold voltage (HVT) standard cells to reduce leakage current. While the digital system requires a additional debug hardware and improved floorplanning before tapeout, these initial results provide a more accurate estimate for evaluating this design relative to the goals of the project.

The target clock period of $200ns$ ($5MHz$ clock frequency) is relatively slow in a $65nm$ process, as evidenced by the ample critical path slack of $182ns$. While this design could easily run at faster clock frequencies, $5MHz$ was chosen in order to reduce dynamic power in the digital domain, as well as the power required by the on-chip oscillator for clock generation.

The dynamic power for this design is $139\mu W$, and the leakage power is $17.8\mu W$. These values are calculated using an operating voltage of $1.2V$, and can be further improved by scaling down the operating voltage. The LP (low-power) process was chosen since its high-threshold transistors significantly reduce leakage current. The main cost is that operating voltages are typically higher than in GP (general-purpose) processes in order to have the same on-current and run at similar speeds.

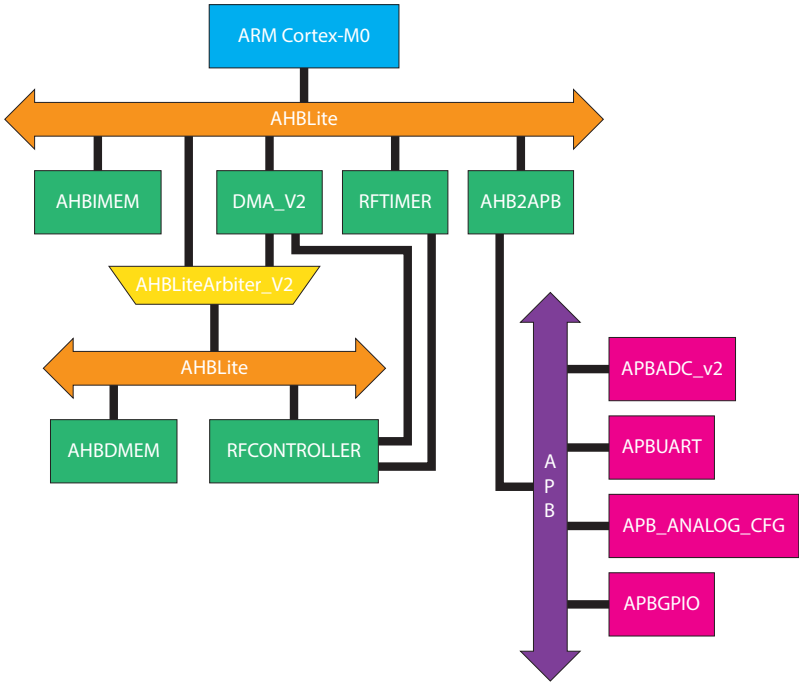


Figure 1.2: High-level block diagram of the Single Chip Mote digital system

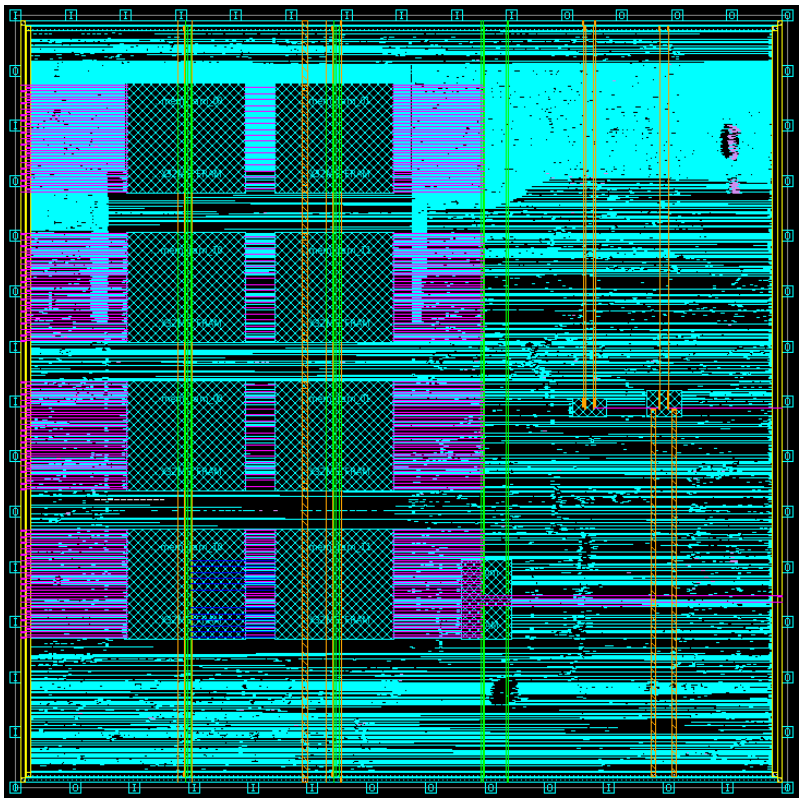


Figure 1.3: Preliminary layout for the Single Chip Mote digital system

However, since the Single Chip Mote digital system has a relatively slow clock frequency, it is acceptable to operate these transistors at a lower operating voltage and still meet timing requirements. Since dynamic power scales with V_{dd}^2, reducing the operating voltage to $0.9V$ reduces the dynamic power to $78.3\mu W$. Since leakage power scales with V_{dd}, reducing the operating voltage to $0.9V$ reduces the leakage power to $13.4\mu W$. Therefore, at $0.9V$, the Single Chip Mote digital system has a power budget of $91.7\mu W$. Reducing voltage requires that the standard cells are re-characterized at $0.9V$, since this information is not provided by TSMC.

The initial Single Chip Mote design does not have any power management hardware, clock gating, or power gating. Therefore, this design does not have any low power modes like the commercial microprocessors. However, the leakage power represents the power consumed when the Cortex-M0 is idle or asleep, and the sum of dynamic and leakage power indicates the power consumed while the Cortex-M0 is active. These results can be compared to the active and idle power consumption of commercial motes. The future addition of power gating to the Single Chip Mote will allow for more direct comparison to the low power modes of commercial microcontrollers. Also note that the power consumption reported does not include the radio or any additional hardware outside of the digital system.

One of the more popular microcontroller boards used for OpenWSN is the TelosB. This board contains the TI MSP430 microcontroller, which has an input voltage range of $1.8V$ to $3.6V$, and an active current of $330\mu A$ at $1MHz$ and $2.2V$, according to the datasheet [21]. These measurements indicate an active power of $726\mu W$. The datasheet does not specifically state the current draw while the CPU is idle, but it does have the current draw for four of the five low power modes: $50\mu A$, $11\mu A$, $1.1\mu A$, and $0.2\mu A$. These low power modes use clock gating, power gating, and powering down oscillators to reduce current draw. At $2.2V$, the resulting power for each of the measured low power modes is $110\mu W$, $24.2\mu W$, $2.42\mu W$, and $0.44\mu W$. The active power of the Single Chip Mote digital system is much smaller than that of the MSP430, while still running at a higher frequency. Although the MSP430 low power modes cannot be directly compared to the idle power of the Single Chip Mote, it is clear that the Single Chip Mote, without any clock or power gating, has lower power while idle than the MSP430 does in its first low power mode.

Another board developed specifically for OpenWSN development is the OpenMote-CC2538, which uses the TI CC2438 microcontroller. According to the datasheet [6], the CC2538 has an input voltage range of $2V$ to $3.6V$, and its current draw varies based on which peripherals are active. The current draw while the CPU is running and clocked with an RC oscillator (and the radio, crystals, and peripherals are turned off), is $7mA$. The datasheet does not list the current draw while idle, but it does have three low power modes implemented using clock gating, power gating, and powering down oscillators. These three low power modes have a current draw of $0.6mA$, $1.3\mu A$, and $0.4\mu A$. Assuming an input voltage of $2V$ is used, the power while active is $14mW$, and the power during the three low power modes is $1.2mW$, $2.6\mu W$, and $0.8\mu W$. The active power of the Single Chip Mote digital system is much smaller than that of the CC2538. Although the CC2538 low power modes cannot be directly compared to the idle power of the Single Chip Mote, it is clear that the Single Chip Mote, without any clock or power gating, has lower power while idle than the CC2538 does in its first low power mode.

While these numbers are optimistic, the main reason for the improvement is

most likely due the use of the $65nm$ LP processes. The Single Chip Mote digital system also has significantly fewer on-chip peripherals than the MSP430 or CC2538, reducing both dynamic and leakage power. The CC2538 also uses an ARM Cortex-M3 processor, which requires more power when compared to the ARM Cortex-M0.

Area is also an important consideration, since this chip must be light enough to be carried by a MEMS microrobot. The preliminary design for the Single Chip Mote digital system has a total cell area of $856600\mu m^2$, which easily fits within a die area of $1mm^2$. Assuming an incident power of $1mW$ per mm^2 in direct sunlight, CMOS solar cells with at least 10% conversion efficiency should be able to provide $100\mu W$ per mm^2 of die area in direct sunlight. Therefore, this design (when run with an operating voltage $0.9V$) requires approximately $1mm^2$ of solar cells to power the Single Chip Mote digital system. It is estimated that the analog, radio, and voltage converters for the Single Chip Mote will require $2mm^2$ of area for the circuits themselves, and $2mm^2$ of area for additional solar cells. With these numbers in mind, the Single Chip Mote requires a total die area of $6mm^2$. Given that the thickness of the die is about $200\mu m$, and the density is similar to that of crystalline silicon ($2.33g/cm^3$), the estimated mass of the die is $2.8mg$.

Researchers in our group are currently designing MEMS motors and legs for walking microrobots. Each leg outputs a downward force of $300\mu N$, and can move a mass of $30mg$. The mass of the legs themselves are $15mg$ each, which allows for $15mg$ of payload per leg. With these values in mind, a one-legged MEMS microrobot generates enough downward force to support the weight of the Single Chip Mote.

Future Work

While the RTL design of the preliminary Single Chip Mote digital system is complete, there is still more work required before the Single Chip Mote is ready for widespread use. The Single Chip Mote team submitted a tapeout in March 2016 containing the first version of the analog and radio circuits, which will be fabricated and tested in May 2016. Building off of the results of the first tapeout, the team is targeting a second tapeout in August 2016, which will include the first version of the digital system (as described in this report) and the second version of the analog and radio circuits. The results of this second tapeout, as well as feedback from software developers from the OpenWSN project, will determine the direction of this project in 2017 and beyond.

Report Outline

The rest of this report covers the details of the Single Chip Mote digital system design, as well as the development tools and testing procedures. Chapter 2 provides an overview of the tools for hardware development for the FPGA and software development for the Cortex-M0 microprocessor, including the basics of their installation and use. Chapter 3 contains a detailed explanation of the Single Chip Mote digital system hardware, and chapter 4 demonstrates how to write software that uses the hardware peripherals. Chapter 5 covers the details on loading software onto an FPGA or ASIC containing the Single Chip Mote digital system. Chapter 6 describes the current testing procedures, including simulation and real-time verification. Chapter 7 details the changes required to convert the Single Chip Mote digital system from an FPGA design to an ASIC design. Chapter 8 concludes this report

with a discussion the accomplishments of this project and areas for improvement. The bibliography beginning on page 207 contains a list of the references included in this report.

Chapter 2

Getting Started

This chapter is a basic introduction into the hardware and software tools used for the development of the Single Chip Mote digital system. While the details on the design of this system are presented in later chapters, the purpose of this chapter is simply to introduce these tools and provide basic instructions in their installation and use. This guide is by no means comprehensive and further experimentation and study is required in order to truly understand and master the use of these tools.

2.1 Git Repository

All of the source code and project files for this design is found on the following git repository:
`https://repo.eecs.berkeley.edu/git/projects/pistergroup/scm-digital.git`

The repository contains two main directories, one for source code and one for project files specific to the hardware and software development environments. The source code section further divides into hardware and software code, and each of those sections divide further based on FPGA target or software project. The project files directory is also divided into sections for hardware and software IDEs, and each of those sections also divide further based on FPGA target or software project. This hierarchy is shown in Figure 2.1.

There is also a deprecated repository containing the history of the project during its early development. This repository has been maintained solely for historical purposes:
`https://repo.eecs.berkeley.edu/git/projects/pistergroup/singlechip-digital.git`

The information in this user guide is meant for those using the latest source code found on the current repository and may not be applicable to the code found on the deprecated repository.

2.2 ARM Cortex-M0 DesignStart Processor

The digital microprocessor used for this system is the ARM Cortex-M0 DesignStart processor [11] [5], provided at no cost by ARM for educational purposes. The processor is fully software-compatible with the commercial Cortex-M0; however, the provided Verilog is obfuscated and does not support JTAG debugging.

```
/scm-digital
  /proj
    /ise
      /artix7
        /SingleChipMote           # ISE project files for main digital
                                  # system implemented on Artix-7
        /testbench                # ISE project files for testbenches
      /spartan6
        /bootloader               # ISE project files for the bootloading
                                  # hardware implemented on Spartan-6
        /testbenches              # ISE project files for testbenches
        /uRobotDigitalController  # ISE project files for main digital
                                  # system imlpemented on Spartan-6
    /keil
      /firmware                   # Keil project files for bootloading
                                  # software
      /uRobotDigitalController    # Keil project files for main digital
                                  # system software
  /src
    /hw
      /artix7
        /uRobotDigitalController  # Verilog files for main digital system
                                  # implemented on Artix-7
      /spartan6
        /bootloader               # Verilog files for bootloading hardware
                                  # implemented on Spartan-6
        /uRobotDigitalController  # Verilog files for main digital system
                                  # implemented on Spartan-6
    /sw
      /firmware                   # C source and assembly source files
                                  # for bootloading software
      /uRobotDigitalController    # C source and assembly source files
                                  # for main digital system software
```

Figure 2.1: Git repository directory structure

2.3 FPGA Boards

2.3.1 Digilent Nexys 3

The Digilent Nexys 3 [23] is a digital circuit development platform containing the Xilinx Spartan-6 XC6LX16-CS324 FPGA along with various peripherals. The digital system of the Single Chip Mote was originally designed and developed on the Nexys 3 FPGA. This board was chosen because it was the recommended prototyping board to be used with the ARM Cortex-M0 DesignStart processor package. The DesignStart package also came with example projects and Verilog code specifically designed for the Nexys 3 board. However, as the size of the digital system grew, the FPGA on the Nexys 3 could not support the amount of RAM and logic devices needed, and thus the project was moved to the Nexys 4 DDR board. The Nexys 3 is still used for bootloading purposes (see chapter 5 for more information). This board and FPGA is compatible with Xilinx's ISE Design Suite and can also be programmed separately through Digilent's Adept software.

2.3.2 Digilent Nexys 4 DDR

The Digilent Nexys 4 DDR [24] is a digital circuit development platform containing the Xilinx Artix-7 XC7A100T-1CSG324C FPGA along with various peripherals. This board is currently used for design, development, and testing of the Single Chip Mote digital system. This board and FPGA is compatible with Xilinx's ISE Design Suite and Vivado Design Suite.

2.4 Hardware Development Tools

2.4.1 Xilinx ISE Design Suite 14.6

Overview

Xilinx ISE 14.6 is the integrated development environment used for the hardware development of the Single Chip Mote on both the Artix-7 and Spartan-6 FPGAs. This version of ISE can be download directly from Xilinx, and is known to work on Windows 7 computers. It can also be installed in Windows 8 and 10 with a few extra steps in the installation procedure. This version can also be installed on Linux. Xilinx ISE 14.7 should also work for the purposes of this project; however, this has not been tested or confirmed.

ISE contains multiple tools required for this project:

Project Navigator The main IDE used to write and synthesize RTL for the FPGA.

ChipScope Pro A tool used to create and embed a logic analyzer into an FPGA design for debugging.

CORE Generator A tool used to generate Xilinx IP for FPGA designs such as memories, FIFOs, and clock generators.

ISim A Verilog simulator.

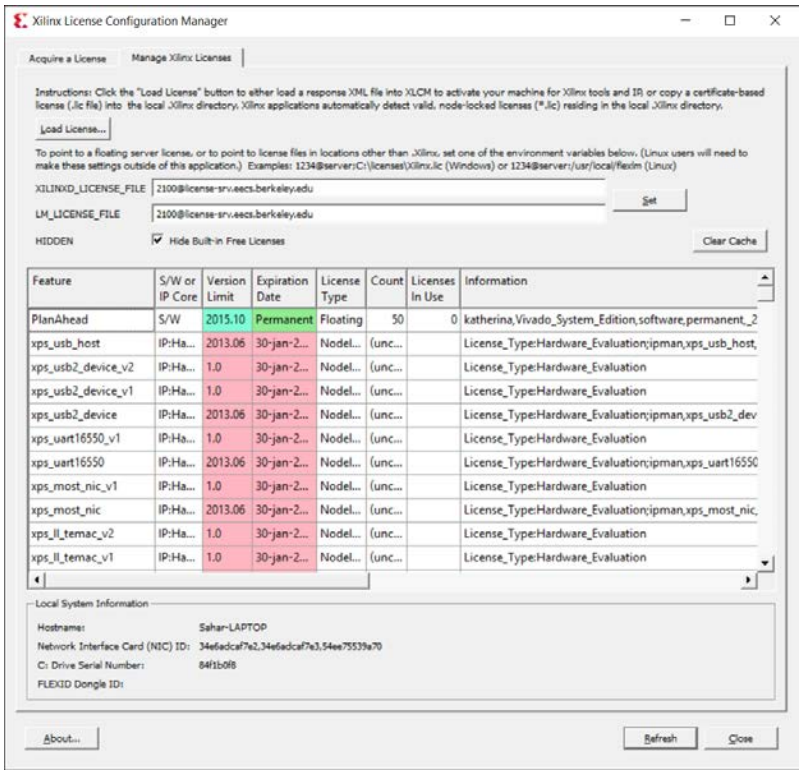


Figure 2.2: Xilinx license manager settings for UC Berkeley license

iMPACT A took used to load configuration bitstreams onto FPGAs.

Xilinx has chosen to deprecate its ISE Design Suite in favor of its Vivado Design Suite. However, Vivado supports only 7-series FPGAs, and is not compatible with the Spartan-6 FPGA. Therefore, this project continues to use ISE.

Installation

First, download ISE 14.6 from Xilinx [22]. Then, install using the default settings. Once finished the installer may open the license manager and ask for a license. Students and researchers at UC Berkeley may use the license provided to the EECS department for instructional purposes by setting the XILINXD_LICENSE_FILE and LM_LICENSE_FILE values to 2100@license-srv.eecs.berkeley.edu, as shown in Figure 2.2. This license supports all versions of Xilinx tools released prior to October 2015. Xilinx also provides free WEBPACK licenses.

Additional Install Directions for Windows 8/8.1/10

Xilinx has also chosen to stop updating and supporting ISE installations, including compatibility updates for Windows 8/8.1/10. ISE 14.6 can still be installed on Windows 8/8.1/10 computers; however, it requires the following modifications [13] in order to work:

1. Open the following directory: `C:\Xilinx\14.6\ISE_DS\ISE\lib\nt64`
2. Find and rename `libPortability.dll` to `libPortability.dll.orig`

3. Make a copy of `libPortabilityNOSH.dll` (copy and paste it to the same directory) and rename it `libPortability.dll`

4. Copy `libPortabilityNOSH.dll` again, but this time navigate to `C:\Xilinx\14.6\ISE_DS\common\lib\nt64` and paste it there

5. In `C:\Xilinx\14.7\ISE_DS\common\lib\nt64` Find and rename `libPortability.dll` to `libPortability.dll.orig`

6. Rename `libPortabilityNOSH.dll` to `libPortability.dll`

Synthesizing a Design and Loading a Bitstream

The following instructions demonstrate how to synthesize a design in Project Navigator and load the resulting bitstream onto an Artix-7 FPGA (on the Digilent Nexys 4 DDR board) using iMPACT.

Generating a bitstream file

1. Open Project Navigator and open a project. The project file used for this demonstration is `scm-digital/proj/ise/artix7/SingleChipMote/SingleChipMote.xise`.

2. Select the top module for this project, in this case uCONTROLLER, in the Design Hierarchy panel. From there a list of processes that can be run on this module will appear in the Processes panel (see Figure 2.3).

3. Run the Synthesis, Translate, Map, Place & Route, and Generate Programming File processes in that order. This series of processes generate a bitstream file, ucontroller.bit, used to program a compatible FPGA (in this case the Artix-7 XC7A100T).

Connecting the Nexys 4 board to load a bitstream file

1. Ensure that the jumper JP1 on the Nexys 4 board is in the JTAG position and that the power switch is in the ON position.

2. Connect the Nexys 4 board to the computer using the micro-USB port on the board labeled PROG UART. This USB port is used for both programming and UART communication.

3. Once the board has been recognized by the computer and the proper drivers have been installed, open iMPACT by running the Configure Target Device process (see Figure 2.3).

There may be a warning saying that no iMPACT project file exists, or that a target device has not been designated. The next few steps create a new project and then configure iMPACT to write to the target FPGA.

Using iMPACT to load a bitstream file

1. Go to the File menu and select New Project. Select Yes when asked to have a project file automatically created.

2. A new window will open with various options on how to program the FPGA (see Figure 2.4). Choose Configure devices using Boundary-Scan (JTAG), and underneath that choose Enter a Boundary-Scan chain manually. The manual approach is better in the case where there is more than one FPGA board connected to the computer at once; it provides the user with the option to connect to a specific board instead of allowing iMPACT choose the first one it sees.

3. Select the Cable Setup... option in the Output menu to open the Cable Communication Setup window. From here select Digilent USB JTAG Cable in the Communication Mode section (this is necessary for both Nexys boards). Then go to the drop-down menu under Port to see all devices connected to the computer (see Figure 2.5). Choose the appropriate board, based on the serial number written on the board and shown in the drop-down menu, and select Ok.

4. Inside the main window in iMPACT, right click and select Add Xilinx Device... (see Figure 2.6). From here choose the bitstream file to be loaded onto the FPGA (in this case ucontroller.bit). The device will now appear in the main window in iMPACT (see Figure 2.7).

5. Select the device by clicking on the image of the Xilinx chip (the chip will change colors from grey to green). Then double-click Program process from the iMPACT Processes list on the left side of the main window to load the bitstream onto the board.

Once this process has completed, the main window will display Program Succeeded (see Figure 2.8). The FPGA has now been configured to run the hardware synthesized in Project Navigator.

For more information on how to create a project in ISE for the Artix-7, see section 3.1 on ISE project settings.

2.4.2 Digilent Adept

Digilent Adept is a utility provided by Digilent that can be used to load bitstream files onto some of their FPGA boards. The version of Digilent Adepet used for this project is 2.15.3; however, the latest version [12], 2.16.1, is also compatible. Adept is used to program the Nexys 3 board but does not support the Nexys 4 DDR. The Nexys 3 also has additional memory external to the FPGA, written through Adept and accessed on the FPGA. This memory is used for bootloading (see chapter 5 for more information on bootloading), and must be written *before* the bitstream is loaded.

Using Adept to load a bitstream file (Nexys 3 only)

1. Ensure that the power switch on the Nexys 3 board is in the ON position.

2. Connect the Nexys 3 board to the computer using the micro-USB port on the board labled USB PROG. There is a separate micro-USB port on the board for UART communication which cannot be used for loading a bitstream.

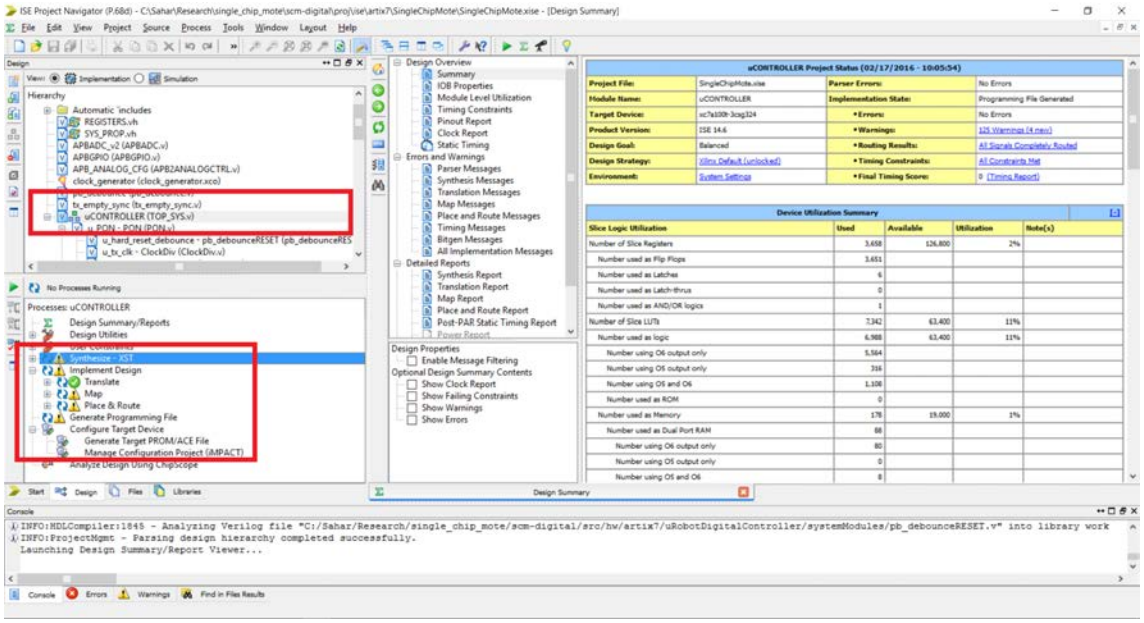


Figure 2.3: ISE Project Navigator for the SingleChipMote project

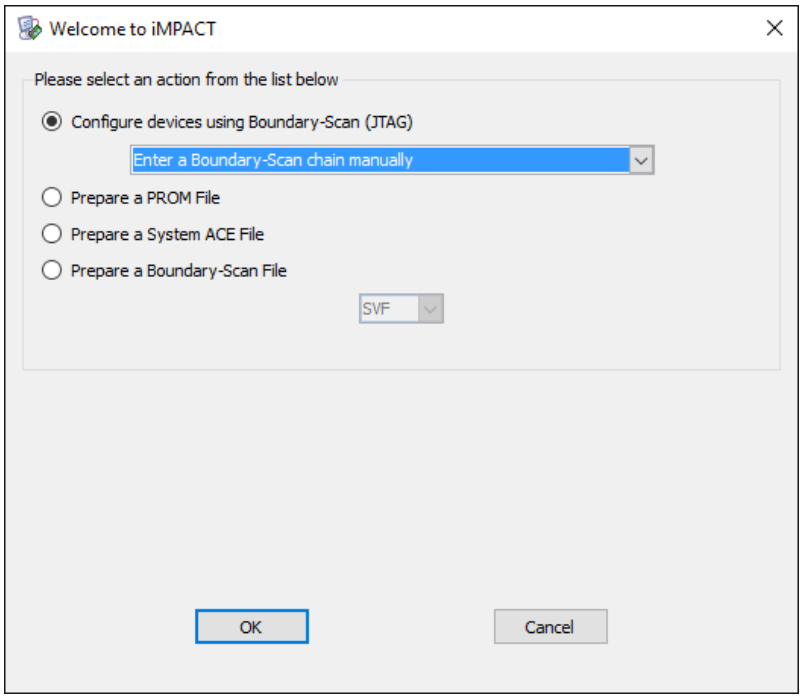


Figure 2.4: Configuring a new iMPACT Project

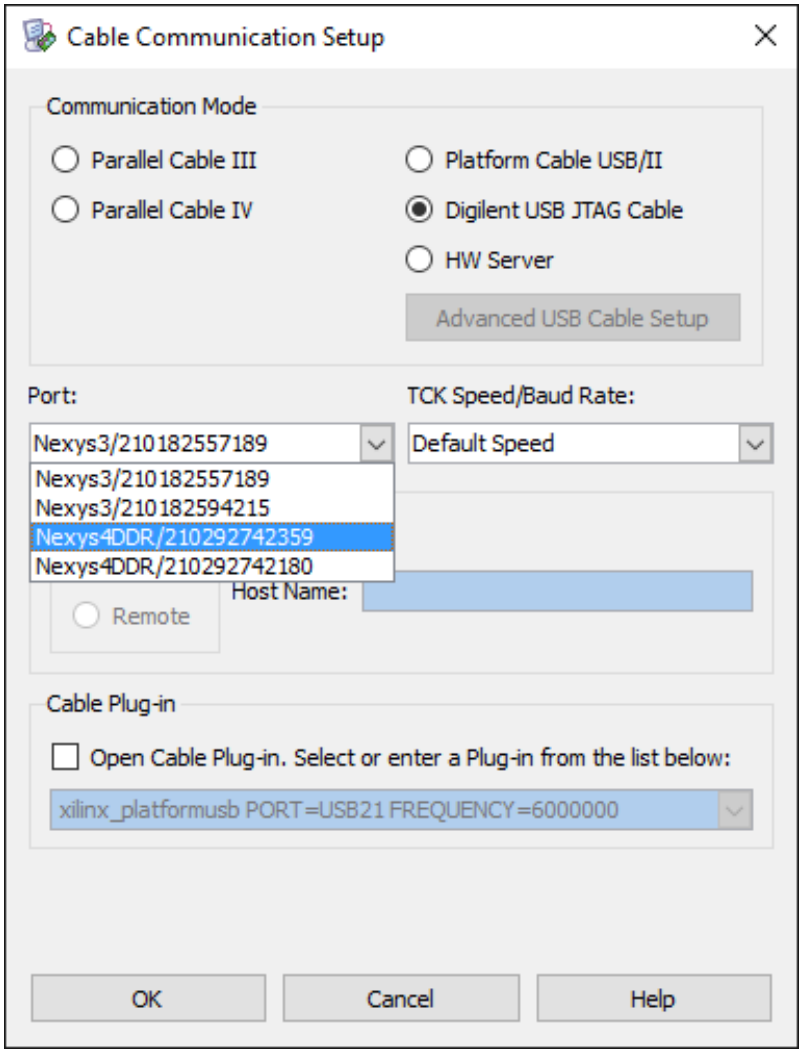


Figure 2.5: Selecting the device to program in iMPACT

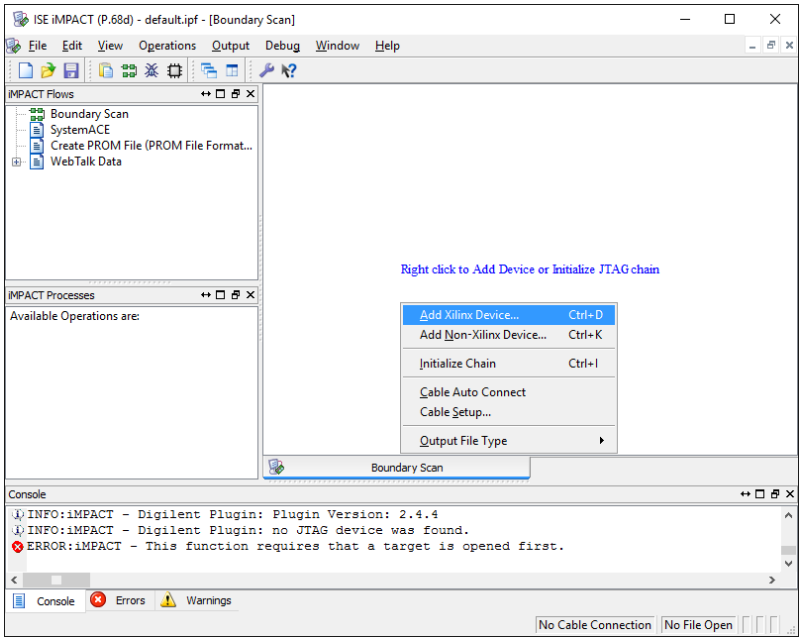


Figure 2.6: Adding a Xilinx device in iMPACT

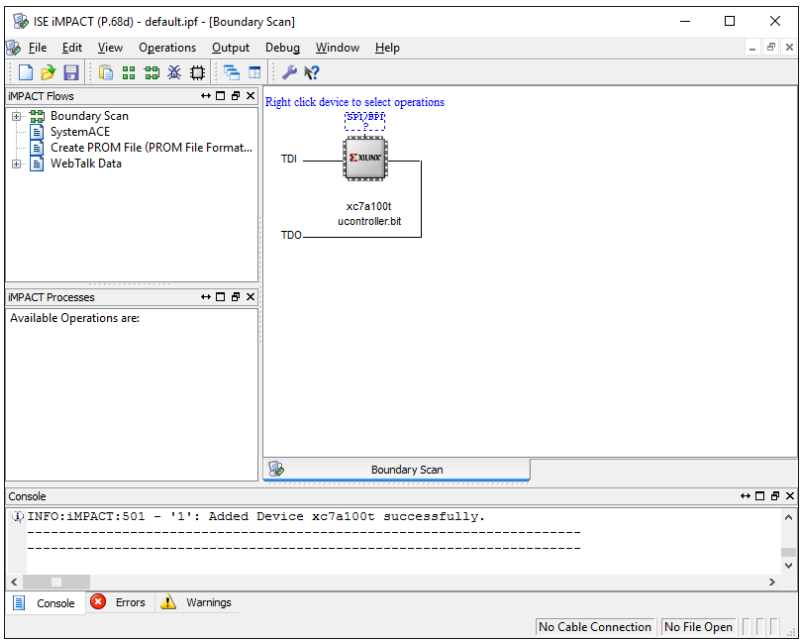


Figure 2.7: Main iMPACT window with a selected device to program

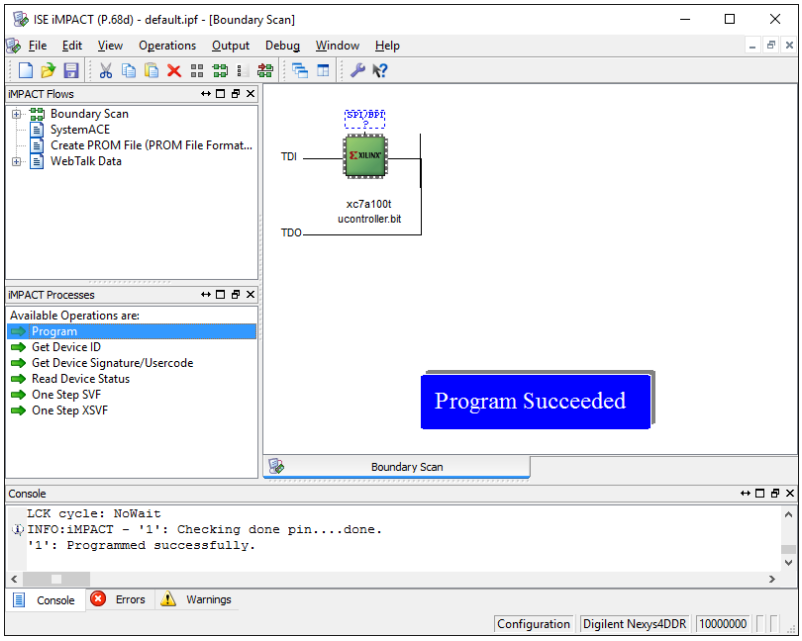


Figure 2.8: Main iMPACT window after successfully programming a device

3. Once the board has been recognized by the computer and the proper drivers has been installed, launch Adept.

4. On the upper-right of the window, there is a drop-down menu listing all of the Digilent devices connected to the computer. Select the Nexys3.

5. Go to the Config tab, select the Browse... button, and select the bitstream file generated for a Spartan-6 FPGA (for an example project on the Spartan-6, go to `scm-digital/proj/ise/spartan6/uRobotDigitalController/ucontroller.bit`).

6. Press the Program button to program the FPGA (see Figure 2.9).

Using Adept to load a file into external RAM

NOTE: The memory must be written *before* the bitstream is loaded.

1. Ensure that the power switch on the Nexys 3 board is in the ON position.

2. Connect the Nexys 3 board to the computer using the micro-USB port on the board labled USB PROG. There is a separate micro-USB port on the board for UART communication which cannot be used for loading a bitstream.

3. Once the board has been recognized by the computer and the proper drivers has been installed, launch Adept.

4. On the upper-right of the window, there is a drop-down menu listing all of the Digilent devices connected to the computer. Select the Nexys3.

5. Go to the Memory tab. There are options to program SPI Flash, BPI Flash, and RAM. The Nexys 3 designs used for the Single Chip Mote use the RAM.

6. Select the RAM option on the right side of the Memory tab. Then select the Browse... button in the Write File to Memory section and select the file to be written. Check the Verify check box.

7. Select the Write button (see Figure 2.10).

For more information on how Digilent Adept is used for bootloading onto the Single Chip Mote, see section 5.7 on connecting and loading software.

2.4.3 Xilinx Vivado Design Suite

As mentioned previously, the Vivado Design Suite can be used for designs on the Artix-7 FPGA. Xilinx and Digilent are providing increasing support for Vivado and decreasing support for ISE. However, importing the Single Chip Mote project from ISE to Vivado has not been tested, nor has it been confirmed that the UC Berkeley Xilinx license will work on earlier versions of Vivado (it will not work on anything released after October 2015). Xilinx does offer a free WEBPACK license which also has not been used or tested for the Single Chip Mote project.

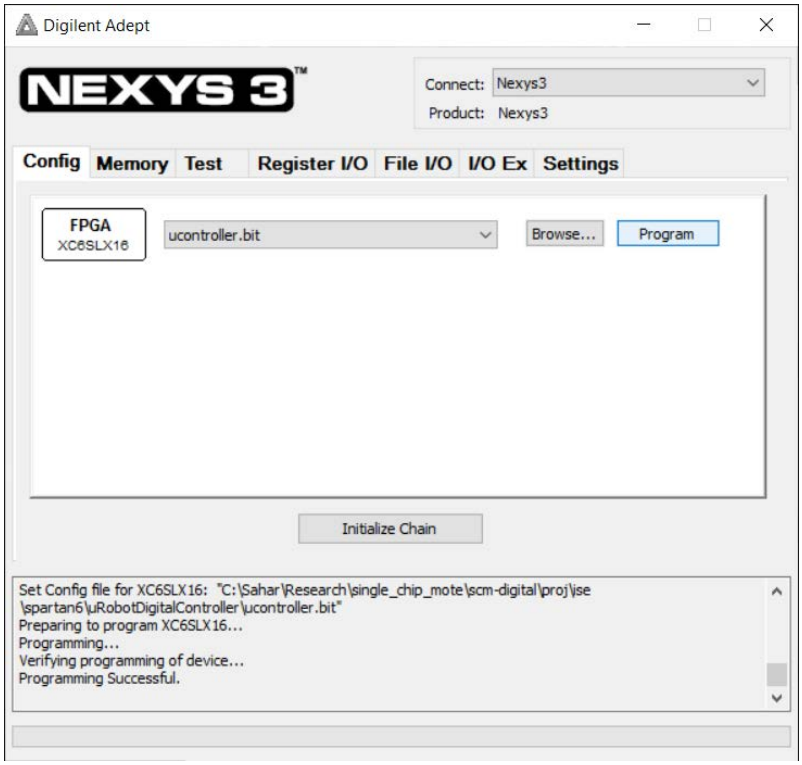


Figure 2.9: Adept settings for programming a bitstream

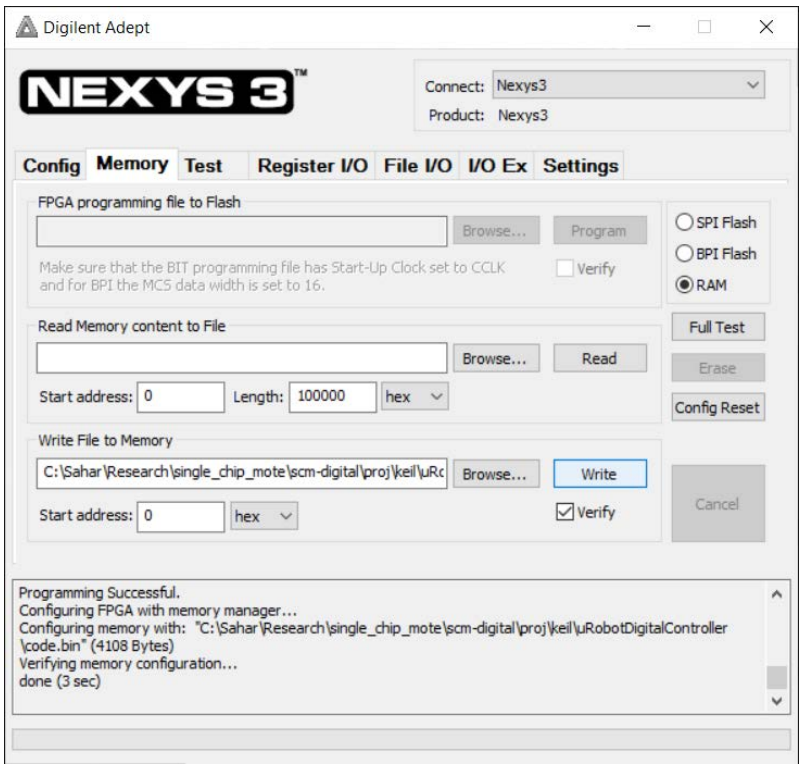


Figure 2.10: Adept settings for writing memory on the Nexys 3 board

2.5 Software Development Tools

2.5.1 Keil uVision5

Keil uVision5 is an IDE used for developing software running on ARM microprocessors. This IDE is part of the ARM MDK 5 Microcontroller Development Kit [19]. The version of the MDK used for this project is 5.11; however, the latest version, 5.18, is also compatible.

To compile code for the ARM Cortex M0 on the Single Chip Mote, open an existing project in Keil uVision5. The project file used for this demonstration is `scm-digital/proj/keil/uRobotDigitalController/code.uvprojx`. Then go to the Project menu and select Build Target. This compiles the code, creates a C binary image called code.bin, and also creates a text file called disasm.txt, containing a disassembled version of the code. The C binary image is loaded into the instruction memory of the Single Chip Mote on an FPGA using the bootloader (see chapter 5 for more information on bootloading).

For more information on how to make a Keil uVision5 project for the Single Chip Mote, see section 4.1 on Keil project settings.

2.5.2 Bin2coe

Bin2coe [4] is a small Windows executable used to convert C binary image files (bin) to COE files used by Xilinx to initialize FPGA memories. COE files are used to initialize the instruction ROM with software written and compiled in Keil. This is used for the bootloading ROM on the Single Chip Mote. This program limits the data widths of the COE file to 32 bits, and therefore the generated COE files are limited to memories with a width of 32 bits.

Chapter 3

Single Chip Mote Hardware

This chapter provides a detailed overview of all the hardware components of the Single Chip Mote digital system. The Single Chip Mote hardware encompasses all of the Verilog files, Verilog header files, and ISE project files used to describe the Single Chip Mote digital system. The intention of this chapter is to provide clarification and guidance to those planning on reading or modifying the hardware, and its is highly recommended that this chapter be read alongside the Verilog code described in each section. This chapter may also provide some useful insight for software developers designing applications for the Single Chip Mote, in particular the sections on register interfaces.

Some of the files and designs described in this chapter are provided by ARM with the Cortex-M0 DesignStart kit (see section 2.2 for more information), such as the Verilog for the Cortex-M0 and AHB controllers for various peripherals on the Nexys 3 board. There are also Verilog modules designed by Francesco Bigazzi, a visiting scholar who originally worked on the Single Chip Mote digital system, such as the bridge between AHB and APB, and some of the APB peripherals. All other work described in this section not attributed to ARM, Bigazzi, or any other designer is original.

3.1 ISE Project Settings

ISE projects have already been created for the Single Chip Mote digital system on the Artix-7 and Spartan-6, as well as the bootload hardware (chapter 5) for the Spartan-6. However, it may be necessary in the future to make more ISE projects for additional FPGA designs, such as running the bootload hardware (chapter 5) on the Artix-7. This section contains the information needed to create a new project for the versions of these chips running on the Nexys 4 DDR and Nexys 3 boards.

3.1.1 Artix-7

In order to create a new ISE project for the Artix-7 FPGA on the Nexys 4 DDR board, open ISE and choose New Project in the File menu. In the New Project Wizard window, enter a name for the project and specify the location of the project files. It is suggested that all project files related to Single Chip Mote be checked into the repo in `scm-digital/proj/ise/artix7/`. After selecting Next, the New Project Wizard will display the options for the device and design flow for the project.

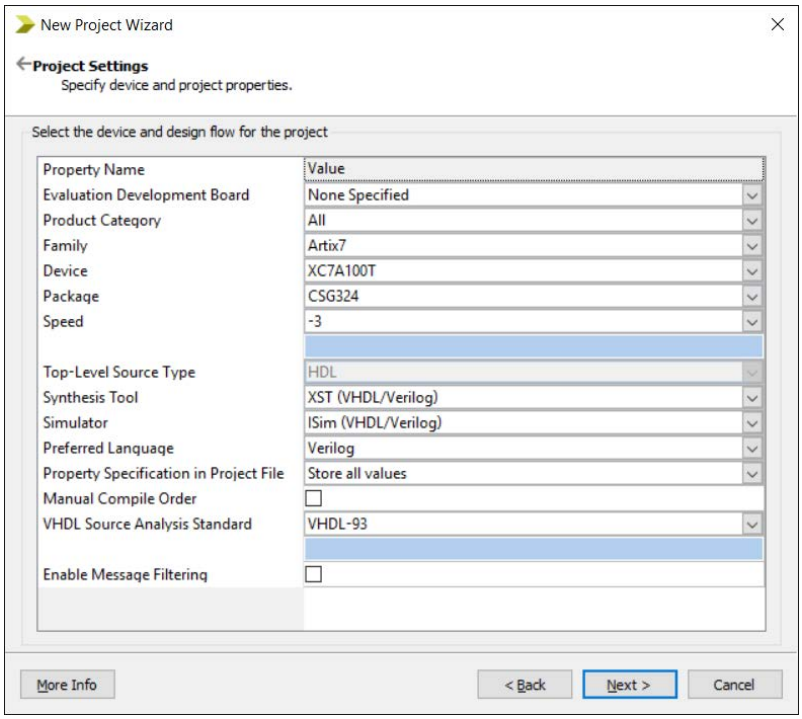


Figure 3.1: Project settings used to create a new ISE project for the Artix-7 FPGA on the Nexys 4 DDR

Figure 3.1 shows all of the appropriate options for the Artix-7 on the Nexys 4 DDR. Select Next and then Finish to create the new project.

From here, add any pre-written Verilog files using the Add Source... option in the Project menu. Use the New Source... option in the Project menu to create new source files including Verilog files. All of the project source files will appear in the Design Hierarchy panel. Once the top Verilog module is added or created, select that module in the Design Hierarchy panel and go to the Source menu and select Set as Top Module. Now when this module is selected in the Design Hierarchy Panel, all of the synthesis and other compilation options will appear in the Processes panel.

3.1.2 Spartan-6

In order to create a new ISE project for the Spartan-6 FPGA on the Nexys 3 board, open ISE and choose New Project in the File menu. In the New Project Wizard window, enter a name for the project and specify the location of the project files. It is suggested that all project files related to Single Chip Mote be checked into the repo in `scm-digital/proj/ise/spartan6/`. After selecting Next, the New Project Wizard will display the options for the device and design flow for the project. Figure 3.2 shows all of the appropriate options for the Spartan-6 on the Nexys 3. Select Next and then Finish to create the new project.

From here, add any pre-written Verilog files using the Add Source... option in the Project menu. Use the New Source... option in the Project menu to create new source files including Verilog files. All of the project source files will appear in the Design Hierarchy panel. Once the top Verilog module is added or created, select that module in the Design Hierarchy panel and go to the Source menu and select Set as Top Module. When this module is selected in the Design Hierarchy Panel

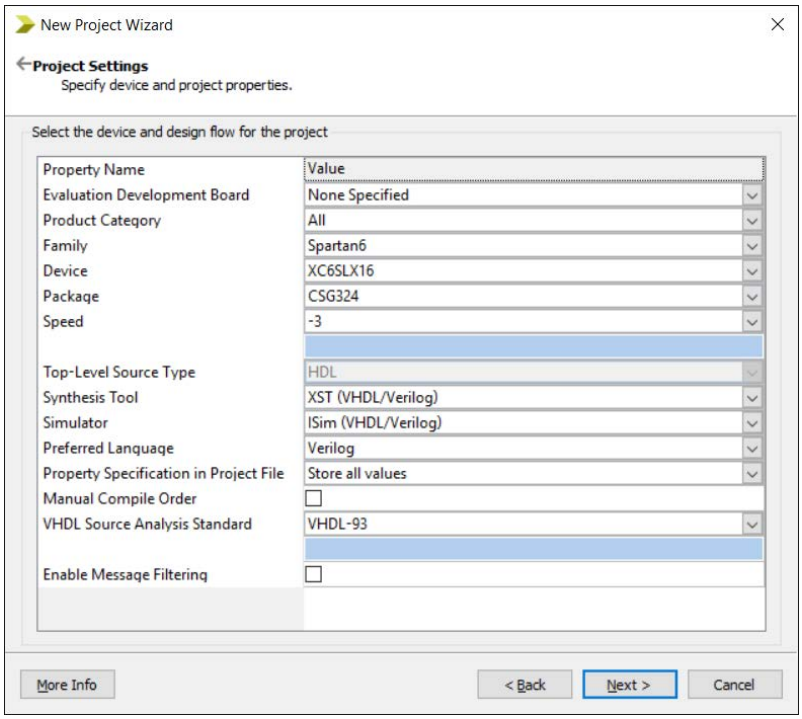


Figure 3.2: Project settings used to create a new ISE project for the Spartan-6 FPGA on the Nexys 3

again, all of the synthesis and other compilation options will appear in the Processes panel.

3.1.3 User Constraints File

All ISE projects require a User Constrains File (UCF) in order to map the top module's IOs to the pins on the FPGA package. Given that the FPGAs are soldered onto boards designed by Digilent, not all of the available pins on the package are routed to pins accessible on the Digilent boards. While generic UCF files for the Spartan-6 and Artix-7 FPGAs exist (or are generated using Xilinx tools), Digilent provides master UCF files for their Nexys 3 and Nexys 4 DDR boards listing only the pins that are accessible through one of the various connectors on the boards. These UCF files provide net names and descriptions in the comments of each line to describe how each of the actual FPGA pins map to a physical connector on the board. Digilent also publishes the schematics of their boards, containing the same information as the UCF file comments in a visual form. It is recommended that a clean UCF file is downloaded from Digilent's resource center for every project, and added in ISE using the Add Source... option in the Project menu.

Figure 3.3 contains an excerpt of the UCF file used for the Single Chip Mote digital system on the Nexys 4 DDR. The file is a modified version of the UCF file provided by Digilent for any Nexys 4 DDR design. All of the lines beginning with a hash symbol (#) are comments. A UCF file must only include pins on the FPGA that are currently in use. All unused pins must be either omitted from the file, or as seen in the example, commented-out.

The top line in Figure 3.3 contains the definition of the `CLK` net used for the 100MHz input clock. Underneath is a series of lines designating this net as a clock,

and specifying its properties such as frequency and duty cycle.

Each input or output is described using a line beginning with `NET`. This is then followed by the net name, in quotes. The net name corresponds to the name of the input or output in the top module. Input and output buses require that each signal in the bus have its own NET in the UCF file, with the index indicated using angle brackets (⟨⟩) instead of square brackets ([]).

The location of the pin connected to the net is specified after the net name, using `LOC=J15`, where J15 is a particular pin on the FPGA. In the case of the Nexys 4 DDR, the pin J15 is connected to one of the switches, hence why this net is included in the Switches section of the provided UCF. The original name of this net was `sw⟨0⟩` to indicate that it was connected to the first switch on the board. This is why it is recommended that the UCF files provided by Digilent are used. However, net names must be changed to match the top module (or vice versa) when using the UCF file provided by Digilent.

The `IOSTANDARD=LVCMOSS33` is an optional attribute, used to specify the attributes for the pin such as voltage, drive, and slew. It is recommended that the default IOSTANDARD specified in the Digilent UCF file is used and not changed unless the proper Xilinx documentation is first consulted.

The comments after each line indicate the name of the pin on the Artix-7 package. The name also contains information on how the pin may be used. For example, all pins with `MRCC` or `SRCC` in the name can be used as an input for clock signals. However, if the clock is single-ended, then the pin must also have a P in the second part of the name, for example `IO_L13P_T2_MRCC_15`. Differential clock inputs require P/N pairs. The `IO_L13P_T2_MRCC_15` pin is the only pin of its kind that is accessible through one of the Pmod connectors on the Nexys 4 DDR board, and therefore this single pin is used for both the input clock for the radio (see section 3.25 for more information) and the input clock for the 3 Wire Bus (see section 5.4 for more details).

3.2 Digital System Architecture Overview

Figure 3.4 contains a block diagram of the top module of the Single Chip Mote digital system, `uCONTROLLER`, along withs its inputs and outputs to/from the other parts of the Single Chip Mote, such as the analog/RF circuits. The Single Chip Mote digital system consists of one ARM Cortex-M0 DesignStart processor connected to various peripherals through a hierarchy of buses. These peripherals include instruction and data memory, a radio controller, a radio timer, an ADC controller, a UART transmitter and receiver, analog configuration registers for the radio, and general-purpose digital inputs and outputs.

The `PON` (short for Power-ON) module contains all of the hardware to generate the clocks and handle resets. This module is only required for the FPGA version of the Single Chip Mote digital system. On an ASIC, it is assumed that an external analog circuit handles the generation of all clock and reset signals.

The AHB-Lite is a 32-bit bus used by the ARM Cortex-M0 to connect to memory and peripherals. The main AHB-Lite bus is composed of two modules, `AHBDCD` and `AHBMUX`. An example of these modules is provided in the ARM Cortex-M0 DesignStart kit and is used as the basis for this design. This bus has 1 master, the ARM Cortex-M0 and 5 slaves: the instruction memory (`AHBIMEM`), another

```
NET "CLK"   LOC = "E3"  |IOSTANDARD = "LVCMOS33";       #Bank = 35, Pin name = #
    IO_L12P_T1_MRCC_35,     Sch name = clk100mhz
NET "CLK" TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 100 MHz HIGH 50%;

## Switches
NET "gp_in<0>" LOC=J15 |IOSTANDARD=LVCMOS33; #IO_L24N_T3_RS0_15
NET "gp_in<1>" LOC=L16 |IOSTANDARD=LVCMOS33; #IO_L3N_T0_DQS_EMCCLK_14
NET "gp_in<2>" LOC=M13 |IOSTANDARD=LVCMOS33; #IO_L6N_T0_D08_VREF_14
NET "gp_in<3>" LOC=R15 |IOSTANDARD=LVCMOS33; #IO_L13N_T2_MRCC_14
#NET "sw<4>"   LOC=R17 |IOSTANDARD=LVCMOS33; #IO_L12N_T1_MRCC_14
#NET "sw<5>"   LOC=T18 |IOSTANDARD=LVCMOS33; #IO_L7N_T1_D10_14
#NET "sw<6>"   LOC=U18 |IOSTANDARD=LVCMOS33; #IO_L17N_T2_A13_D29_14
#NET "sw<7>"   LOC=R13 |IOSTANDARD=LVCMOS33; #IO_L5N_T0_D07_14
...
## Buttons
NET "RESETn"   LOC=C12 |IOSTANDARD=LVCMOS33; #IO_L3P_T0_DQS_AD1P_15
#NET "btnc"    LOC=N17 | IOSTANDARD=LVCMOS33; #IO_L9P_T1_DQS_14
#NET "btnd"    LOC=P18 | IOSTANDARD=LVCMOS33; #IO_L9N_T1_DQS_D13_14
#NET "btnl"    LOC=P17 | IOSTANDARD=LVCMOS33; #IO_L12P_T1_MRCC_14
#NET "btnr"    LOC=M17 | IOSTANDARD=LVCMOS33; #IO_L10N_T1_D15_14
#NET "btnu"    LOC=M18 | IOSTANDARD=LVCMOS33; #IO_L4N_T0_D05_14
...
## Pmod Header JB
NET "data_3wb"  LOC=D14 |IOSTANDARD=LVCMOS33; #IO_L1P_T0_AD0P_15
NET "latch_3wb" LOC=F16 |IOSTANDARD=LVCMOS33; #IO_L14N_T2_SRCC_15
#NET "jb<3>"    LOC=G16 |IOSTANDARD=LVCMOS33; #IO_L13N_T2_MRCC_15
#NET "jb<4>"    LOC=H14 |IOSTANDARD=LVCMOS33; #IO_L15P_T2_DQS_15
NET "tx_clk"    LOC=E16 |IOSTANDARD=LVCMOS33; #IO_L11N_T1_SRCC_15
NET "tx_dout"   LOC=F13 |IOSTANDARD=LVCMOS33; #IO_L5P_T0_AD9P_15
NET "rx_din"    LOC=G13 |IOSTANDARD=LVCMOS33; #IO_0_15
NET "rx_clk"    LOC=H16 |IOSTANDARD=LVCMOS33; #IO_L13P_T2_MRCC_15
```

Figure 3.3: An example of a UCF file for the Artix-7 on the Nexys 4 DDR board

AHB-Lite bus (through the arbiter `AHBLiteArbiter_V2`), the direct memory access controller (`DMA_V2`), the radio timer (`RFTIMER`), and an APB bus. The second AHB-Lite bus is designed to have two masters (using the `AHBLiteArbiter_V2` module as an arbiter) and two slaves, the data memory (`AHBDMEM`) and the radio controller (`RFcontroller`). This structure was chosen because the two slaves need to be accessed by both the Cortex-M0 and the DMA. The DMA is used to automatically transfer radio packet data between the radio controller and the data memory without any intervention from the Cortex-M0. The first AHB-Lite bus is referred to as the AHB. The second AHB-Lite bus is referred to as the AHBsub.

The APB is a 16-bit peripheral bus used to access peripherals that do not require the full 32-bit data size or the low latency of the AHB-Lite. The APB is connected to the AHB-Lite using the `AHB2APB` module designed by Bigazzi. The bus itself is composed of the `APBMUX` module designed by Bigazzi. This bus has four slaves: the ADC controller (`APBADC_V2`), the UART transmitter/receiver (`APBUART`), configuration registers for the analog circuits of the Single Chip Mote (`APB_ANALOG_CFG`), and a small set of digital inputs and outputs (`APBGPIO`).

For more information on the AHB-Lite protocol, the APB protocol, and each of the modules mentioned above, see the rest of this chapter.

3.3 ARM Cortex-M0 Memory Map Specification

The AHB-Lite bus on the ARM Cortex-M0 allows for the use of 4GB addressable memory with 32-bit addresses. In ARM documentation this addressable memory

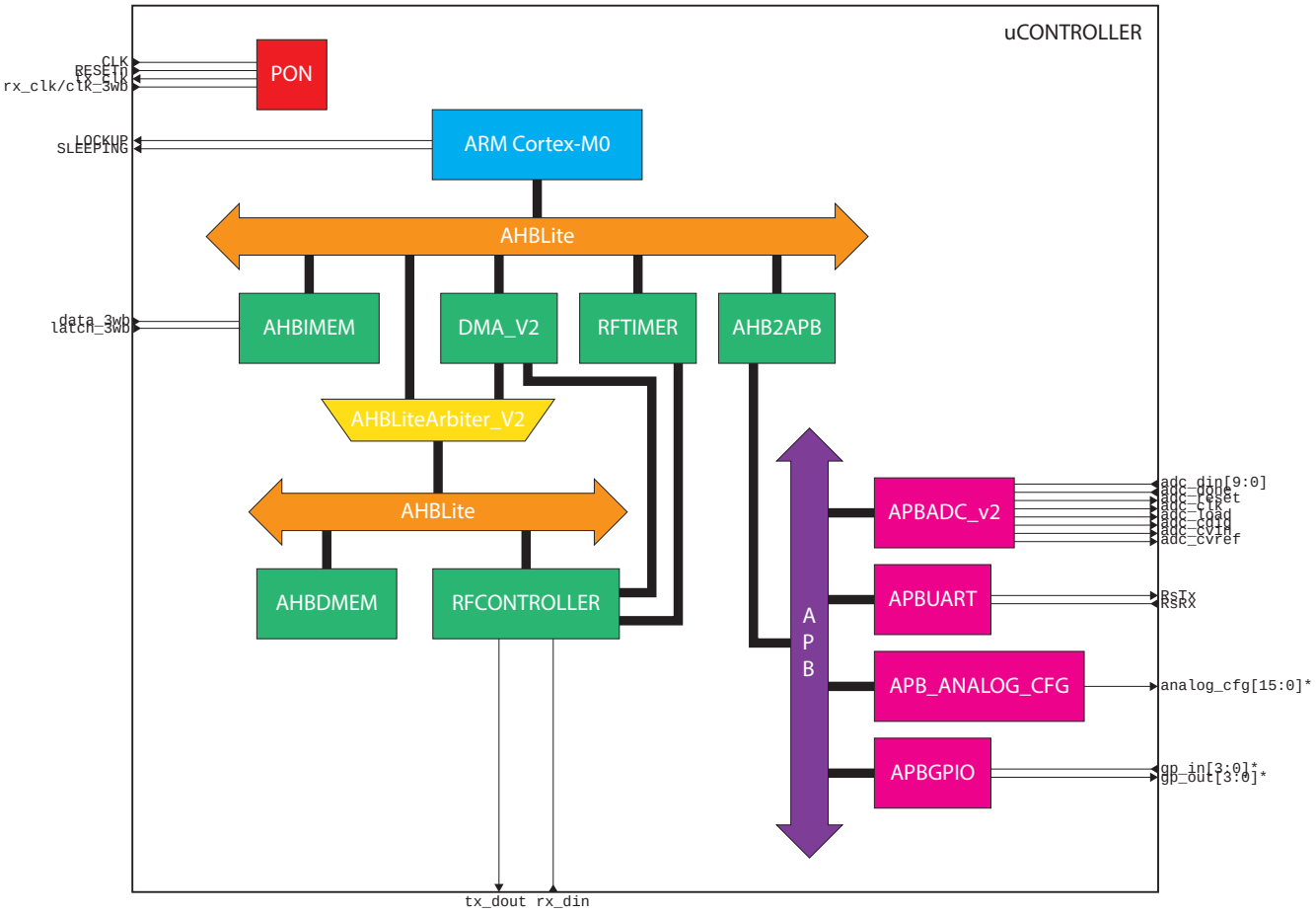


Figure 3.4: Block diagram of the Single Chip Mote digital system. Inputs/outputs with a * have parameterizable bus widths.

Address Range	Address Prefix	Memory Region	Description
0x00000000-0x1FFFFFFF	0x00-0x1F	Code	Executable region for instruction memory. This can be ROM, RAM, or both. Data can also go here.
0x20000000-0x3FFFFFFF	0x20-0x3F	SRAM	Executable region for data memory. Instructions can also go here.
0x40000000-0x5FFFFFFF	0x40-0x5F	Peripheral	External device memory. This memory is not executable.
0x60000000-0x9FFFFFFF	0x60-0x9F	External RAM	Executable region for external data memory.
0xA0000000-0xDFFFFFFF	0xA0-0xDF	External Device	Non-executable region for external device memory.
0xE0000000-0xE00FFFFF	0xE00	Private Peripheral Bus	Non-executable region including special Cortex-M0 registers such as the NVIC, system timer, and system control block.
0xE0100000-0xFFFFFFFF	0xE01-0xFFF	Device	Implementation-specific device memory. This region is reserved for additional ARM Cortex-M0 features not available on the DesignStart processor.

Figure 3.5: Memory map of the ARM Cortex-M0

is referred to as the memory map. This does not mean that all Cortex-M0 designs contain at least 4GB of memory storage; instead, all Cortex-M0 designs have 4GB of address space used to access either actual memory or memory-mapped peripherals.

Each address refers to a single byte in the memory; however, in certain regions of the memory map, the ARM Cortex-M0 DesignStart processor only allows word-aligned accesses. Overall, it is recommended that all memory-mapped peripherals used word-aligned addresses. In this case, the only valid addresses are multiples of 4, such as `0x0101010C` or `0xABCDEF08`.

Continuous regions of this address space are reserved for instruction memory, data memory, debug access, and peripherals. Addresses are divided into these regions based on the upper bits of the address. In the Single Chip Mote digital system, the 8 upper bits (referred to in this document as the address prefix) are used to distinguish between memory regions or memory-mapped peripherals. One notable exception is the Private Peripheral Bus, with a 12-bit prefix of `0xE00`.

Figure 3.5 contains a summary of the memory map for the ARM Cortex-M0. Not all regions of this memory map are currently used in the Single Chip Mote digital system, such as the external data memory or the external device memory.

For more information on the ARM Cortex-M0 memory map see the Cortex-M0 Devices Generic User Guide [10]. A copy is also found in `scm-digital/doc/`.

3.4 AMBA 3 AHB-Lite Protocol

This section summarizes the basics of the AMBA 3 AHB-Lite protocol, including specific details involving the implementation of this bus on the Single Chip Mote digital system.

The AHB-Lite bus is a high-bandwidth single-master bus. All slaves on this bus use the same clock, `HCLK`, and have the same reset, `HRESETn`. On the Single Chip

Mote digital system, almost all modules use HCLK and HRESETn as well, including APB slaves.

The bus master drives the following signals:

HADDR[31:0] The address bus.

HBURST[2:0] Indicates whether the transfer is a single transfer or some kind of burst. The DesignStart processor does not generate any BURST transfers [29]. Therefore, this signal is omitted in the Single Chip Mote digital system.

HMASTLOCK Indicates that the current transfer is part of a locked sequence. The DesignStart processor does not generate any locked transfers [29]. Therefore, this signal is omitted in the Single Chip Mote digital system.

HPROT[3:0] Protection control signal. This can be ignored by the slave [29], and is omitted in the Single Chip Mote digital system.

HSIZE[2:0] Indicates the size of the transfer as a byte, halfword, or word.

HTRANS[1:0] Indicates the transfer type. The DesignStart processor only uses non-sequential transfers [29], making HTRANS[0] always 0. Therefore, HTRANS[0] is omitted in the Single Chip Mote digital system.

HWDATA[31:0] The write data when the master writes to a slave.

HWRITE Indicates if the transfer is a read (0) or write (1).

Slaves each drive their own set of the following signals:

HRDATA[31:0] The read data given to the master when it reads from a slave.

HREADYOUT Indicates that the transfer is finished. As long as this signal is 0, the master waits until it is 1 before considering the transfer complete.

HRESP Used to indicate an error in the transfer. This is not used by any of the slaves in the Single Chip Mote digital system and is omitted.

The AHBDCD module takes the address for the current transfer and selects the correct slave using the address prefix. This module is connected to the AHBMUX module, which selects the correct set of slave signals to send to the master. The AHBDCD module also drives the various HSEL signals to each slave, used to indicate that the transfer is intended for that particular slave.

Each bus transfer requires two phases, the address phase and the data phase. In the address phase, the master sets the HADDR, HWRITE, HTRANS, and other relevant signals. The next cycle is the beginning of the data phase, where the slave sets HRDATA (if the transfer is a read), performs a write (if the transfer is a write) and sets HREADYOUT if the transfer is complete. The slave can stall the master by leaving HREADYOUT low.

This protocol supports pipelined transfers. This means that the data phase of one transfer can also be the address phase of the next transfer. The address phase signals remain constant/valid while the master is stalled during a data phase, and do not change until HREADYOUT from the slave is high.

For more information on the AHB-Lite, see the AMBA 3 AHB-Lite Protocol Specification [1]. A copy is also found in scm-digital/doc/.

3.5 AMBA 3 APB Protocol

This section summarizes the basics of the AMBA 3 APB protocol, including specific details involving the implementation of this bus on the Single Chip Mote digital system.

The APB bus is a low-power reduced-complexity bus for peripherals that do not require high bandwidth or low latency. The protocol defines separate clock (`PCLK`) and reset (`PRESETn`) signals for APB slaves. In the Single Chip Mote digital system, `HCLK` and `HRESETn` are used instead.

The master of this bus is the composed of the bridge connecting the AHB and APB (the `AHB2APB` module) and the `APBMUX` module that indicates which slave is being accessed and sends the correct set of slave signals to the master. The master drives the following signals:

`PADDR[15:0]` The address bus.

`PSEL` Each slave has one of these signals to indicate that the transfer is intended for that particular slave.

`PENABLE` Indicates the second and subsequent cycles of a transfer.

`PWRITE` Indicates if the transfer is a read (0) or write (1).

`PWDATA[15:0]` The write data when the master writes to a slave.

The slaves each drive their own set of the following signals:

`PREADY` Indicates that the transfer is finished. As long as this signal is 0, the master waits until it is 1 before considering the transfer complete.

`PRDATA[15:0]` The read data given to the master when it reads from a slave.

`PSLVERR` Indicates a transfer failure. This is optional and is not used by any of the slaves in the Single Chip Mote digital system and is omitted.

Each bus transfer requires two phases, the setup phase and the access phase. The first clock cycle is the setup phase, where `PADDR`, `PWDATA`, and `PWRITE` are set by the master. During the second clock cycle, the `PENABLE` signal is asserted to indicate that it is now the access phase. All control signals stay the same during the access phase as the APB protocol does not allow for pipelined transfers. The access phase is extended by keeping the `PREADY` signal low. The transfer completes once `PREADY` is high.

For more information on the APB, see the AMBA 3 APB Protocol Specification [2]. A copy is also found in `scm-digital/doc/`.

3.6 Header Files and Parameters

The Verilog for the Single Chip Mote digital system contains two header files, `SYS_PROP.vh` and `REGISTERS.vh`, used parameterize the design and make it easy to modify.

3.6.1 SYS_PROP.vh

SYS_PROP.vh contains `define statements used to tweak module parameters (such as the baud rate for APBUART or the number of outputs in APBGPIO). Each parameterizable module has a parameter defined in the module definition. If a module is not instantiated in the top level, then its parent module defines the same parameter, and passes the value on during instantiation. If there are several submodules between the top level and the module requiring the parameter, then each module in that chain must instantiate the parameter and pass it down. At the top level, the name defined in SYS_PROP.vh is passed into the module instantiation.

For example, consider the compare_unit module, with the following definition:

```
module compare_unit(
        ...
        ...
        ...
);

// Parameters
parameter COUNTER_WIDTH = 32;
```

This module is instantiated in the RFTIMER module using the following syntax:

```
compare_unit #(.COUNTER_WIDTH(COUNTER_WIDTH)) u_compare_unit (
        ...
        ...
        ...
);
```

The RFTIMER module also contains the same parameter, COUNTER_WIDTH, along with its own parameters:

```
module RFTIMER(
        ...
        ...
        ...
);

// Parameters
parameter NUM_COMPARE_UNITS = 8;
parameter NUM_CAPTURE_UNITS = 4;
parameter COUNTER_WIDTH = 32;
```

And RFTIMER is instantiated at the top level, uCONTROLLER, using the values defined in SYS_PROP.vh:

```
RFTIMER #(
        .NUM_COMPARE_UNITS(`RFTIMER_NUM_COMPARE_UNITS),
        .NUM_CAPTURE_UNITS(`RFTIMER_NUM_CAPTURE_UNITS),
        .COUNTER_WIDTH(`RFTIMER_COUNTER_WIDTH)
) u_RFTIMER (
        ...
        ...
        ...
);
```

```
// RFTIMER Specifications
`define RFTIMER_NUM_COMPARE_UNITS    8   // the number of compare units for the
    timer, if this changes REGISTERS.vh must be updated
`define RFTIMER_NUM_CAPTURE_UNITS    4   // the number of capture units for the
    timer, if this changes REFISTERS.vh must be updated
`define RFTIMER_COUNTER_WIDTH       32  // the width of the counter for the timer,
    maximum is 32
```

Not all parameters need to be exposed all the way to the top level. For example, state encodings are typically enumerated using parameters. However, these states

are specific only to the module itself and are not system-level parameters. Therefore, it is recommended that they are defined as localparams instead of parameters.

3.6.2 REGISTERS.vh

`REGISTERS.vh` contains `` `define `` statements used to assign addresses to peripherals on the AHB and APB and each of their registers. This is first done by assigning an 8-bit address prefix to each peripheral. Then these prefixes are used to define a base address for each peripheral. Then this base address is used to define all register addresses for that peripheral.

As stated in section 3.3, all addresses with a prefix in the range of `0x40-0x5F` can be used for peripheral devices. In the Single Chip Mote digital system, all addresses with a prefix in the range of `0x40-0x4F` are reserved for AHB peripherals, and all addresses with a prefix in the range of `0x50-0x5F` are reserved for APB peripherals. The only exceptions are the instruction instruction memory (with a prefix of `0x00`), the bootloader (`0x01`), and the data memory (`0x20`), as these are not really system peripherals but are memories used by the ARM Cortex-M0. The first section of `REGISTERS.vh` defines these prefixes:

```
// AHB Address Prefixes
`define AHB_PREFIX__IMEM            8'h00
`define AHB_PREFIX__BOOTLOADER      8'h01
`define AHB_PREFIX__DMEM            8'h20
`define AHB_PREFIX__RFCONTROLLER    8'h40
`define AHB_PREFIX__DMA             8'h41
`define AHB_PREFIX__RFTIMER         8'h42
`define AHB_PREFIX__APB             8'b0101_xxxx

// APB Address Prefixes
`define APB_PREFIX__ADC             8'h50
`define APB_PREFIX__UART            8'h51
`define APB_PREFIX__ANALOG_CFG      8'h52
`define APB_PREFIX__GPIO            8'h53
```

These prefixes are also used in the `AHBDCD` and `APBMUX` modules to determine the AHB/APB signals for each slave based on address.

The next section of `REGISTERS.vh` uses these prefixes to define a base address for each peripheral:

```
// AHB Peripheral Base Addresses
`define AHB_BASE__IMEM              { `AHB_PREFIX__IMEM, 24'h00_0000 }
`define AHB_BASE__BOOTLOADER        { `AHB_PREFIX__BOOTLOADER, 24'h00_0000 }
`define AHB_BASE__DMEM              { `AHB_PREFIX__DMEM, 24'h00_0000 }
`define AHB_BASE__RFCONTROLLER      { `AHB_PREFIX__RFCONTROLLER, 24'h00_0000 }
`define AHB_BASE__DMA               { `AHB_PREFIX__DMA, 24'h00_0000 }
`define AHB_BASE__RFTIMER           { `AHB_PREFIX__RFTIMER, 24'h00_0000 }
`define AHB_BASE__APB               { `AHB_PREFIX__APB, 24'h00_0000 }

// APB Peripheral Base Addresses
`define APB_BASE__ADC               { `APB_PREFIX__ADC, 8'h00 }
`define APB_BASE__UART              { `APB_PREFIX__UART, 8'h00 }
`define APB_BASE__ANALOG_CG         { `APB_PREFIX__ANALOG_CFG, 8'h00 }
`define APB_BASE__GPIO              { `APB_PREFIX__GPIO, 8'h00 }
```

This base address is the first address in the region of addresses allocated to the peripheral. For example, the `RFcontroller` module has a prefix of `0x40` and thus a base address of `0x40000000`. All register addresses are described relative to an offset from the base. Any changes in the prefix automatically change the base address and all register addresses for that peripheral. Note that the APB uses 16-bit addresses instead of 32-bit addresses.

The last section of `REGISTERS.vh` uses the base addresses to define all of the register addresses for the AHB/APB peripherals:

```
// RF Controller
`define RFCONTROLLER_REG__CONTROL           `AHB_BASE__RFCONTROLLER + 32'h0000_0000
`define RFCONTROLLER_REG__STATUS            `AHB_BASE__RFCONTROLLER + 32'h0000_0004
`define RFCONTROLLER_REG__TX_DATA_ADDR      `AHB_BASE__RFCONTROLLER + 32'h0000_0008
`define RFCONTROLLER_REG__TX_PACK_LEN       `AHB_BASE__RFCONTROLLER + 32'h0000_000C
...

// GPIO
`define APBGPIO_REG__INPUT          `APB_BASE__GPIO          + 16'h0000
`define APBGPIO_REG__OUTPUT         `APB_BASE__GPIO          + 16'h0004
```

These address definitions are used in any Verilog code that refers to specific register addresses or prefixes. Using define statements for each address ensures that they are easily accessible in one file and all changes propagate to any modules relying on these addresses or prefixes.

The `REGISTERS.vh` file could also be parsed by a script to generate a C header file for the purposes of software development. Another option is to define all prefixes and registers in a CSV file, and use a script to create both `REGISTERS.vh` and a C header file. Neither of these options are currently implemented, though it is recommended that such a script is created for future work on the Single Chip Mote digital system.

3.7 Module Hierarchy

Figure 3.6 contains a list of all of the Single Chip Mote digital system modules and their submodules. This list matches the Design Hierarchy panel in ISE Project Navigator.

3.8 uCONTROLLER

3.8.1 Description

`uCONTRLLER` (found in `TOP_SYS.v`) is the top module of the Single Chip Mote digital system. This module instantiates the Cortex-M0, power-on moudle, the AHB/APB peripherals, and the AHB/APB busses. This module also connects the above mentioned modules together and to the main inputs and outputs of the Single Chip Mote digital system.

3.8.2 Input/Output Ports

`CLK` 100MHz clock input from the FPGA board.

`RESETn` Reset input from a button on the FPGA board.

`LOCKUP` Output to an LED on the FPGA board, connected to the LOCKUP output from `CORTEXM0DS`.

`SLEEPING` Output to an LED on the FPGA board, connected to the SLEEPING output from `CORTEXM0DS`.

`RsRx` Receive data input for UART.

```
uCONTROLLER
    PON
        pb_debounceRESET
        ClockDiv
    CORTEXM0DS
        cortexm0ds_logic
    DMA_V2
    AHBDCD
    AHBMUX
    AHBIMEM
        instruction_ROM
        instruction_RAM
    RFTIMER
        compare_unit
        capture_unit
    AHBLiteArbiter_V2
    AHBDCDsub
    AHBMUXsub
    AHBDMEM
        dmem_ram
    RFCONTROLLER
        tx_fifo2
            tx_fifo_mem
            tx_rdptr_empty
            tx_wrptr_full
            tx_async_comp
        spreader
            symbol2chips
        bit_sync
        corr_despreader
            correlator
        bus_sync
        rx_fifo
            rx_fifo_mem
            rx_rdptr_empty
            rx_wrptr_full
            rx_async_comp
        crcParallel
    AHB2APB
    APBMUX
    APBADC_V2
    APBUART
        BAUDGEN
        FIFO
        UART_RX
        UART_TX
    APB_ANALOG_CFG
    APBGPIO
```

Figure 3.6: Module Hierarchy for the Single Chip Mote digital system

`RsTx` Transmit data output for UART.

`adc_din[9:0]` Data input from the ADC.

`adc_done` Done input from the ADC.

`adc_reset` Reset output to the ADC.

`adc_clk` Clock output to the ADC.

`adc_load` Load output to the ADC.

`adc_cdig` Cdig output to the ADC.

`adc_cvin` Cvin output to the ADC.

`adc_cvref` Cvref output to the ADC.

`analog_cfg[(`ANALOGCFG_NUM_REG*16)-1:0]` Analog configuration outputs to the rest of the Single Chip Mote system. The size of this port depends on the number of 16-bit analog configuration registers in the `APB_ANALOG_CFG` module.

`gp_in[`GPIO_NUM_INPUTS-1:0]` General-purpose digital inputs. The size of this port depends on the number of general-purpose digital inputs in the `APBGPIO` module.

`gp_in[`GPIO_NUM_OUTPUTS-1:0]` General-purpose digital outputs. The size of this port depends on the number of general-purpose digital outputs in the `APBGPIO` module.

`rx_clk` Input clock aligned with the data received from the radio circuit.

`rx_din` Data input for data received from the radio circuit. On the FPGAs this input is also used as the clock input for the 3 Wire Bus used for bootloading (chapter 5).

`tx_clk` Output clock aligned with the data sent to the radio circuit.

`tx_dout` Data output for data sent to the radio circuit.

`data_3wb` Data input for the 3 Wire Bus used for bootloading (chapter 5).

`latch_3wb` Latch input for the 3 Wire Bus used for bootloading (chapter 5).

3.8.3 Design Details

`uCONTROLLER` is the top-level module, used to connect all other modules to one another and to the inputs and outputs of the design as a whole. It contains the instantiations of all the other modules (see Figure 3.6 for its submodules), and declarations for the wires connecting these modules. Any new buses or peripherals must be instantiated in this module.

3.9 CORTEXM0DS

3.9.1 Description

This module, provided by ARM in the DesignStart kit, is an interface between the obfuscated Verilog describing the ARM Cortex-M0 DesignStart processor (in `cortexm0ds_logic`) and the rest of the system.

3.9.2 Input/Output Ports

`HCLK` Clock input.

`HRESETn` Asynchronous reset input.

`HADDR[31:0]` AHB transfer address output.

`HBURST[2:0]` AHB burst output. This is always 0 and is omitted in the Single Chip Mote digital system.

`HMASTLOCK` AHB locked transfer output. This is always 0 and is omitted in the Single Chip Mote digital system.

`HPROT[3:0]` AHB transfer protection output. AHB slaves do not have to use this signal and therefore it is omitted in the Single Chip Mote digital system.

`HSIZE[2:0]` AHB transfer size output. Indicates a byte, half-word, or word transfer.

`HTRANS[1:0]` AHB transfer type output. This is only set to idle or non-sequential, and therefore `HTRANS[0]` is always 0. `HTRANS[0]` is omitted in the Single Chip Mote digital system.

`HWDATA[31:0]` AHB write data output.

`HWRITE` AHB write output. Indicates that the transfer is a write when 1.

`HRDATA[31:0]` AHB read data input.

`HREADY` AHB transfer finished input. This is used to stall the Cortex-M0 when the slave is not finished with the transfer.

`HRESP` AHB error response input. AHB slaves in the Single Chip Mote digital system do not use this signal and therefore it is assigned to 0.

`NMI` Non-maskable interrupt input. This interrupt is not used in the Single Chip Mote digital system and therefore it is assigned to 0.

`IRQ[15:0]` Interrupt request inputs for up to 16 interrupts. Only four are in use right now and the rest are assigned to 0.

`TXEV` Event output. This is not used in the Single Chip Mote digital system.

`RXEV` Event input. This is not used in the Single Chip Mote digital system and is assigned to 0.

`LOCKUP` Lockup output. Indicates that the core is locked-up.

`SYSRESETREQ` System reset request output. Used to send a request for a reset to the `PON` module.

`SLEEPING` Sleeping output. Indicates that the core and NVIC are sleeping.

3.10 cortexm0ds_logic

3.10.1 Description

This module, provided by ARM in the DesignStart kit, contains the obfuscated Verilog describing the ARM Cortex-M0 DesignStart processor. This module is instantiated only in `CORTEXM0DS` and must not be instantiated by any other module in the design. This module must not be modified.

3.11 PON

3.11.1 Description

This module is designed to handle all of the clock and reset signals in the Single Chip Mote digital system. This includes dividing down the 100MHz input clock into all other required clocks, buffering additional clock inputs, debouncing the input reset signal, and listening for reset requests from the Cortex-M0.

3.11.2 Input/Output Ports

`CLK_100Mz` 100MHz input clock from the FPGA board.

`CLK_RX_IN` External clock input for both receiving radio packets (`CLK_RX`, 2MHz) and receiving bootloading data (chapter 5) over the 3 Wire Bus (`CLK_3WB`, 5MHz).

`CLK_RX_EN` Clock enable input for the radio receive clock (`CLK_RX`). This enable signal is used to ensure that the `RFcontroller` module only uses the external clock when listening for radio packets.

`CLK_3WB_EN` Clock enable input for the 3 Wire Bus clock (`CLK_3WB`). This enable signal is used to ensure that the `AHBIMEM` module only uses the external clock when listening for bootloading data (chapter 5).

`RESETn_in` Input reset signal from a button on the FPGA board.

`SYSRESETREQ` Reset request input from the Cortex-M0.

`CLK_5MHz` 5MHz clock output used by most of the Single Chip Mote digital system. This is assigned at the top level to `HCLK`.

`CLK_TX` 2MHz output clock used by the `RFcontroller` module to send radio packets. This is assigned at the top level to `CLK_TX`.

`CLK_TX_OUT` 2MHz clock output. The output is a copy of `CLK_TX` routed to an output on FPGA. This output is used by the radio circuit to send packets. This is assigned at the top level to the `tx_clk` output pin.

`CLK_RX` 2MHz clock output used by the `RFcontroller` module to listen for radio packets. This is a buffered version of the `CLK_RX_IN` input clock, enabled or disabled with the `CLK_RX_EN` input. This is assigned at the top level to `CLK_RX`.

`CLK_3WB` 5MHz clock output used by the `AHBIMEM` module to listen for bootloading data (chapter 5). This is a buffered version of the `CLK_RX_IN` input clock, enabled or disabled with the `CLK_3WB_EN` input. This is assigned at the top level to `CLK_3WB`.

`CLK_RFTIMER` 500kHz clock output used by the `RFTIMER` module for its timer. This is assigned at the top level to `CLK_RFTIMER`.

`HARD_RESETn_out` Hard reset output. The hard reset is active-low. This reset is activated when the external reset button is pressed and not when `SYSRESETREQ` is asserted. This hard reset is only used in the `AHBIMEM` module for bootloading purposes (chapter 5). This is assigned at the top level to `HARD_RESETn`.

`SOFT_RESETn_out` Soft reset output. This soft reset is active-low. This reset is activated when either the external reset button is pressed or when `SYSRESETREQ` is asserted. This soft reset is used by every module in the Single Chip Mote digital system. This is assigned at the top level to `HRESETn`.

3.11.3 Design Details

On an FPGA this module instantiates special primitives used to deal with buffering input clocks, dividing input clocks, attaching derived clocks to the FPGA's clock nets, and buffering any output clocks. See Xilinx documentation for more information on how to use the primitives mentioned in this section. This module uses the `pb_debounceRESET` module written by Bigazzi, to debounce the input reset signal. This module also uses the `ClockDiv` module written by Bigazzi to divide the 100MHz clock into slower clocks that cannot be achieved using FPGA primitives. Note that there are slight differences in the available FPGA primitives on the Artix-7 and Spartan-6.

Input Clock Buffering

Both the Artix-7 and Spartan-6 versions of the `PON` module use the `IBUFG` primitive to buffer input clocks. This is required whenever a clock is fed into the FPGA from outside. There is an instantiated `IBUFG` module for both `CLK_100MHz` and `CLK_RX`.

Clock Division

The Artix-7 and Spartan-6 FPGAs use separate primitives for clock division. The Artix-7 version of this module uses `MMCME2_ADV` and the Spartan-6 version uses `DCM_SP`. On both FPGAs these primitives have a limited range of division. With a 100MHz input clock, the lowest possible frequency is 2.5MHz. Therefore these primitives are only used to divide the 100MHz input to 5MHz for `CLK_5MHz`. All slower

clocks, such as CLK_TX at 2MHz, and CLK_RFTIMER at 500kHz, use the ClockDiv module.

While Xilinx provides documentation on how to instantiate the primitives for clock division, it is not recommended that this be done manually. Instead, it is better to use the Clocking Wizard in CORE Generator to create a core that meets the required specifications (in this case to divide 100MHz to 5MHz). Once the core is been created and added to the project, running the View HDL Functional Model process (see Figure 3.7) generates the Verilog code for instantiating all of the required clocking primitives. The code in PON was created by copying the generated code and adding the additional input clocks and clock dividers.

The ClockDiv module is a counter-based clock divider with an output that inverts after a maximum count is reached. Therefore, the output clock period is $T_{out} = 2 \times MAX\_COUNT \times T_{in}$. This method of clock division is not recommended on FPGAs; however, this method is the only way to generate clocks less than 2.5MHz, and therefore is used for CLK_TX and CLK_RFTIMER.

Derived Clock Buffering

After the input clocks have been divided, using either primitives or ClockDiv, the derived clocks must be attached to a clock buffer in order to route the clocks on the dedicated clock nets within the FPGAs. Any input clocks that are not divided, such as CLK_RX_IN, must also be attached to a clock buffer directly from the input clock buffer primitive. This is done using either the BUFG or BUFGCE primitives. The BUFG primitive is used for continuously running clocks such as CLK_5MHz, CLK_TX, and CLK_RFTIMER. The BUFGCE primitive is used for all clocks requiring an enable signal such as CLK_RX and CLK_3WB.

Output Clock Buffering

Any clocks sent to an output pin on the FPGA must first be routed to an the input of an output buffer through a method called clock forwarding. On the Artix-7, clock forwarding is accomplished by first feeding the clock into an ODDR primitive, and then feeding the output of that into an OBUF primitive. On the Spartan-6, the ODDR2 primitive is used instead of the ODDR.

Resets

The pb_debounceRESET module is designed by Bigazzi to debounce the input reset signal before sending it to the rest of the Single Chip Mote digital system. This is necessary because pressing buttons on the FPGA board leads to unstable or fluctuating outputs before the signal converges. A simple debouncer waits for the signal to be stable for multiple cycles before changing the output, to remove any glitches. This module also sends out a pulse when the signal is stable. This module is a slightly modified version of the pb_debouce module provided in the ARM Cortex-M0 DesignStart kit. Bigazzi modified this code such that the output is active-low instead of active-high. The pulse generated when the input is stable is used as the hard reset, HARD_RESETn_out.

In order to deal with reset requests from the Cortex-M0, the hard reset signal is also combined with the SYSRESETREQ signal and then clocked into a register

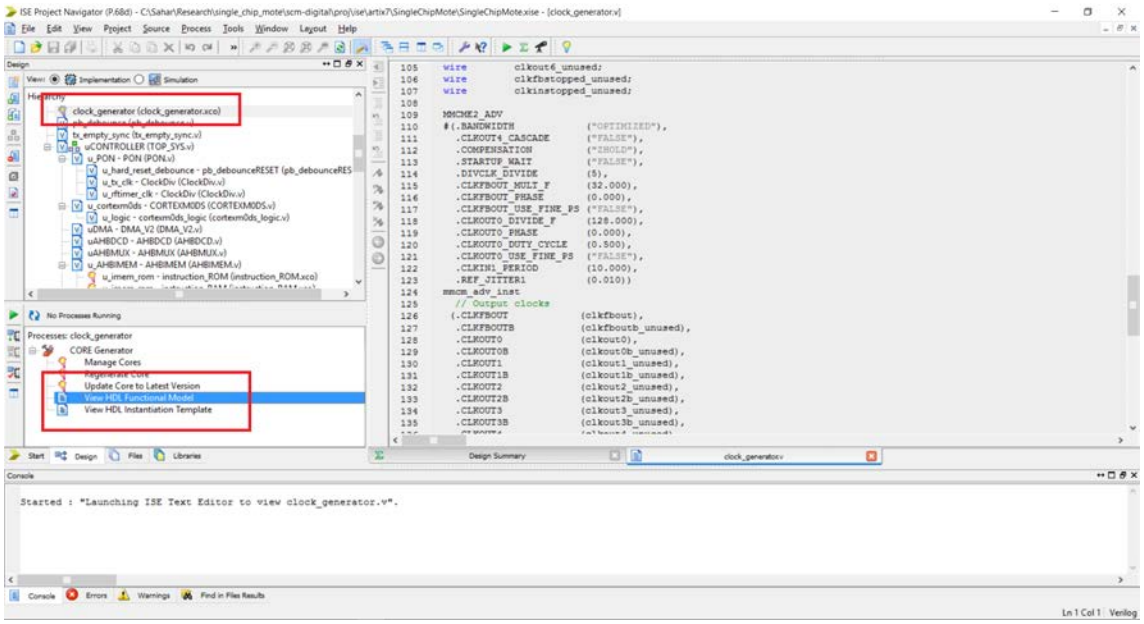


Figure 3.7: View HDL Functional Model for a generated core

(see Figure 3.8). The output of this register is the soft reset, `SOFT_RESETn_out`. The `SYSRESETREQ` signal is asynchronous, and must be synchronized outside of the Cortex-M0 before being used for a reset, hence the register. While this does force the soft reset to be 1 cycle behind the hard reset, only one part of the AHBIMEM module uses the hard reset signal instead of the soft reset, and the timing difference is acceptable.

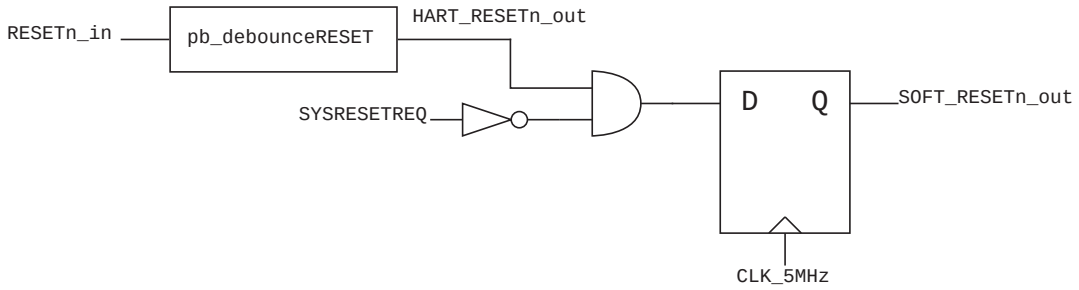


Figure 3.8: Reset handling in the `PON` module

3.12 pb_debounceRESET

3.12.1 Description

This module is used to debounce the output signal from a button on the FPGA board used as the reset input to the Single Chip Mote digital system. This module is a slightly modified version of the `pb_debouce` module provided in the ARM Cortex-M0 DesignStart kit. Bigazzi modified this code such that the output is active-low instead of active-high.

3.12.2 Input/Output Ports

`clk` The input clock used to sample and synchronize the input signal.

`resetn` Active-low reset input. This is not the reset signal that is being debounced. Instead, this is the reset signal used to reset the `pb_debounceRESET` module itself. This is a remnant of the original `pb_debounce` module and is not used in `pb_debounceRESET` when instantiated inside the `PON` module. Therefore this signal is assigned to its inactive value, 1.

`pb_in` Input signal to be sampled and debouced. This is where the input reset signal is attached.

`pb_out` Stabilized output signal. This is a remnant of the original `pb_debounce` module and is not used in `pb_debounceRESET` when instantiated in the `PON` module. Therefore this output is ignored.

`pb_tick` Output pulse when the input has changed and is stable. When the input signal changes and the debouncer considers it stable, this active-high output changes to low for 1 cycle. This is used as the reset signal in the `PON` module.

3.12.3 Design Details

This module samples the `pb_in` input using `clk`. When the input changes from 0 to 1, a counter begins counting down from `{21{1´b1}}` to 0. If the signal remains stable by the time the counter reaches 0, the `pb_out` output changes from 0 to 1. At the same time, the `pb_tick` output, with a default value of 1, changes to 0 for a single cycle. When the input goes back to 0, the output follows. If the signal changes to 0 before the counter reaches 0, the counter is reset and the module waits for `pb_in` to change again.

The button used for the reset signal on the Nexys 3 board is an active-high button. The output is low when the button is not pressed, and the output is high when the button is pressed. In contrast, the button used for the reset signal on the Nexys 4 DDR board is an active-low button, where the output is high when not pressed, and low when pressed. The same `pb_debounceRESET` module is used for both, because the `pb_tick` output is used for the reset output instead of the `pb_out` output, and thus the behavior when a button is pressed is almost the same. The `pb_tick` output sends an active-low single-cycle pulse when the input changes from low to high and is stable. In the case of the Nexys 3 this happens when the button is pressed and held. In the case of the Nexys 4 DDR this happens when

a pressed button is released. It may be worthwhile in the future to modify the behavior of pb_debounceRESET to work for both types of buttons using the pb_out output instead.

3.13 ClockDiv

3.13.1 Description

This module was originally designed by Bigazzi to divide the 100MHz input clock down to 5MHz. This module has since been parameterized to divide a clock down by any amount and is currently used to divide the 100MHz clock to 2MHz and 500kHz.

3.13.2 Input/Output Ports and Parameters

CLK_IN Input clock to be divided.

RESETn Input reset.

CLK_OUT Divided output clock.

MAX_COUNT Parameter describing the number of cycles to count before inverting the output clock signal.

3.13.3 Design Details

This clock divider is implemented with a counter that increments from 0 to MAX_COUNT-1 and then wraps around. Every time the counter reaches MAX_COUNT-1 the CLK_OUT output inverts. The clock period of the output is:

$$T_{out} = 2 \times MAX\_COUNT \times T_{in}$$

This method of clock division is not recommended on FPGAs; however, this method is the only way to generate clocks less than 2.5MHz given that there are no FPGA primitives able to divide a 100MHz clock below 2.5MHz.

3.14 AHBDCD

3.14.1 Description

This module, adapted from the example provided in the ARM Cortex-M0 DesignStart kit, is used to determine which AHB slave is being accessed during an AHB transfer. It decodes HADDR[31:24] to generate a HSEL signal for each slave, as well as the MUL_SEL[3:0] signal sent to the AHBMUX module. This module supports up to 15 AHB slaves.

3.14.2 Input/Output Ports

`HADDR[31:24]` Address input. Only the 8 upper bits are necessary and the other bits are omitted.

`HSEL_S0` Slave select output for the first AHB slave. In the Single Chip Mote digital system this is the `AHBIMEM` module. The `AHBIMEM` module is connected to the AHB using two AHB interfaces, with one for fetching instructions and another for bootloading (chapter 5). `HSEL_S0` is for fetching instructions.

`HSEL_S1` Slave select output for the second AHB slave. In the Single Chip Mote digital system this is the AHBsub bus, via the `AHBLiteArbiter_V2` module.

`HSEL_S2` Slave select output for the third AHB slave. In the Single Chip Mote digital system this is the `DMA_V2` module.

`HSEL_S3` Slave select output for the fourth AHB slave. In the Single Chip Mote digital system this is the APB, via the `AHB2APB` module.

`HSEL_S4` Slave select output for the fifth AHB slave. In the Single Chip Mote digital system this is the `AHBIMEM` module. The `AHBIMEM` module is connected to the AHB using two AHB interfaces, with one for fetching instructions and another for bootloading (chapter 5). `HSEL_S4` is for bootloading.

`HSEL_S5` Slave select output for the sixth AHB slave. In the Single Chip Mote digital system this is the APB, via the `RFTIMER` module.

`HSEL_NOMAP` Slave select output indicating that the current address does not map to any AHB slaves. This output is omitted in the Single Chip Mote digital system.

`MUL_SEL[3:0]` Output to the `AHBMUX` module indicating which slave is selected. This is used to route the correct set of slave signals to the AHB master.

3.14.3 Design Details

This module only contains combinational logic to set the `HSEL` and `MUL_SEL` outputs using the 8 upper bits of `HADDR`. This is done using a casex statement with the prefixes defined in `REGISTERS.vh`. A casex statement is used in order to facilitate the use one single prefix for all APB slaves, by having the 4 upper bits match `4´b0101` and the 4 lower bits match `4´bxxxx`. As long as the 4 upper bits of all APB slave prefixes begin with `4´b0101`, then the `HSEL` signal for the APB bridge (`AHB2APB`) is correct.

Inside the casex statement, the 16-bit `dec` bus is assigned along with `MUX_SEL`. If the case is for the nth slave (where n=0 for the first slave, n=1 for the second), then `dec[n]` must be assigned to 1. All other bits in `dec` must be assigned 0. `MUX_SEL` must also be assigned to the binary value of n.

3.14.4 Adding Another AHB Slave

Adding a new AHB slave in this module requires the following steps:

1. Define an address prefix for the slave in `REGISTERS.vh`.
2. Add another `HSEL` output.
3. Add another case to the casex statement using the new address prefix. Make sure to set the `dec` and `MUX_SEL` signals correctly as described above.
4. Assign the new `HSEL` output to the corresponding bit in `dec`.
5. Connect the new `HSEL` output to the new slave/peripheral in the top module, `uCONTROLLER`.

3.15 AHBMUX

3.15.1 Description

This module, adapted from the example provided in the ARM Cortex-M0 DesignStart kit, is used to select the correct set of AHB slave signals (`HRDATA` and `HREADYOUT`) from the APB slave being accessed during the current transfer. The slave signals are chosen based on the `MUX_SEL` signal provided by `AHBDCD`. This module supports up to 15 slaves.

3.15.2 Input/Output Ports

`HCLK` Input clock.

`HRESETn` Input reset.

`MUX_SEL[3:0]` Input from `AHBDCD` indicating which AHB slave is selected for the current transfer.

`HRDATA_S0[31:0]` Read data input from the first AHB slave. In the Single Chip Mote digital system this is the `AHBIMEM` module. The `AHBIMEM` module is connected to the AHB using two AHB interfaces, with one for fetching instructions and another for bootloading (chapter 5). `HRDATA_S0` is for fetching instructions.

`HRDATA_S1[31:0]` Read data input from the second AHB slave. In the Single Chip Mote digital system this is the AHBsub bus, via the `AHBLiteArbiter_V2` module.

`HRDATA_S2[31:0]` Read data input from the third AHB slave. In the Single Chip Mote digital system this is the `DMA_V2` module.

`HRDATA_S3[31:0]` Read data input from the fourth AHB slave. In the Single Chip Mote digital system this is the APB, via the `AHB2APB` module.

HRDATA_S4[31:0] Read data input from the fifth AHB slave. In the Single Chip Mote digital system this is the AHBIMEM module. The AHBIMEM module is connected to the AHB using two AHB interfaces, with one for fetching instructions and another for bootloading (chapter 5). HRDATA_S4 is for bootloading.

HRDATA_S5[31:0] Read data input from the sixth AHB slave. In the Single Chip Mote digital system this is the RFTIMER module.

HRDATA_NOMAP[31:0] Read data input to be used when the current address does not map to any AHB slaves.

HREADYOUT_S0 Transfer finished input from the first AHB slave. In the Single Chip Mote digital system this is the AHBIMEM module. The AHBIMEM module is connected to the AHB using two AHB interfaces, with one for fetching instructions and another for bootloading (chapter 5). HREADYOUT_S0 is for fetching instructions.

HREADYOUT_S1 Transfer finished input from the second AHB slave. In the Single Chip Mote digital system this is the AHBsub bus, via the AHBLiteArbiter_V2 module.

HREADYOUT_S2 Transfer finished input from the third AHB slave. In the Single Chip Mote digital system this is the DMA_V2 module.

HREADYOUT_S3 Transfer finished input from the fourth AHB slave. In the Single Chip Mote digital system this is the APB, via the AHB2APB module.

HREADYOUT_S4 Transfer finished input from the fifth AHB slave. In the Single Chip Mote digital system this is the AHBIMEM module. The AHBIMEM module is connected to the AHB using two AHB interfaces, with one for fetching instructions and another for bootloading (chapter 5). HREADYOUT_S4 is for bootloading.

HREADYOUT_S5 Transfer finished input from the sixth AHB slave. In the Single Chip Mote digital system this is the RFTIMER module.

HREADYOUT_NOMAP Transfer finish input to be used when the current address does not map to any AHB slaves. This input must always be assigned to 1 to avoid indefinitely stalling the Cortex-M0 if it tries to access an unmapped address.

HREADY Multiplexed transfer finish output to the AHB master.

HRDATA[31:0] Multiplexed read data output to the AHB master.

3.15.3 Design Details

The address, HADDR, is valid during an address phase of a transfer, and therefore that the MUX_SEL is also only valid during the address phase. The MUX_SEL signal must be latched before moving on to the data phase. This is done by storing MUX_SEL into a register, APHASE_MUX_SEL, whenever HREADY is 1. APHASE_MUX_SEL is then used in a case statement to assign the correct slave signals to HRDATA and HREADY.

3.15.4 Adding Another AHB Slave

Adding a new AHB slave in this module requires the following steps:

1. Add this slave to `AHBDCD`. See section 3.14.4 for more details.
2. Add another `HRDATA` and `HREADYOUT` input.
3. Add another case to the case statement using the new value of `MUX_SEL` corresponding to the new slave. Assign `HRDATA` and `HREADY`.
4. Connect the new `HRDATA` and `HREADYOUT` inputs to the new slave/peripheral in the top module, `uCONTROLLER`.

3.16 AHBLiteArbiter_V2

3.16.1 Description

This module is an arbiter designed to allow for two masters (referred to as M0 and M1) to share a single AHB-Lite bus. This module is necessary in order to allow the Cortex-M0 and the DMA to both access the data memory and the radio controller via the AHBsub bus. Master M0 is the Cortex-M0, via the main AHB bus. Master M1 is the DMA. In this module, the slave refers to the AHBsub bus. The intended behavior of this arbiter is as follows:

- M0 does not experience any latency for any AHB transfers when M1 is not attempting to transfer at the same time. This means that, while the DMA is idle, the Cortex-M0 has to access both the data memory and radio controller as if it were on the main AHBbus, with no extra latency.
- M1 may experience at least 1 cycle of additional latency when accessing the bus, even if master M0 is idle. This is a consequence of the previous rule.
- When both M0 and M1 initiate a transfer at the same time, after a period of no transfers, M0 has priority. Initiating a transfer means the master asserts `HTRANS[1]` and `HSEL`; this also indicates the address phase of the transfer.
- When both M0 and M1 initiate a transfer at the same time, during a series of back-to-back transfers, the master that was not granted the last transfer is granted the next transfer.

Given that the AHBsub bus is primarily used by the Cortex-M0 to access data memory, it is important that the Cortex-M0 experience no latency when accessing the data memory under normal conditions. In contrast, the DMA only accesses the AHBsub bus when the Single Chip Mote is sending or receiving a radio packet, and the relatively slow data rate does not require that the DMA have high throughput or low latency. Therefore, it is acceptable to have additional latency for M1 and give priority during an initial collision to M0.

3.16.2 Input/Output Ports

`HCLK` Input clock.

`HRESETn` Input reset.

`HSEL_M0` Slave select input from master M0.

`HADDR_M0[31:0]` Address input from master M0.

`HTRANS_M0[1]` Transfer type input from master M0.

`HSIZE_M0[1:0]` Transfer size input from master M0.

`HWRITE_M0` Write select input from master M0.

`HWDATA_M0[31:0]` Write data input from master M0.

`HRDATA_M0[31:0]` Read data output to master M0.

`HREADY_M0` Transfer finished output to master M0.

`HSEL_M1` Slave select input from master M1.

`HADDR_M1[31:0]` Address input from master M1.

`HTRANS_M1[1]` Transfer type input from master M1.

`HSIZE_M1[1:0]` Transfer size input from master M1.

`HWRITE_M1` Write select input from master M1.

`HWDATA_M1[31:0]` Write data input from master M1.

`HRDATA_M1[31:0]` Read data output to master M1.

`HREADY_M1` Transfer finished output to master M1.

`HADDR_S[31:0]` Address output to slave.

`HTRANS_S[1]` Transfer type output to slave.

`HSIZE_S[1:0]` Transfer size output to slave.

`HWRITE_S` Write select output to slave.

`HWDATA_S[31:0]` Write data output to slave.

`HRDATA_S[31:0]` Read data input from slave.

`HREADYOUT_S` Transfer finished input from slave.

`error` Indicates that the arbiter reached an invalid state at some point. This signal is included for debugging purposes but is typically ignored. Currently, synthesis tools remove this signal during optimization.

3.16.3 Design Details

ARM provides in their Cortex-M System Design Kit [9] a bus matrix designed to allow for multiple masters to control a single AHB bus or even a single AHB slave. This module is both flexible in its use and allows for customization, and also benefits from being tested and proven to work. Unfortunately, the System Design Kit is not part of the DesignStart kit and must be purchased separately. There were multiple attempts prior to this work by visiting scholars Francesco Bigazzi and Lorenz Schmid (both working independently) to create a simplified but analogous arbiter design, all of which failed to function without error on real-time FPGA tests. Arbitration errors caused unpredictable behavior when both masters attempted to access peripherals on the AHBsub bus at the same time.

Design Issues

The main cause of difficulty in the design of this arbiter is the inability to completely describe its function in the form of a finite state machine or a series of basic rules/steps. This is attributed to the following complications:

- AHB transfers happen in two phases, the address and data phase. The arbiter must have a way to keep track of the phases for each master.
- AHB transfers can be pipelined, and each master could be in both the address and data phase at the same time.
- The AHB protocol expects that the slave latches address phase signals when necessary because of pipelined transfers. Given that there are two masters, this means that the each master must be tracked to ensure that address phase signals are latched properly when one master is stalled.

These complications imply that the arbiter must keep track of the following state:

- Which master's address phase signals are routed to the slave.
- Which master's dataphase signals are routed to the slave.
- Which master is connected to the slave's dataphase signals. This concerns the `HREADYOUT_S` signal in particular, as the stalled master must *not* see the `HREADYOUT_S` signal when the other master is using the bus.
- If either of the masters have address phase signals latched from when the other master was using the bus.

And the arbiter must make the following decisions depending on the state:

- Which master's address phase signals to route to the slave during the next cycle. This includes address phase signals that were latched from a stalled master. This is where the arbitration takes place; the result depends on which master is currently using the bus.
- Which master's data phase signals to route to the slave during the next cycle. This depends on whether or not the transfer is complete (`HREADYOUT_S` is asserted), and which address phase signals are currently routed to the slave.

- Whether or not any address phase signals need to be latched because one master must be stalled. Address phase signals only need to be latched when the master is not waiting for its own data phase to finish. When the master is waiting for a data phase to complete, it is sufficient to keep `HREADY` low. This stalls the master causing it to continue to hold its address phase signals. However, if the master is not waiting (from its point of view it is initiating its first transfer after being idle), the master assumes the slave latches the address phase signals and only holds the valid signals for 1 cycle. This behavior is a consequence of pipelined transfers in the AHB protocol.

The amount of state that must be tracked for both masters and the slave makes it difficult to create a concise description of the arbiter's behavior. It was eventually determined that the best way to design this module was to enumerate every possible state and determine the actions and next state in a large table. From there, the address phase and data phase signals could be routed properly and the combinational logic to describe the table could be written in Verilog. While this solution is far from the best possible design practice, the current `AHBLiteArbiter_V2` implementation continuously proves to work properly when tested in real-time on an FPGA.

State Variables and Inputs

The state variables used to route signals and determine the next state in this module are:

`current_address_phase[1:0]` This state describes which set of address phase signals are currently routed to the slave. Possible values are `APHASE_PASS_M0`, `APHASE_LATCH_M0`, and `APHASE_LATCH_M1`.

`current_data_phase[1:0]` This state describes which set of data phase signals are currently routed to the slave and back. Possible values are `DPHASE_NONE`, `DPHASE_M0`, and `DPHASE_M1`.

`inputs_latched_M0` This state indicates whether or not there are currently any latched address signals from M0. This can happen when M0 must be stalled while the bus is in use. This state is needed when determining the next address phase.

`inputs_latched_M1` This state indicates whether or not there are currently any latched address signals from M1. This can happen when M1 must be stalled while the bus is in use. This also happens when M1 uses the bus after an idle period since M0 continues to have priority when idle, and its address phase signals are always connected to the slave when idle. This state is needed when determining the next address phase.

In addition to the state variables, the following inputs are needed to determine the next state in this module:

`req_M0` This signal is the bitwise AND of the `HSEL_M0` and `HTRANS_M0[1]` inputs, used to indicate that M0 is requesting use of the bus. This combination of inputs is needed when determining the next address phase.

`req_M1` This signal is the bitwise AND of the `HSEL_M1` and `HTRANS_M1[1]` inputs, used to indicate that M1 is requesting use of the bus. This combination of inputs is needed when determining the next address phase.

Combinational Logic Based on State Variables and Inputs

The `current_address_phase` state is used to route the address phase signals from one of the masters to the slave, and the `current_data_phase` state is used to route the dataphase signals between one master and the slave.

`APHASE_PASS_M0` means that address phase signals are passed directly from M0 to the slave, with no registers in between. `APHASE_LATCH_M0` means that address phase signals latched from M0 during a previous cycle are routed to the slave. `APHASE_LATCH_M1` means that address phase signals latched from M1 during a previous cycle are routed to the slave. The default value is `APHASE_PASS_M0`.

`DPHASE_NONE` means that all data phase signals are 0 because neither master is waiting for a slave. `DPHASE_M0` means that `HWDATA_M0` is routed to the slave, `HREADYOUT_S` is routed to `HREADY_M0`, and `HREADY_M1` is 0. `DPHASE_M1` means that `HWDATA_M1` is routed to the slave, `HREADYOUT_S` is routed to `HREADY_M1`. and `HREADY_M0` is 0.

In addition to choosing the next state, new address phase signals may need to be latched or cleared. This is indicated by the `latch_M0`, `latch_M1`, `clr_M0`, and `clr_M1` signals assigned within the next state logic.

State Transition and Action Table

Appendix A.1 contains the table listing the inputs, state variables, next state, and actions to be taken, for every combination of state and input. This table was used to implement the next state combinational logic in `AHBLiteArbiter_V2`. Note that some states are labeled as "invalid state"; these states should be impossible to reach. However, the code is designed such that if one of those states were detected, the `error` output is set to 1 and stays that way until the system is reset.

3.17 AHBDCDsub

3.17.1 Description

This module is the same as the `AHBDCD` module described in section 3.14. The main difference is that this module is designed to be used for the AHBsub bus and has two slaves: the `AHBDMEM` module and the `RFcontroller` module. Also, the width of the `MUX_SEL` signal is reduced such that this module now only supports three slaves.

3.17.2 Input/Output Ports

`HADDR[31:24]` Address input. Only the 8 upper bits are necessary and the other bits are omitted.

`HSEL_S0` Slave select output for the first AHB slave. In the Single Chip Mote digital system this is the `AHBDMEM` module.

`HSEL_S1` Slave select output for the second AHB slave. In the Single Chip Mote digital system this is the `RFcontroller` module.

`HSEL_NOMAP` Slave select output indicating that the current address does not map to any AHB slaves. This output is omitted in the Single Chip Mote digital system.

`MUL_SEL[1:0]` Output to the `AHBMUX` module indicating which slave is selected. This is used to route the correct set of slave signals to the AHB master.

3.17.3 Design Details

See section 3.14.

3.17.4 Adding Another AHB Slave

See section 3.14.

3.18 AHBMUXsub

3.18.1 Description

This module is the same as the `AHBMUX` module described in section 3.15. The main difference is that this module is designed to be used for the AHBsub bus and has two slaves: the `AHBDMEM` module and the `RFcontroller` module. Also, the width of the `MUX_SEL` signal is reduced such that this module now only supports three slaves.

3.18.2 Input/Output Ports

`HCLK` Input clock.

`HRESETn` Input reset.

`MUX_SEL[3:0]` Input from `AHBDCD` indicating which AHB slave is selected for the current transfer.

`HRDATA_S0[31:0]` Read data input from the first AHB slave. In the Single Chip Mote digital system this is the `AHBDMEM` module.

`HRDATA_S1[31:0]` Read data input from the second AHB slave. In the Single Chip Mote digital system this is the `RFcontroller` module.

`HRDATA_NOMAP[31:0]` Read data input to be used when the current address does not map to any AHB slaves.

`HREADYOUT_S0` Transfer finished input from the first AHB slave. In the Single Chip Mote digital system this is the `AHBDMEM` module.

`HREADYOUT_S1` Transfer finished input from the second AHB slave. In the Single Chip Mote digital system this is the `RFcontroller` module.

`HREADYOUT_NOMAP` Transfer finish input to be used when the current address does not map to any AHB slaves. This input must always be assigned to 1 to avoid indefinitely stalling the Cortex-M0 if it tries to access an unmapped address.

`HREADY` Multiplexed transfer finish output to the AHB master.

`HRDATA[31:0]` Multiplexed read data output to the AHB master.

3.18.3 Design Details

See section 3.15.

3.18.4 Adding Another AHB Slave

See section 3.15.

3.19 AHBIMEM

3.19.1 Description

This module provides access to the instruction memory for the Cortex-M0. The instruction memory is composed of a 16kB ROM and a 64kB SRAM (referred to from now on as the instruction RAM). Only one of the two memories are in use at any particular time. Instructions are fetched from the ROM when the Single Chip Mote digital system initially turns on, or after a hard reset. The code in the ROM is designed to load the main software code into the RAM, and then initiate a soft reset. After the soft reset, instructions are fetched from the RAM. The main software code is written into the RAM directly through an external 3 Wire Bus interface, or via the AHB bus from the Cortex-M0. The Cortex-M0 has the option to listen for instructions via UART, over the radio, or even through an optical interface (this has not been designed yet). For more information on bootloading, see chapter 5.

This module has two AHB slave interfaces. The first slave interface is used to read from the instruction memory. The `HWRITE` signal is ignored, as instruction memory is considered read-only, even when the SRAM is in use. All instruction fetches are assumed to be the size of a word (32 bits) and word-aligned, and therefore the `HSIZE` signal from the AHB is omitted. The second slave interface is used to load data into the instruction RAM. This interface has one special address allocated for a configuration register. All other addresses correspond to an address in the instruction RAM. AHB writes overwrite data in either the configuration register or the instruction RAM. AHB reads return the contents of a special status register, regardless of the address used. All writes are assumed to be the size of a word (32 bits) and word-aligned, and therefore the `HSIZE` signal from the AHB is omitted.

3.19.2 Input/Output Ports and Parameters

`RESETn` Hard reset input. This reset is triggered externally from the FPGA board.

`HRESETn` Soft reset input. This reset is triggered either externally from the FPGA board or internally via a reset request from the Cortex-M0.

HSEL_IMEM Slave select input for the instruction memory AHB interface.

HSEL_BOOTLOAD Slave select input for the bootloading AHB interface.

HREADY Transfer finished input. This input indicates that the previous transfer on the bus has finished and that address phase signals must be latched.

HADDR[31:0] Address input.

HTRANS[1] Transfer type input.

HWRITE Write select input.

HWDATA[31:0] Write data input.

HREADYOUT_IMEM Transfer finished output from the instruction memory AHB interface.

HRDATA_IMEM[31:0] Read data output from the instruction memory AHB interface.

HREADYOUT_BOOTLOAD Transfer finished output from the bootloading AHB interface.

HRDATA_BOOTLOAD[31:0] Read data output from the bootloading AHB interface.

clk_3wb Clock input for the 3 Wire Bus.

data_3wb Data input for the 3 Wire Bus.

latch_3wb Latch input for the 3 Wire Bus.

clk_3wb_en Clock enable output to disable clk_3wb when not in use.

ROM_ADDR_WIDTH Parameter describing the size of the instruction ROM. The number of address bits needed for the word-addressable instruction ROM is equal to this parameter, and the depth of the instruction ROM is $2^{ROM\_ADDR\_WIDTH}$. Note that the addresses in HADDR are byte, not word, addresses.

RAM_ADDR_WIDTH Parameter describing the size of the instruction RAM. The number of address bits needed for the word-addressable instruction RAM is equal to this parameter, and the depth of the instruction RAM is $2^{RAM\_ADDR\_WIDTH}$. Note that the addresses in HADDR are byte, not word, addresses.

3.19.3 Design Details

This module uses the imem_mode register to determine the source of instruction data, and connects that source to the AHB interface for fetching instructions. Instruction data can come from either the ROM or the RAM, and therefore, the two possible states for the imem_mode register are IMEM_MODE_ROM and IMEM_MODE_RAM. The imem_mode register is only updated on a hard or soft reset, to ensure that the source of instruction data does not change while the Single Chip Mote digital system is running. On a hard reset, imem_mode is set by default to IMEM_MODE_ROM. On a soft reset, imem_mode is set to the value stored in the next_imem_mode register.

imem_mode	Encoding
IMEM_MODE_ROM	1'b0
IMEM_MODE_RAM	1'b1

Figure 3.9: Encodings for the two imem_mode states

boot_mode	Encoding
BOOT_MODE_NONE	2'b00
BOOT_MODE_3WB	2'b10
BOOT_MODE_AHB	2'b11

Figure 3.10: Encodings for the three boot_mode states

This register is set by the Cortex-M0 via the bootloading AHB interface to be either IMEM_MODE_ROM or IMEM_MODE_RAM. Figure 3.9 contains the encoded values for IMEM_MODE_RAM and IMEM_MODE_ROM.

This module uses the boot_mode register to determine from which source the instruction RAM is written. The three possible states for this register are BOOT_MODE_NONE, BOOT_MODE_3WB, and BOOT_MODE_AHB. BOOT_MODE_NONE means that the instruction RAM cannot be written. BOOT_MODE_3WB means that the instruction RAM is written externally via the 3 Wire Bus. BOOT_MODE_AHB means that the instruction RAM is written by the Cortex-M0 via the bootloading AHB interface. This state register is configured by the Cortex-M0 via the bootloading AHB interface. Figure 3.10 contains the encoded values for all three boot modes.

When the boot_mode register is set to BOOT_MODE_3WB, the AHBIMEM module enables the 3 Wire Bus clock input (using clk_3wb_en), and waits for 64kB of data to be written into the instruction RAM. This is accomplished using a counter, doubling as the write address to the instruction RAM. This counter is initialized to 0 on a soft reset, and increments as each 32-bit word is written into the RAM. Once this counter has reached its maximum value, indicating that 64kB has been written to the RAM, the boot_3wb_done signal is asserted, and all other writes via the 3 Wire Bus are disabled. The 3 Wire Bus clock is also disabled. For more information on the 3 Wire Bus protocol, see section 5.4.

When the boot_mode register is set to BOOT_MODE_AHB, the AHBIMEM module connects the write port of the instruction RAM to the bootloading AHB interface. This allows for the Cortex-M0 to write directly into the instruction RAM (although it cannot read what is in the instruction RAM). The two AHB interfaces correspond to two separate AHB slaves with their own address prefixes. The prefix for the instruction fetching interface is 0x00 and the prefix for the bootloading interface is 0x01. Any writes to the RAM via the bootloading interface result in a write to the corresponding address in the RAM where the 0x01 prefix is replaced with 0x00. For example, writing an instruction to address 0x0100CAD0 translates to address 0x0000CAD0 in the actual instruction memory. Since the instruction RAM is only 64kB, any writes to addresses beyond the first 64kB of instruction memory result in overwriting instruction data within the first 64kB.

The bootloading AHB interface also has a dedicated write-only configuration register used to change next_imem_mode and boot_mode. Any AHB read transfer, regardless of the address, from the bootloading AHB interface result in reading the status register. This status register returns the values of imem_mode, next_imem_mode,

boot_mode, and boot_3wb_done. See the next section for details on these registers.

3.19.4 Register Interface

AHB Interface for Instruction Fetching

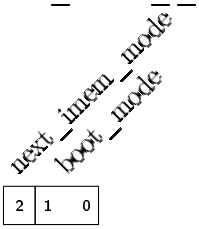
This AHB interface has no memory-mapped registers. All reads return instruction data at the corresponding address in either the ROM or the RAM, and writes have no effect. Valid addresses are in the range of 0x00000000-0x0000FFFF for a memory size of 64kB.

AHB Interface for Bootloading

This AHB interface is used to write to the instruction RAM, and has one read-only memory-mapped register and one write-only memory-mappped register. The first 64kB of addresses, 0x01000000-0x0100FFFF, are allocated to the write-only instruction RAM. The address 0x01F00000 corresponds to the write-only BOOTLOADER_REG__CFG register. The fields of this register are boot_mode and next_imem_mode. Reading from any register with the 0x01 prefix returns the BOOTLOADER_REG__STATUS register. The fields of this register contain imem_mode, next_imem_mode, boot_mode, and boot_3wb_done.

Register Descriptions

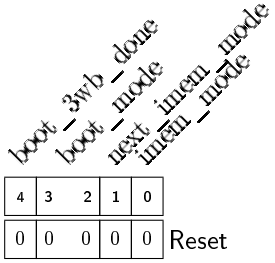
Register 3.1: BOOTLOADER_REG__CFG (0x010F00000)



boot_mode (Write-only) Bootloader data source. 00 = 01 = NONE, 10 = 3WB, and 11 = AHB.

next_imem_mode (Write-only) Instruction data source. 0 = ROM and 1 = RAM.

Register 3.2: BOOTLOADER_REG__STATUS (0x010F00004)



imem_mode Instruction data source. 0 = ROM and 1 = RAM.

next_imem_mode Instruction data source after next soft reset. 0 = ROM and 1 = RAM.

boot_mode Bootloader data source. 00 = 01 = NONE, 10 = 3WB, and 11 = AHB.

boot_3wb_done Booting through the 3 Wire Bus is finished. 0 = not done and 1 = done.

3.20 instruction_ROM

3.20.1 Description

This module is a 16kB read-only memory constructed out of FPGA primitives using CORE Generator. This ROM has a width of 32 bits and a depth of 4096, a read enable port, and a read latency of 1 cycle. This ROM is initialized with a COE file containing the compiled binary C code for the bootloading firmware. See chapter 5 for more information on bootloading.

3.20.2 Input/Output Ports

`clka` Input clock for the read port.

`ena` Enable input for the read port.

`addra[11:0]` Address input for the read port.

`douta[31:0]` Data output for the read port. This data is valid 1 cycle after `ena` is asserted.

3.20.3 Design Details

The ROM was designed using the Block Memory Generator in CORE Generator. Any changes to the parameters of this module, such as the width or depth, requires opening the `instruction_ROM.xco` file in CORE Generator, changing the parameters, and then regenerating the core. To open this design in CORE Generator, open Project Navigator and double-click the `instruction_ROM` module in the Design Hierarchy panel.

3.20.4 Initialization

Initializing the ROM requires a COE file containing data with the same width as the ROM, and a depth less than or equal to the depth of the ROM. In the

case of the Single Chip Mote digital system, the COE file used to initialize the instruction ROM contains the compiled C binary code for the bootloading firmware. This firmware is first compiled into a binary (.bin) file using Keil. From there the Bin2coe tool (2.5.2) is used to convert the binary file into a COE file. And finally, the `instruction_ROM.xco` file must be opened in CORE Generator, wherein there is an option to specify a COE file to initialize the instruction ROM. Then the core must be generated again after entering the proper specifications. Any changes to the COE file require regeneration of the core.

3.21 instruction_RAM

3.21.1 Description

This module is a 64kB simple dual port SRAM constructed out of FPGA primitives using CORE Generator. This RAM has a width of 32 bits and a depth of 16384. The first port is for writing and the second port is for reading. The read port has a read enable input, with a read latency of 1 cycle. The write port contains separate write enable inputs for each byte 8-bit byte in the wordline (4 write enable inputs in total), and writes take effect after a single cycle.

3.21.2 Input/Output Ports

`clka` Input clock for the write port.

`ena` Enable input for the write port. This must be asserted to write, even if `wea` is asserted.

`wea[3:0]` Write enable input for each byte for the write port.

`addra[13:0]` Address input for the write port.

`dina[31:0]` Data input for the write port.

`clkb` Input clock for the read port.

`enb` Enable input for the read port.

`addrb[13:0]` Address input for the read port.

`doutb[31:0]` Data output for the read port. This data is valid 1 cycle after `ena` is asserted.

3.21.3 Design Details

The RAM was designed using the Block Memory Generator in CORE Generator. Any changes to the parameters of this module, such as the width or depth, requires opening the `instruction_RAM.xco` file in CORE Generator, changing the parameters, and then regenerating the core. To open this design in CORE Generator, open Project Navigator and double-click the `instruction_RAM` module in the Design Hierarchy panel.

3.22 AHBDMEM

3.22.1 Description

This module provides access to the data memory for the Cortex-M0. The data memory is composed of a 64kB SRAM with a width of 32 bits and a depth of 16384. This module supports byte, half-word, and word sized writes using the `HSIZE` input from the AHB. This module is based on the `AHB2MEM` module provided in the ARM Cortex-M0 DesignStart kit.

3.22.2 Input/Output Ports and Parameters

`HCLK` Input clock.

`HRESETn` Input reset.

`HSEL` Slave select input.

`HREADY` Transfer finished input. This input indicates that the previous transfer on the bus has finished and that address phase signals must be latched.

`HADDR[31:0]` Address input.

`HTRANS[1]` Transfer type input.

`HWRITE` Write select input.

`HSIZE[1:0]` Transfer size input. `HSIZE[2]` is omitted because this module does not support writes larger than one word.

`HWDATA[31:0]` Write data input.

`HREADYOUT` Transfer finished output.

`HRDATA[31:0]` Read data output.

`MEMWIDTH` Parameter describing the size of the data memory. The number of address bits needed for the word-addressable data memory is $MEMWIDTH-2$, and the depth of the instruction RAM is $2^{MEMWIDTH-2}$. Note that the addresses in `HADDR` are byte, not word, addresses.

3.22.3 Design Details

The original `AHB2MEM` module provided by ARM used an inferred SRAM with an asynchronous read. However, the block RAMs on Xilinx FPGAs only support synchronous reads, and asynchronous SRAM blocks cannot be used on the ASIC version of the Single Chip Mote digital system. Therefore, the code was modified to replace the inferred RAM with the instantiation of the `dmem_ram` module, created in CORE Generator. All that remains of the original code is the logic used to generate the write enable signal for each byte in the wordline. These write enable signals are created by combining `HADDR[1:0]` and `HSIZE[1:0]` (`HSIZE[2]` is omitted because this module does not support writes larger than one word).

3.22.4 Register Interface

This AHB interface has no memory-mapped registers. All reads return the data at the corresponding address in the RAM, and writes overwrite the data at the corresponding address in the RAM. Valid addresses are in the range of `0x20000000-0x2000FFFF` for a memory size of 64kB.

3.23 dmem_ram

3.23.1 Description

This module is a 64kB simple dual port SRAM constructed out of FPGA primitives using CORE Generator. This RAM has a width of 32 bits and a depth of 16384. The first port is for writing and the second port is for reading. The read port has a read enable input, with a read latency of 1 cycle. The write port contains separate write enable inputs for each byte 8-bit byte in the wordline (4 write enable inputs in total), and writes take effect after a single cycle.

3.23.2 Input/Output Ports

`clka` Input clock for the write port.

`ena` Enable input for the write port. This must be asserted to write, even if `wea` is asserted.

`wea[3:0]` Write enable input for each byte for the write port.

`addra[13:0]` Address input for the write port.

`dina[31:0]` Data input for the write port.

`clkb` Input clock for the read port.

`enb` Enable input for the read port.

`addrb[13:0]` Address input for the read port.

`doutb[31:0]` Data output for the read port. This data is valid 1 cycle after `ena` is asserted.

3.23.3 Design Details

The RAM was designed using the Block Memory Generator in CORE Generator. Any changes to the parameters of this module, such as the width or depth, requires opening the `dmem_ram.xco` file in CORE Generator, changing the parameters, and then regenerating the core. To open this design in CORE Generator, open Project Navigator and double-click the `dmem_ram` module in the Design Hierarchy panel.

3.24 DMA_V2

3.24.1 Description

This module is the interface between the RFcontroller module and the data memory in the AHBDMEM module. This module copies packet data from the data memory to the RFcontroller for packet transmission, and also copies received packet data from the RFcontroller to the data memory. This module operates independently without any intervention from the Cortex-M0, allowing for packets to be autonomously sent and received when the Cortex-M0 is sleeping.

A more complicated version of this module, DMA, was originally created by Bigazzi to handle the transfer of both packet data and sampled data from the ADC. This module was designed to interface with the original versions of the ADC and radio controller written by Bigazzi. These modules are no longer in use since the analog and radio circuits have been updated. The DMA_V2 module is a slimmed-down version of the original DMA module, designed to meet the minimum needs of the current Single Chip Mote project and interface with the new radio circuit. In the future, this module should be re-designed to include control for the APBADC_V2 module, as well as any other features that may be useful for application development on the Single Chip Mote.

3.24.2 Input/Output Ports

HCLK Input clock.

HRESETn Input reset.

HSEL Slave select input.

HTRANS[1] Transfer type input.

HWRITE Write select input.

HADDR[31:0] Address input.

HWDATA[31:0] Write data input.

HRDATA[31:0] Read data output.

HREADY Transfer finished input. This input indicates that the previous transfer on the bus has finished and that address phase signals must be latched.

HREADYOUT Transfer finished output.

oHSEL Slave select output for the AHB master interface.

oHSIZE[1:0] Transfer size output for the AHB master interface.

oHADDR[31:0] Address output for the AHB master interface.

oHWDATA[31:0] Write data output for the AHB master interface.

oHRDATA[31:0] Read data input for the AHB master interface.

oHTRANS[1] Transfer type output for the AHB master interface.

oHWRITE Write select output for the AHB master interface.

oHREADY Transfer finished input for the AHB master interface. Used to stall the AHB master when the slave is not finished with the transfer.

rf_data_req Input from the RFcontroller module requesting packet data to be fetched from the data memory.

rf_data_store Input from the RFcontroller module requesting packet data to be stored to the data memory.

3.24.3 Design Details

Mode Select FSM

This module operates in three separate modes: IDLE, RF_DATA_STORE, and RF_DATA-_GET. In IDLE mode, the module waits for requests from the RFcontroller module via the rf_data_store and rf_data_req inputs. If rf_data_store is asserted, the mode changes to RF_DATA_STORE. If rf_data_req is asserted, the mode changes to RF_DATA_GET. These two modes trigger a series of AHB transfers to/from the RFcontroller and AHBDMEM modules, using the AHB master interface of this module connected to the AHBsub bus. Once the transfers are complete (as indicated by the rf_store_done and rf_req_done signals), the mode changes back to IDLE mode.

The RF_DATA_STORE mode copies received packet data from the RFcontroller module and writes it to the data memory. This involves two AHB transfers. The first is to read the packet data from the RFCONTROLLER_REG__RX_DATA_DMA register on the RFcontroller module. The second is to write the packet data to the AHBDMEM module using the address stored in the RFRxAddr register. The RFRxAddr register in the code corresponds to the DMA_REG__RF_RX_ADDR memory-mapped register, set by the Cortex-M0. This address is incremented by 4 by the DMA_V2 module after every write.

The RF_DATA_GET mode fetches packet data from the data memory and writes it to the RFcontroller module. This involves three AHB transfers. The first is to read the address of the data to fetch from the RFCONTROLLER_REG__TX_DATA_ADDR_DMA register on the RFcontroller module. The data from this register is stored on the RFTxAddr register. The second transfer reads the data from the AHBDMEM module using the address stored in the RFTxAddr register. The data fetched from the memory is stored on the RFTxData register. The third transfer writes the data stored on the RFTxData register to the RFCONTROLLER_REG__TX_DATA_DMA register on the RFcontroller module.

The mode select state machine controls the AHB master interface using the num_aphase, addr1, addr2, addr3, addr4, and trtype signals. The num_aphase signal indicates how many AHB transfers (address phases) are needed for that particular mode (num_aphase == 0 corresponds to 1 transfer, num_aphase == 3 corresponds to 4 transfers). The RF_DATA_STORE mode requires two AHB transfers. The RF_DATA_GET mode requires three AHB transfers; however, four are used as one transfer is a dummy inserted after the first transfer to allow time for it to complete and update the RFTxAddr register. The addr1 through addr4 signals indicate the

addresses for transfers 1 through 4. The trtype signal indicates the transfer direction (read or write) for each transfer. For example, trtype[0] == 0 indicates that the first transfer is a read, and trtype[3] == 1 indicates that the fourth transfer is a write.

AHB Master Interface and FSM

The main purpose of this module is carried out using the AHB master interface. This interface allows the DMA_V2 module to act as an AHB master for the RFcontroller and AHBDMEM modules. A simple four phase state machine is used in both RF_DATA-_STORE and RF_DATA_GET modes to handle all AHB transfers.

During the IDLEPHASE state, the AHB master state machine waits for the start signal from the mode state machine. This signal indicates that the mode has changed to RF_DATA_STORE or RF_DATA_GET and initiates the series of AHB transfers. This signal is also used to clear the aphase_count register, which keeps track of the number of completed address phases in the series of transfer. The state transitions to ADDRPHASE for the first address phase of the AHB transfers.

During the ADDRPHASE state, used for the first AHB transfer, the aphase_count register is 0. The oHADDR output is connected to addr1, and the oHWRITE output is connected to trtype[0]. The aphase_count register is incremented at the end of the cycle. Since this is the first address phase of a series of back-to-back transfers, the AHB protocol indicates that the AHB master immediately transitions to the first data phase. There are two states in the AHB master state machine used for data phases. The first is ADDRDATAPHASE, used when the next phase is both a data phase for the current transfer and an address phase for the next transfer. The second is DATAPHASE, used when the next phase is only a data phase and there are no more transfers remaining. The aphase_count signal is compared to num_aphase to see if there are any more transfers left and choose the next state.

During the ADDRDATAPHASE state, used for pipelined transfers where the address and data phases overlap, the oHADDR and oHWRITE outputs are connected to one of the addr and trtype signals according to the value of aphase_count. The oHWDATA output is connected to either RFTxData or RFRxData, depending on the mode. Since this phase is a data phase, the AHB protocol indicates that all signals remain valid until the transfer is complete. Therefore, the AHB master state does not change until oHREADY is asserted. At that time the state transitions to either ADDRDATAPHASE or DATAPHASE, depending on the value of aphase_count, and aphase_count is incremented.

During the DATAPHASE state, used for the last data phase in a series of AHB transfers, the oHWDATA output is connected to either RFTxData or RFRxData, depending on the mode. Since this phase is a data phase, the AHB protocol indicates that all signals remain valid until the transfer is complete. Therefore, the AHB master state does not change until oHREADY is asserted. At that time the state transitions to IDLEPHASE, the rf_store_done or rf_req_done signal is asserted, and the aphase_count register is cleared.

RF Data and Address State Registers

The RFTxAddr register holds the address of packet data to be fetched from data memory for transmission. This register is updated by the DMA_V2 module during

the RF_DATA_GET mode before every fetch.

The RFTxData register holds the packet data fetched from data memory for a transmission. This register is updated by the DMA_V2 module during the RF_DATA-_GET mode after every fetch.

The RFRxAddr register holds the address where the next word of received packet data is stored in the data memory. This register must first be written by the Cortex-M0 (using the DMA_REG__RF_RX_ADDR memory-mapped register) to the beginning of a section of memory dedicated to received packet data. This register is then updated by the DMA_V2 module during the RF_DATA_STORE mode after each memory write.

The RFRxData register holds the received packet data to be stored in the data memory. This register is updated by the DMA_V2 module during the RF_DATA_STORE mode before every memory write.

Registers updated after AHB transfers are controlled using individual write enable signal. These write enable signals depend on the mode (RF_DATA_STORE or RF_DATA_GET), the phase of the AHB master interface (the phase must be either ADDRDATAPHASE or DATAPHASE), the number of transfers indicated by aphase_count (since RFTxAddr is updated after the first address phase, during the first data phase, and RFTxData is updated after the third address phase, during the third data phase), and if the slave has completed the transfer (indicated by the oHREADY input). The inputs to the RFTxAddr, RFTxData, and RFRxData registers are connected to the oHRDATA input. The input to the RFRxAddr is connected to either the HWDATA input (for writes from the Cortex-M0) or an adder (to increment its value by 4).

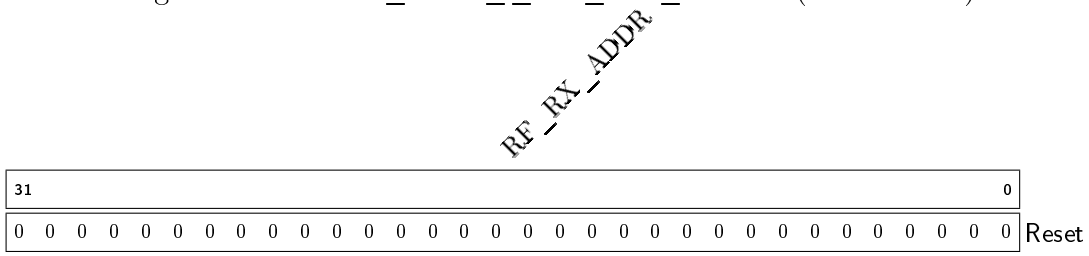
3.24.4 Register Interface

Receive Data Address Register

The DMA_REG__RF_RX_ADDR register holds the 32-bit address where the next 32-bit word of received packet data is stored in the data memory. The address in this register must be word-aligned. In order to store the largest possible packet (127 bytes) with three additional bytes for the packet length and CRC, this address must refer to a location within a continuous, word-aligned section of data memory designated in the software, with a length of at least 130 bytes. This module automatically increments this address after every write, and continues to write to every subsequent address unless this register is set to another value by the Cortex-M0. To avoid overwriting other sections of memory, this register must be updated with the first address of the allocated section of memory before listening for an incoming packet.

Register Descriptions

Register 3.3: DMA_REG__RF_RX_ADDR (0x41000014)



RF_RX_ADDR Address in data memory where received packet data is stored.

3.25 RFcontroller

3.25.1 Description

This module is the interface between the Single Chip Mote digital system and the radio circuit. This module is responsible for both transmitting (TX) and receiving (RX) packets using the IEEE 802.15.4 standard [15]. This requires several sub-blocks including: the mode select state machine to choose between TX and RX, a TX state machine to assemble packets, an RX state machine to store received packets in memory, a spreader to convert packet data into single bits for transmission, a correlator to detect received packets, a despreader to convert received bits into packet data, and two FIFOs to hold TX/RX packet data before transmission/storage. The `RFcontroller` module has memory-mapped registers to control the radio by triggering certain events. The `RFTIMER` module also sends triggers to the radio (see section 3.35 for more details). The `RFcontroller` module also generates interrupts to the Cortex-M0 and the `RFTIMER` module, used to indicate when the radio has finished executing particular tasks. This module relies on the `DMA_V2` module to fetch TX packet data from memory, and store RX packet data to memory (see section 3.24 for more details).

3.25.2 Input/Output Ports and Parameters

`HCLK` Input system clock.

`HRESETn` Input reset.

`HSEL` Slave select input.

`HWRITE` Write select input.

`HTRANS[1]` Transfer type input.

`HADDR[31:0]` Address input.

`HWDATA[31:0]` Write data input.

`HRDATA[31:0]` Read data output.

HREADY Transfer finished input. This input indicates that the previous transfer on the bus has finished and that address phase signals must be latched.

HREADYOUT Transfer finished output.

tx_clk Input clock for transmitting data to the radio circuit.

tx_dout Output data sent to the radio circuit for transmission.

rx_clk Input clock aligned to data received by radio circuit.

rx_din Input data received from radio circuit transmission.

rx_clk_en Clock enable output for rx_clk. This is needed because rx_clk comes from a shared input on the FPGA and must be disabled when it is not needed to prevent any unexpected behavior.

rf_data_req Output to the DMA_V2 module requesting data from the data memory for packet transmission.

rf_data_store Output to the DMA_V2 module requesting that received packet data is written to the data memory.

tx_load_rftimer Input from the RFTIMER module for the TX_LOAD trigger.

tx_send_rftimer Input from the RFTIMER module for the TX_SEND trigger.

rx_start_rftimer Input from the RFTIMER module for the RX_START trigger.

rx_stop_rftimer Input from the RFTIMER module for the RX_STOP trigger.

tx_load_done_pulse Output to the RFTIMER module for the TX_LOAD_DONE interrupt.

tx_sfd_done_pulse Output to the RFTIMER module for the TX_SFD_DONE interrupt.

tx_send_done_pulse Output to the RFTIMER module for the TX_SEND_DONE interrupt.

rx_sfd_done_pulse Output to the RFTIMER module for the RX_SFD_DONE interrupt.

rx_done_pulse Output to the RFTIMER module for the RX_DONE interrupt.

rf_irq Interrupt output to the Cortex-M0.

CORRELATOR_THRESHOLD Parameter used when scanning the rx_in data stream for the beginning of a packet. This parameter is passed into the corr_despreader submodule.

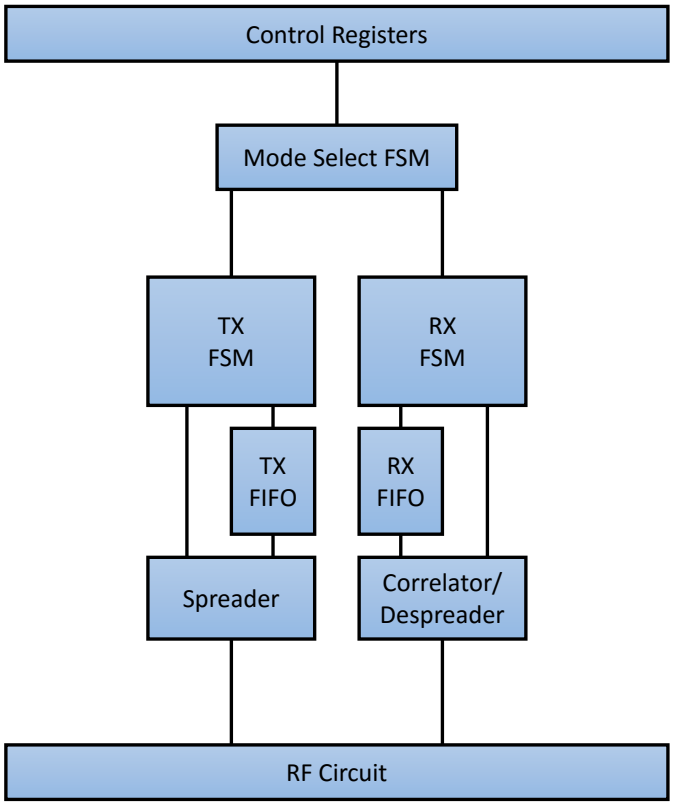


Figure 3.11: Block diagram overview of the `RFcontroller` module and its connections to the radio circuit

3.25.3 Design Details

Figure 3.11 contains a block diagram showing the main parts of the `RFcontroller` module and their connections to each other. The three finite state machines are implemented in the `RFcontroller` module. The TX FIFO is implemented in the `tx_fifo2` module (section 3.26), and the RX FIFO is implemented in the `rx_fifo` module (section 3.27). The spreader is in the `spreader` module (section 3.28) and the correlator/despreader is in the `corr_despreader` module (section 3.30). The RF circuit is the radio circuit on the Single Chip Mote.

IEEE 802.15.4 Packets

The radio circuit and `RFcontroller` module on the Single Chip Mote are designed to transmit and receive packets compliant with the IEEE 802.15.4 standard [15]. In particular, the radio circuit and `RFcontroller` module are responsible for implementing the physical (PHY) layer of the standard. This standard defines several different PHYs which use different frequency bands and modulation schemes. The

Single Chip Mote implements the O-QPSK PHY, defined in section 10 of the standard.

The binary-encoded data in the packet is separated into groups of 4-bit symbols, where 2 symbols is equivalent to a byte. If the data is arranged in bytes then the least significant bits in a byte `[3:0]` make up the first symbol, and the most significant bits `7:4` make up the second symbol.

Each packet begins with a preamble, beginning with 8 copies of the `4'b0000` symbol. The preamble is followed by the 8-bit (2 symbol) start-of-frame delimiter (SFD).

Then follows the length of the packet payload, in bytes. This field is 7 bits wide (for a maximum payload length of 127 bytes), with 1 reserved bit to make it 8 bits (2 symbols) wide. Next is the payload, with an upper bound of 127 bytes. At the end of the packet is the 16-bit (4 symbol) cyclic redundancy check (CRC) value of the packet. Note that the length field includes the length of the payload but does *not* include the preamble, SFD, length field, or CRC.

The binary data is not directly transmitted via the radio circuit. Instead, the binary data is converted from symbols to a serial bitstream of chips, and the chips modulate the transceiver on the radio circuit. The `RFcontroller` module operates on binary data; the `spreader` submodule converts symbols to chips for the radio circuit during transmission, and the `corr_despreader` submodule converts received chips into symbols.

For more information, see the IEEE 802.15.4 standard [15]. A copy is also found in `scm-digital/doc/`.

Finite State Machine Triggers

This module uses three state machines to perform all of the functions required to send or receive packets. Both the TX and RX state machines are broken down into two major steps, each of which are initiated by the Cortex-M0 or the `RFTIMER` module via a particular trigger. The Cortex-M0 initiates these steps by setting the appropriate bit in the `RFCONTROLLER_REG__CONTROL` register. The `RFTIMER` module initiates these steps by asserting one of the `rftimer` inputs connected to this module. The four types of triggers are:

`TX_LOAD` This trigger initiates the process of loading packet data into the TX FIFO for transmission.

`TX_SEND` This trigger initiates the process of sending the packet data in the TX FIFO to the radio circuit. This trigger has no effect if the FIFO is not currently loaded.

`RX_START` This trigger initiates the process of scanning the input data from the radio circuit to detect and process an incoming packet.

`RX_STOP` This trigger stops the process of detecting an incoming packet. This trigger has no effect if an incoming packet is already being processed.

These triggers are the only way for the software on the Cortex-M0 to control the radio circuit from a high level. All of the details are handled by the state machines and the `DMA_V2` module.

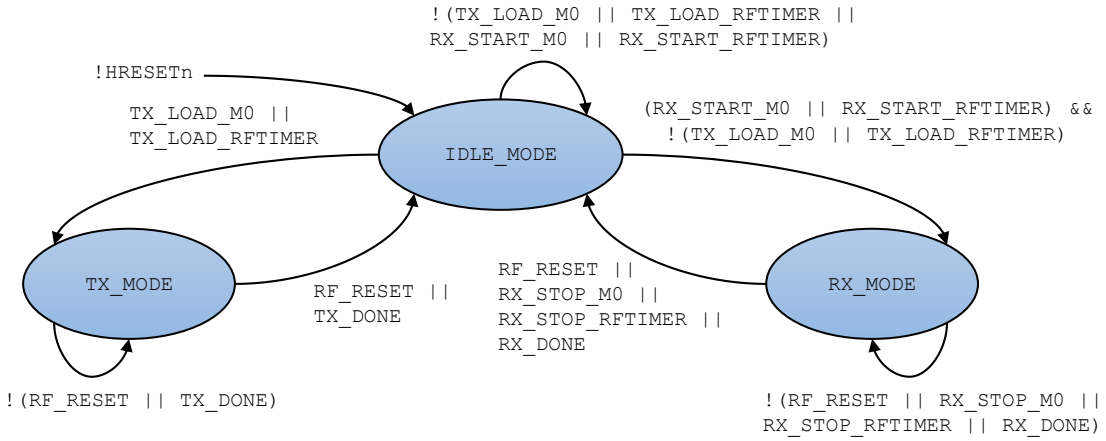


Figure 3.12: Mode select finite state machine for the `RFcontroller` module

State Name	State Encoding [MSB:LSB]
IDLE_MODE	00
TX_MODE	01
RX_MODE	10

Figure 3.13: State encodings for the `RFcontroller` mode select FSM

Mode Select Finite State Machine

The mode select state machine shown in Figure 3.12 is used to activate the TX and RX state machines when sending and receiving packets. The default state is `IDLE_MODE`, where the radio is not sending or receiving packets.

The `TX_LOAD` trigger, which comes from either setting the `TX_LOAD` bit of the `RFCONTROLLER_REG__CONTROL` register or from the `RFTIMER` module, changes the state to `TX_MODE`. When the transmission is finished, the state changes back to `IDLE_MODE`. Setting the `RF_RESET` bit of the `RFCONTROLLER_REG__CONTROL` register forces the state to change back to `IDLE_MODE`.

The `RX_START` trigger, which comes from either setting the `RX_START` bit of the `RFCONTROLLER_REG__CONTROL` register or from the `RFTIMER` module, changes the state to `RX_MODE`. When a packet is received, the state changes back to `IDLE_MODE`. If there is no incoming packet, then setting the `RX_STOP` bit of the `RFCONTROLLER_REG__CONTROL` register (or sending the `RX_STOP` trigger from the `RFTIMER` module) changes the state back to `IDLE_MODE`. `RX_STOP` does not have an effect if the RX state machine is processing an incoming packet. Setting the `RF_RESET` bit of the `RFCONTROLLER_REG__CONTROL` register forces the state to change back to `IDLE_MODE`.

If both `TX_LOAD` and `RX_START` are triggered at the same time (either from the `RFCONTROLLER_REG__CONTROL` register or the `RFTIMER` module), `TX_LOAD` takes precedence.

Figure 3.13 contains the state names and their binary encodings.

TX Finite State Machine

The TX finite state machine shown in Figure 3.14 performs all of the actions needed to transmit a packet. The TX FSM is activated when the mode changes to `TX_MODE`.

The FSM transitions through the following states and performs the following actions:

`TX_INIT` Reset the TX FIFO, `spreader` module, and synchronizer modules. A request is sent to the `DMA_V2` module for the first four bytes of data.

`TX_INIT_WAIT` Wait for the `DMA_V2` module to fetch the first four bytes of data. This also allows time for the FIFO to finish resetting.

`TX_LOAD_PHY` Load the ten bytes of PHY layer headers for the packet into the FIFO. This includes the preamble, start symbol, and packet length.

`TX_LOAD_BYTE0/1/2/3` Load byte 0/1/2/3 of the data fetched by the `DMA_V2` module into the FIFO.

`TX_DMA_WAIT` Wait for the `DMA_V2` module to fetch the next four bytes of data.

`TX_LOAD_CRC0/1` Load byte 0/1 of the CRC into the FIFO.

`TX_LOAD_DONE` Assert the `TX_LOAD_DONE` interrupt. Wait for the `TX_SEND` trigger to begin transmitting the packet.

`TX_FIFO_DRAIN` Wait for all of the packet data to be read from the FIFO and transmitted via the `spreader` module. The `spreader` module also asserts the `TX_SFD_DONE` interrupt when the last bit of the SFD is sent.

`TX_DONE` Assert the `TX_SEND_DONE` interrupt. Reset the TX FIFO, `spreader` module, and synchronizer modules.

If the mode changes to `IDLE_MODE` prematurely, either from an error or from `RF_RESET`, the TX FSM transitions to the `TX_DONE` state (in order to perform any necessary cleanup) and then to the default `TX_SLEEP` state. Figure 3.15 contains the state names and their binary encodings.

RX Finite State Machine

The RX finite state machine shown in Figure 3.16 performs all of the actions needed to listen for and receive a packet. The RX FSM is activated when the mode changes to `RX_MODE`. The FSM transitions through the following states and performs the following actions:

`RX_INIT` Reset the RX FIFO, `corr_despreader` module, and synchronizer modules.

`RX_GET_LEN` Wait for the `corr_despreader` module to detect a packet. Once a packet has been detected, the `corr_despreader` module provides the packet length and loads the packet data into the FIFO. The `corr_despreader` module also asserts the `RX_SFD_DONE` interrupt when a packet is detected.

`RX_GET_BYTE0/1/2/3` Read byte 0/1/2/3 from the FIFO.

`RX_DMA_WAIT` Wait for the `DMA_V2` module to finish storing the previous four bytes in memory.

`RX_DATA_STORE` Signal the `DMA_V2` module to store the current four bytes in memory.

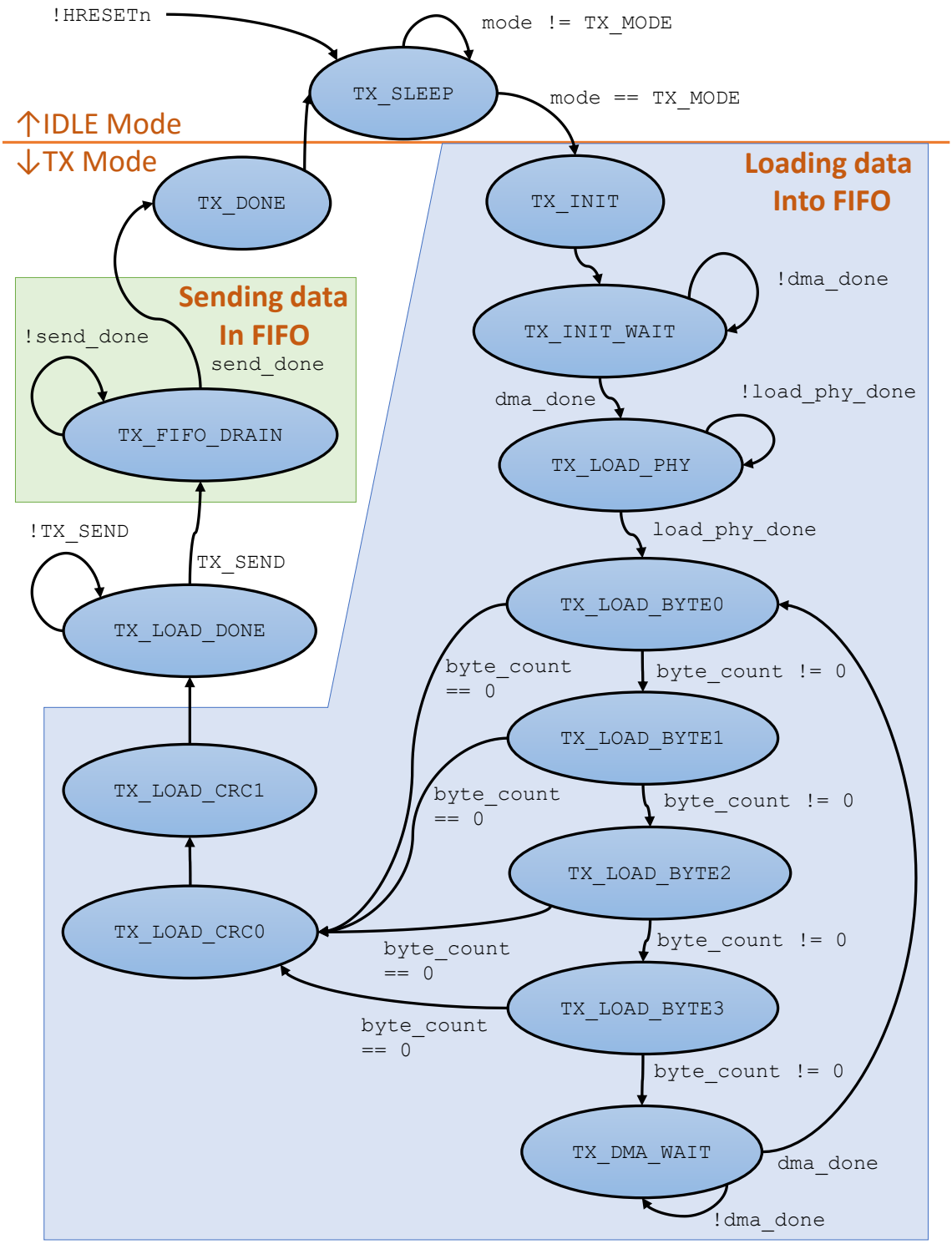


Figure 3.14: TX finite state machine for the `RFcontroller` module

State Name	State Encoding [MSB:LSB]
TX_SLEEP	0000
TX_INIT	0011
TX_INIT_WAIT	0010
TX_LOAD_PHY	0110
TX_LOAD_BYTE0	0111
TX_LOAD_BYTE1	0101
TX_LOAD_BYTE2	1100
TX_LOAD_BYTE3	1101
TX_DMA_WAIT	1111
TX_LOAD_CRC0	0100
TX_LOAD_CRC1	1110
TX_LOAD_DONE	1010
TX_FIFO_DRAIN	1000
TX_DONE	0001

Figure 3.15: State encodings for the RFcontroller TX FSM

RX_DMA_WAIT2 Wait for the DMA_V2 module to finish storying the last bytes of the packet in memory.

RX_CRC_CHECK Check if the packet's CRC is correct.

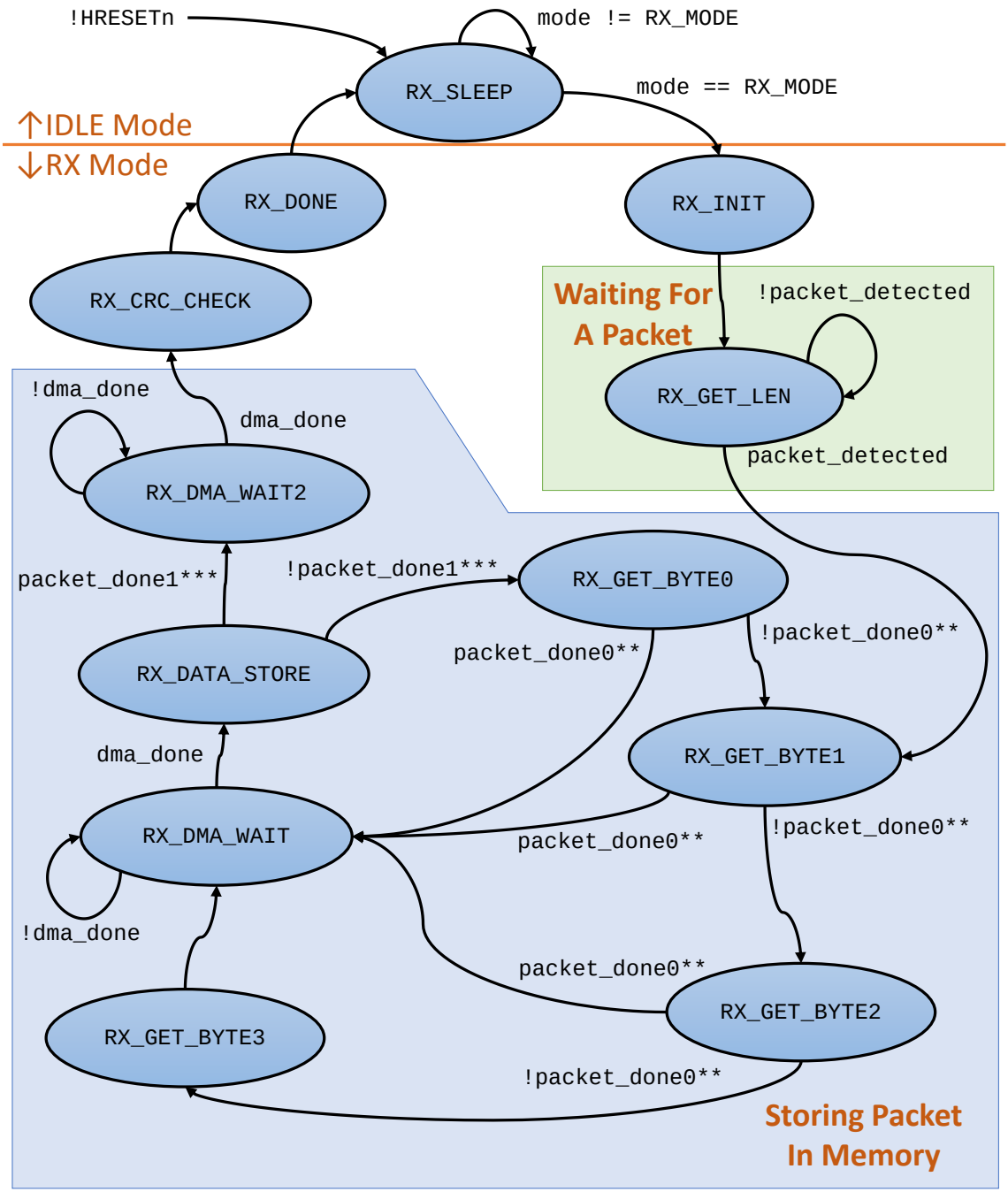
RX_DONE : Send the RX_DONE interrupt. Reset the RX FIFO, corr_despreader module, and synchronizer modules.

The RX finite state machine writes the packet data in the FIFO to the memory in the order that each byte is received. Preceding the packet data in memory is the length of the packet itself. Since the DMA_V2 module writes one word (four bytes) at a time to memory, the packet length is the least significant byte of the first word, and the rest of the packet data follows. This is accomplished by having the RX finite state machine transition to the RX_GET_BYTE1 state (rather than RX_GET_BYTE0) when reading first byte of the packet data out of the RX FIFO. The last two bytes in the RX FIFO contain the CRC value of the packet, and this is also stored in memory after the packet data.

If the mode changes to IDLE_MODE prematurely, either from an error or from RF_RESET, the RX FSM transitions to the RX_DONE state (in order to perform any necessary cleanup) and then to the default RX_SLEEP state. Figure 3.17 contains the state names and their binary encodings.

TX FIFO and Spreader

The TX FIFO (found in the tx_fifo2 module) stores an entire packet, including headers and CRC, while it is waiting to be transmitted. The TX_LOAD trigger enables the first part of the TX FSM. This part assembles the packet headers, packet data, CRC, and loads it all into the TX FIFO. The TX_SEND trigger enables the second part of the TX FSM and the spreader module. This module reads the data out of the TX FIFO and encodes it into a sequence of chips that modulate the transceiver in the radio circuit to send the packet. Both of these modules are controlled by




```
*   length_w_crc = packet_length + 2 - 1
**  packet_done0 = ( byte_count == length_w_crc* )
*** papcket_done1 = ( byte_count > length_w_crc* )
```

Figure 3.16: RX finite state machine for the `RFcontroller` module

State Name	State Encoding [MSB:LSB]
RX_SLEEP	0000
RX_INIT	0011
RX_GET_LEN	0010
RX_GET_BYTE0	1111
RX_GET_BYTE1	0111
RX_GET_BYTE2	0110
RX_GET_BYTE3	0100
RX_DMA_WAIT	0101
RX_DATA_STORE	1100
RX_DMA_WAIT2	1101
RX_CRC_CHECK	1110
RX_DONE	0001

Figure 3.17: State encodings for the RFcontroller RX FSM

the TX FSM. See section 3.26 for more information on the tx_fifo2 module and section 3.28 for more information on the spreader module.

RX FIFO and Correlator/Despreader

The RX FIFO (found in the rx_fifo module) temporarily stores all received packet data and CRC after being decoded by thecorr_despreader module and before being stored into memory. The RX_START trigger enables the RX FSM and also enables the corr_despreader module. This submodule listens to the incoming chips from the radio circuit, and detects the SFD of a packet. Once this has been found, the corr_despreader decodes the incoming chips back into binary data and stores it into the RX FIFO. At the same time, the RX FSM reads data out of the RX FIFO and stores it into memory. Both the RX FIFO and the corr_despreader module are controlled by the RX FSM. See section 3.27 for more information on the rx_fifo module and section 3.30 for more information on the corr_despreader module.

Clocking

The RFcontroller module requires three separate clock domains in order to properly interface with the radio circuit. This is because the system clock (HCLK) used by the Cortex-M0 and all of the digital peripherals runs at 5MHz, and the radio circuit runs at 2MHz. The Single Chip Mote has a 2MHz source for transmitting packets called CLK_TX in the top module and connected to the tx_clk input to the RFcontroller module. The Single Chip Mote also has a separate 2MHz clock for receiving packets called CLK_RX in the top module and connected to the rx_clk input of the RFcontroller module. The RX clock comes from an analog circuit on the Single Chip Mote; this circuit generates a clock that is aligned with the incoming RX data.

Inside the RFcontroller module the control registers and the finite state machines use HCLK. The spreader module uses tx_clk and the corr_despreader module uses rx_clk. The TX FIFO and RX FIFO are designed to be able to synchronize between separate clock domains and any additional cross-domain signals are properly synchronized using the bit_sync and bus_sync modules.

DMA Interface

The RFcontroller module relies on the DMA_V2 module to transfer packet data to and from the data memory. The DMA_V2 module is capable of reading and writing to the RFcontroller via the AHB bus. This is because the DMA_V2 module has an AHB master interface, and this interface shares the AHBsub bus with the Cortex-M0. Both the DMA_V2 module and the Cortex-M0 are AHB masters to the two AHB slaves on this shared bus: the RFcontroller module and the AHBDMEM module. The RFcontroller has two outputs connected directly to the DMA_V2 to send requests for data transfers over the AHB.

The rf_data_req output requests data from the memory for packet transmission. This output port is connected to the txdatareq register in the RFcontroller module. The txdatareq register is set by the TX FSM when it is time to fetch more data for transmission. The DMA_V2 module reads the RFCONTROLLER_REG__TX_DATA_ADDR_DMA register to find the address of the data to fetch. Reading this register also clears the txdatareq register, as this read indicates that the DMA_V2 is servicing the request. The TX FSM waits for the DMA_V2 to write to the RFCONTROLLER_REG__TX_DATA_DMA register with the new packet data. This also causes the address stored in the RFCONTROLLER_REG__TX_DATA_ADDR_DMA register to increment by 4.

The rf_data_store output requests that the DMA_V2 copy the received packet data in the RFCONTROLLER_REG__RX_DATA_DMA register to the data memory.

The rf_data_store output is a register set by the RX FSM after the RFCONTROLLER_REG__RX_DATA_DMA is updated with new packet data. The rf_data_store register is cleared when the DMA_V2 reads from the RFCONTROLLER_REG__RX_DATA_DMA register, as this read indicates that the DMA_V2 is servicing the request.

In this case the DMA_V2 module, rather than the RFcontroller module, keeps track of where the new data is written, using the DMA_REG__RF_RX_ADDR register. The DMA_V2 module increments the address in DMA_REG__RF_RX_ADDR by 4 after every write.

For more information on the DMA_V2 module, see section 3.24.

Interrupts and Errors

The RFcontroller has one interrupt to the Cortex-M0, as well as a register to indicate any errors. This single interrupt is a combination of several interrupt and error sources. This module also has a direct connection to the RFTIMER module to send interrupts (in the form of a single-cycle pulse) to any of its capture units (see section 3.35 spec for more details).

The five interrupt sources are:

TX_LOAD_DONE The TX FSM finishes copying packet data into the TX FIFO

TX_SFD_DONE The spreader finishes transmitting the last bit of the packet's SFD to the radio circuit

TX_SEND_DONE The TX FSM finishes transmitting a packet

RX_SFD_DONE The correlator/despreader detects an incoming packet

RX_DONE The RX FSM finishes receiving an incoming packet and storing the data into memory

The five error sources are:

`TX_OVERFLOW_ERROR` The TX FIFO overflows

`TX_CUTOFF_ERROR` The TX FM is reset while a packet is sending

`RX_OVERFLOW_ERROR` The RX FIFO overflows

`RX_CRC_ERROR` The CRC of a received packet is incorrect (the packet data is still copied into memory)

`RX_CUTOFF_ERROR` The RX FSM is reset while processing an incoming packet

Each of these interrupt or error sources correspond to a bit in the `RFCONTROLLER_REG__INT` or `RFCONTROLLER_REG__ERROR` registers. The `RFCONTROLLER_REG__INT_CONFIG` and `RFCONTROLLER_REG__ERROR_CONFIG` registers contain `INT_EN` and `ERROR_EN` bits used to enable or disable each interrupt or error source. An enabled interrupt source sets the corresponding bit in the `RFCONTROLLER_REG__INT` register when the interrupt is triggered. A disabled interrupt source has no impact on the `RFCONTROLLER_REG__INT` register. The same applies to the error sources.

The Cortex-M0 interrupt is composed of the bitwise OR of the bits in the `RFCONTROLLER_REG__INT` and `RFCONTROLLER_REG__ERROR` registers, after they are masked by the `INT_MASK` and `ERROR_MASK` bits in the `RFCONTROLLER_REG__INT_CONFIG` and `RFCONTROLLER_REG__ERROR_CONFIG` registers. A masked bit in the `RFCONTROLLER_REG__INT` register means that it can be set but it will not trigger the Cortex-M0 interrupt. An unmasked bit will trigger the Cortex-M0 interrupt if it is set in the `RFCONTROLLER_REG__INT` register. The same applies to the bits in the `RFCONTROLLER_REG__ERROR` register.

In addition, each interrupt source is connected to the capture units of the `RFTIMER` module. Triggering the interrupt sends a single-cycle pulse to the `RFTIMER` module, if that interrupt source is enabled. This is set by the `PULSE_EN` bits in the `RFCONTROLLER_REG__INT_CONFIG` register. Error sources are not connected to the `RFTIMER` module

3.25.4 Register Interface

Control Register

The `RFCONTROLLER_REG__CONTROL` register is a 5-bit register with fields that trigger parts of the TX and RX state machines. Its five bits are `TX_LOAD`, `TX_SEND`, `RX_START`, `RX_STOP`, and `RF_RESET`.

The `TX_LOAD` bit changes the mode from `IDLE_MODE` to `TX_MODE`, and launches the first part of the TX FSM responsible for loading packet data into memory. Once this is finished, the `TX_SEND` bit launches the second part of the TX FSM, responsible for sending the packet through the radio circuit. After this is finished, the mode returns to `IDLE_MODE`, and the TX state returns to `TX_SLEEP`. Setting the `TX_LOAD` bit has no effect when the mode is not `IDLE_MODE`, and setting the `TX_SEND` bit has no effect when the FIFO is not loaded (or in other words, the TX state is not `TX_LOAD_DONE`).

The `RX_START` bit changes the mode from `IDLE_MODE` to `RX_MODE`, and launches the first part of the RX FSM responsible for listening for new packets. The `RX_STOP`

bit resets the RX state back to RX_SLEEP and changes the mode back to IDLE_MODE, under the condition that the RFcontroller module is not currently receiving a packet. Setting the RX_START bit has no effect when the mode is not IDLE_MODE, and setting the RX_STOP bit has no effect when the radio is not listening for packets or is currently receiving a packet (or in other words, the RX state is not RX_GET_LEN).

The RF_RESET bit resets all state machines back to their initial states in case of an unrecoverable error.

Status Register

The RFCONTROLLER_REG__STATUS register is a 10-bit register containing the current states of the three state machines. This register is used either to check the progress of sending/receiving a packet, or to check for any unexpected behavior (such as the FSM being 'stuck' in one state) for debugging purposes. This register contains two MODE bits for the state of the mode select FSM, four TX_STATE bits for the TX FSM, and four RX_STATE bits for the RX FSM. The encodings for each state are specified in Figures 3.13, 3.15, and 3.17, respectively.

TX Data Address and Packet Length

The RFCONTROLLER_REG__TX_DATA_ADDR register is a 32-bit register containing the start address of the data to be transmitted. The address in this register must be word-aligned. All of the packet data must be in a continuous, sequential, word-aligned section of data memory beginning at the address stored in RFCONTROLLER-_REG__TX_DATA_ADDR.

The RFCONTROLLER_REG__PACK_LEN register is a 7-bit register containing the length of the data to be transmitted in bytes. The maximum data length, as defined by the IEEE 802.15.4 standard [15] is 127 bytes.

Both the RFCONTROLLER_REG__TX_DATA_ADDR and RFCONTROLLER_REG__PACK-_LEN registers must be updated before sending a packet to ensure that they point to the correct location in memory and indicate the correct length of the packet payload; however, their contents are not modified by the RFcontroller module during its operation.

RX Data Address

The DMA_REG__RF_RX_ADDR register is not part of the RFcontroller module; however, it must be updated before listening for a packet. This 32-bit register in the DMA_V2 module contains the starting address where received data is stored (see section 3.24 for more details). This address must refer to a continuous, word-aligned section of data memory designated in the software, with a length of at least 130 bytes, in order to be able to store the largest possible packet (127 bytes) and three additional bytes for the packet length and CRC. The address stored in DMA_REG__RF_RX_ADDR is modified by the DMA_V2 module while receiving a packet, and therefore must be set back to the correct value before listening for another packet.

DMA Exclusive Registers

The following registers are part of the RFcontroller module, but are not meant to be accessed by the software running on the Cortex-M0. These registers are used by

the DMA_V2 module to transfer packet data between the RFcontroller module and the data memory.

The RFCONTROLLER_REG__TX_DATA_ADDR_DMA register is a 32-bit register containing the address of the next 4 bytes of data the DMA_V2 module must fetch for a packet transmission. The TX FSM copies the contents of the RFCONTROLLER_REG__TX_DATA_ADDR register into this register before loading the TX FIFO, and increments the contents of this register by 4 every time the DMA_V2 fetches new data.

The RFCONTROLLER_REG__TX_DATA_DMA register is a 32-bit register written by the DMA_V2 module with the data fetched for a packet transmission. The contents of this register are copied into the TX FIFO.

The RFCONTROLLER_REG__RX_DATA_DMA register is a 32-bit register containing the received packet data that the DMA_V2 copies into the FIFO. The RX FSM updates this value with new packet data and the DMA_V2 module reads this register and copies the contents to the data memory.

Interrupt and Error Registers

The RFCONTROLLER_REG__INT register is a 5-bit register indicating any interrupts from the RFcontroller module. The five bits correspond to the five types of interrupts: TX_LOAD_DONE, TX_SFD_DONE, TX_SEND_DONE, RX_SFD_DONE, and RX_DONE. These bits are set if the interrupt is triggered and is enabled in the RFCONTROLLER_REG__INT_CONFIG register. These bits are cleared by writing to the RFCONTROLLER_REG__INT_CLEAR register.

The RFCONTROLLER_REG__INT_CONFIG register is a 15-bit register containing the configuration flags for each interrupt source. The five INT_EN bits enable or disable each interrupt source. An enabled interrupt source sets its corresponding bit in RFCONTROLLER_REG__INT when it is triggered, and a disabled interrupt source has no effect. The five PULSE_EN bits enable or disable the pulse sent to the RFTIMER module for each interrupt source. An interrupt with a set PULSE_EN bit sends a single-cycle pulse to the RFTIMER module when it is triggered. The five INT_MASK bits determine whether or not the corresponding bit in RFCONTROLLER_REG__INT triggers the Cortex-M0 interrupt. An interrupt source with a set INT_EN bit and a set INT_MASK bit will set the corresponding bit in RFCONTROLLER_REG__INT when it triggers, but it will not trigger an interrupt to the Cortex-M0. Clearing the INT_MASK bit while the corresponding bit in RFCONTROLLER_REG__INT is still set triggers a Cortex-M0 interrupt.

The RFCONTROLLER_REG__INT_CLEAR register is a 5-bit register that clears the bits in RFCONTROLLER_REG__INT. Setting any bit in this register to 1 clears the corresponding bit in RFCONTROLLER_REG__INT. Any unmasked bits set in the RFCONTROLLER_REG__INT register trigger the Cortex-M0 interrupt and a call to the interrupt service routine. The interrupt service routine must clear any unmasked bits (or mask them); otherwise, the ISR will execute again until all unmasked bits are cleared.

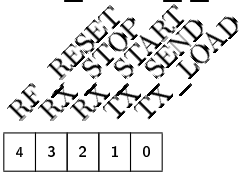
The RFCONTROLLER_REG__ERROR register is a 5-bit register indicating any errors from the RFcontroller module. The five bits correspond to the five types of errors: TX_OVERFLOW_ERROR, TX_CUTOFF_ERROR, RX_OVERFLOW_ERROR, RX_CRC_ERROR, and RX_CUTOFF_ERROR. These bits are set if the error is triggered and is enabled in the RFCONTROLLER_REG__ERROR_CONFIG register. These bits are cleared by writing to the RFCONTROLLER_REG__ERROR_CLEAR register.

The RFCONTROLLER_REG__ERROR_CONFIG register is a 10-bit register containing the configuration flags for each error source. The five ERROR_EN bits enable or disable each error source. An enabled error source sets its corresponding bit in RFCONTROLLER_REG__ERROR when it is triggered, and a disabled error source has no effect. The five ERROR_MASK bits determine whether or not the corresponding bit in RFCONTROLLER_REG__ERROR triggers the Cortex-M0 interrupt. An error source with a set ERROR_EN bit and a set ERROR_MASK bit will set the corresponding bit in RFCONTROLLER_REG__ERROR when it triggers, but this will not trigger an interrupt to the Cortex-M0. Clearing the ERROR_MASK bit while the corresponding bit in RFCONTROLLER_REG__ERROR is still set triggers a Cortex-M0 interrupt.

The RFCONTROLLER_REG__ERROR_CLEAR register is a 5-bit register that clears the bits in RFCONTROLLER_REG__ERROR. Setting any bit in this register to 1 clears the corresponding bit in RFCONTROLLER_REG__ERROR. Any unmasked bits set in the RFCONTROLLER_REG__ERROR register trigger the Cortex-M0 interrupt and a call to the interrupt service routine. The interrupt service routine must clear any unmasked bits (or mask them); otherwise, the ISR will execute again until all unmasked bits are cleared.

Register Descriptions

Register 3.4: RFCONTROLLER_REG__CONTROL (0x40000000)



TX_LOAD (Write-Only) Triggers the TX state machine to load packet data into TX FIFO. 0 = no load and 1 = load.

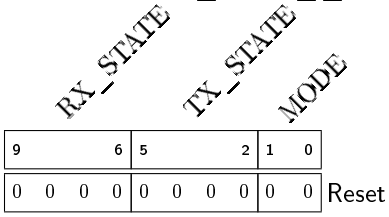
TX_SEND (Write-Only) Triggers the TX state machine to send packet data in the TX FIFO. 0 = no send and 1 = send.

RX_START (Write-Only) Triggers the RX state machine to listen for incoming packets. 0 = no start and 1 = start.

RX_STOP (Write-Only) Stops the RX state machine from listening to incoming packets. 0 = no stop and 1 = stop.

RF_RESET (Write-Only) Resets the mode select, TX, and RX state machines. 0 = no reset and 1 = reset.

Register 3.5: RFCONTROLLER_REG__STATUS (0x40000004)

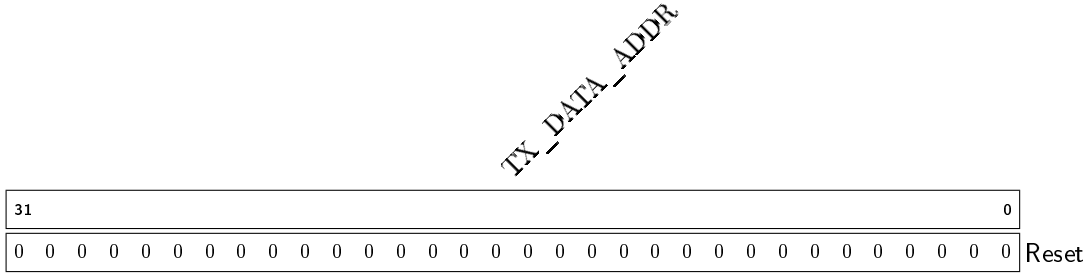


MODE (Read-Only) State of the mode select state machine. See Figure 3.13 for state encodings.

TX_MODE (Read-Only) State of the TX state machine. See Figure 3.15 for state encodings.

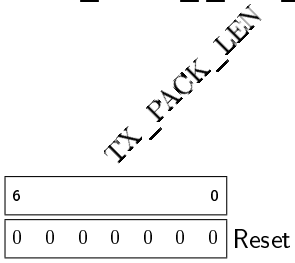
RX_MODE (Read-Only) State of the RX state machine. See Figure 3.17 for state encodings.

Register 3.6: RFCONTROLLER_REG__TX_DATA_ADDR (0x40000008)



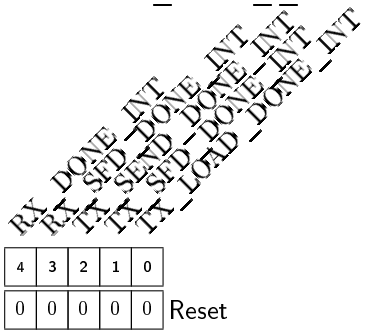
TX_DATA_ADDR Address pointing to the beginning of the packet data to transmit.

Register 3.7: RFCONTROLLER_REG__TX_PACK_LEN (0x4000000C)



TX_PACK_LEN Length of the packet to transmit.

Register 3.8: RFCONTROLLER_REG__INT (0x40000010)



TX_LOAD_DONE_INT (Read-Only) TX load done interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

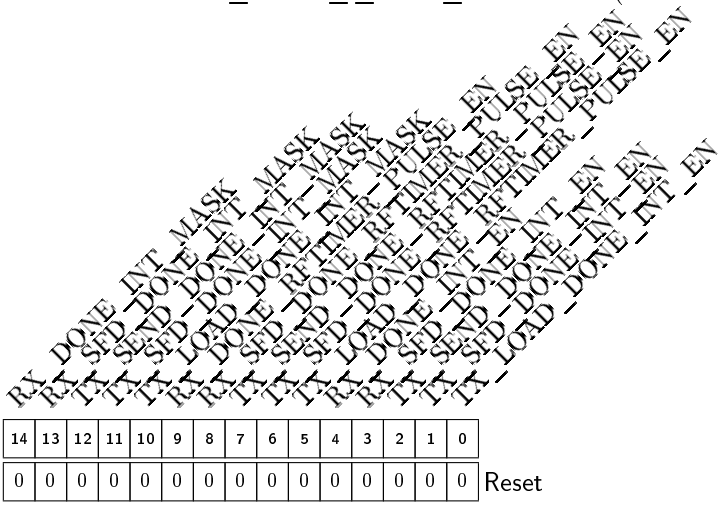
TX_SFD_DONE_INT (Read-Only) TX SFD transmission done interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

TX_SEND_DONE_INT (Read-Only) TX packet transmission done interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

RX_SFD_DONE_INT (Read-Only) RX SFD detection interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

RX_DONE_INT (Read-Only) RX packet stored interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

Register 3.9: RFCONTROLLER_REG__INT_CONFIG (0x40000014)



TX_LOAD_DONE_INT_EN TX load done interrupt enable. 0 = disabled and 1 = enabled.

TX_SFD_DONE_INT_EN TX SFD transmission done interrupt enable. 0 = disabled and 1 = enabled.

TX_SEND_DONE_INT_EN TX packet transmission done interrupt enable. 0 = disabled and 1 = enabled.

RX_SFD_DONE_INT_EN RX SFD detection interrupt enable. 0 = disabled and 1 = enabled.

RX_DONE_INT_EN RX packet stored interrupt enable. 0 = disabled and 1 = enabled.

TX_LOAD_DONE_RFTIMER_PULSE_EN TX load done output pulse enable. 0 = disabled and 1 = enabled.

TX_SFD_DONE_RFTIMER_PULSE_EN TX SFD transmission done output pulse enable. 0 = disabled and 1 = enabled.

TX_SEND_DONE_RFTIMER_PULSE_EN TX packet transmission done output pulse enable. 0 = disabled and 1 = enabled.

RX_SFD_DONE_RFTIMER_PULSE_EN RX SFD detection output pulse enable. 0 = disabled and 1 = enabled.

RX_DONE_RFTIMER_PULSE_EN RX packet stored output pulse enable. 0 = disabled and 1 = enabled.

TX_LOAD_DONE_INT_MASK TX load done interrupt mask. 0 = not masked and 1 = masked.

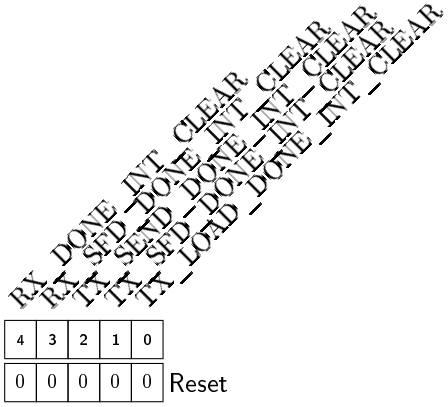
TX_SFD_DONE_INT_MASK TX SFD transmission done interrupt mask. 0 = not masked and 1 = masked.

TX_SEND_DONE_INT_MASK TX packet transmission done interrupt mask. 0 = not masked and 1 = masked.

RX_SFD_DONE_INT_MASK RX SFD detection interrupt mask. 0 = not masked and 1 = masked.

RX_DONE_INT_MASK RX packet stored interrupt mask. 0 = not masked and 1 = masked.

Register 3.10: RFCONTROLLER_REG__INT_CLEAR (0x40000018)



TX_LOAD_DONE_INT_CLEAR (Write-Only) TX load done interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

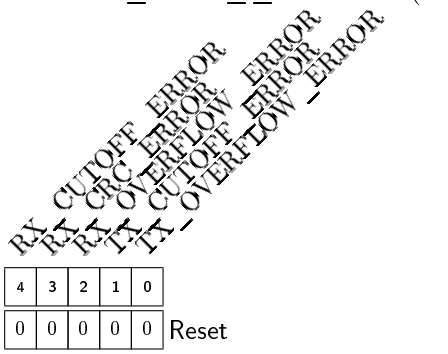
TX_SFD_DONE_INT_CLEAR (Write-Only) TX SFD transmission done interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

TX_SEND_DONE_INT_CLEAR (Write-Only) TX packet transmission done interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

RX_SFD_DONE_INT_CLEAR (Write-Only) RX SFD detection interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

RX_DONE_INT_CLEAR (Write-Only) RX packet stored interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

Register 3.11: RFCONTROLLER_REG__ERROR (0x4000001C)



TX_OVERFLOW_ERROR (Read-Only) TX overflow error flag. 0 = no error pending and 1 = error pending.

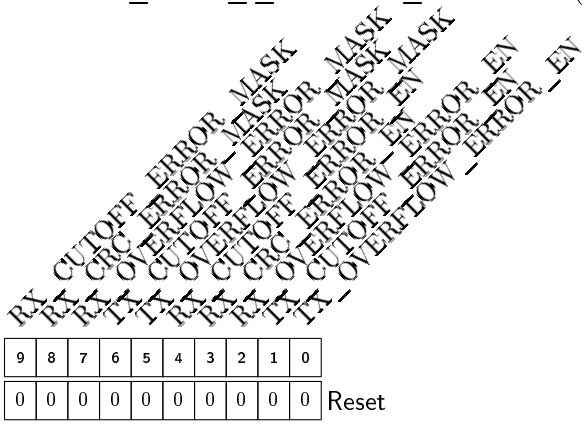
TX_CUTOFF_ERROR (Read-Only) TX cutoff error flag. 0 = no error pending and 1 = error pending.

RX_OVERFLOW_ERROR (Read-Only) RX overflow error flag. 0 = no error pending and 1 = error pending.

RX_CRC_ERROR (Read-Only) RX CRC error flag. 0 = no error pending and 1 = error pending.

RX_CUTOFF_ERROR (Read-Only) RX cutoff error flag. 0 = no error pending and 1 = error pending.

Register 3.12: RFCONTROLLER_REG__ERROR_CONFIG (0x40000020)



TX_OVERFLOW_ERROR_EN TX overflow error enable. 0 = disabled and 1 = enabled.

TX_CUTOFF_ERROR_EN TX cutoff error enable. 0 = disabled and 1 = enabled.

RX_OVERFLOW_ERROR_EN RX overflow error enable. 0 = disabled and 1 = enabled.

RX_CRC_ERROR_EN RX CRC error enable. 0 = disabled and 1 = enabled.

RX_CUTOFF_ERROR_EN RX cutoff error enable. 0 = disabled and 1 = enabled.

TX_OVERFLOW_ERROR_MASK TX overflow error mask. 0 = not masked and 1 = masked.

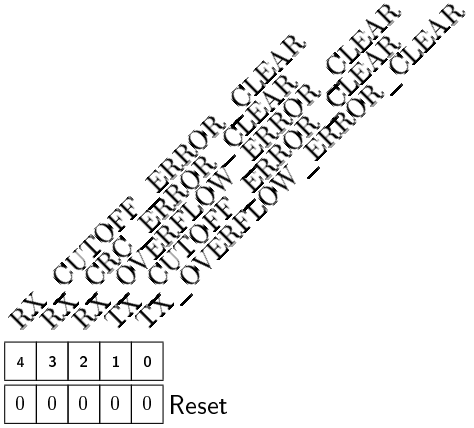
TX_CUTOFF_ERROR_MASK TX cutoff error mask. 0 = not masked and 1 = masked.

RX_OVERFLOW_ERROR_MASK RX overflow error mask. 0 = not masked and 1 = masked.

RX_CRC_ERROR_MASK RX CRC error mask. 0 = not masked and 1 = masked.

RX_CUTOFF_ERROR_MASK RX cutoff error mask. 0 = not masked and 1 = masked.

Register 3.13: RFCONTROLLER_REG__ERROR_CLEAR (0x40000024)



TX_OVERFLOW_ERROR_CLEAR (Write-Only) TX overflow error flag clear. 0 = flag unchanged and 1 = flag cleared.

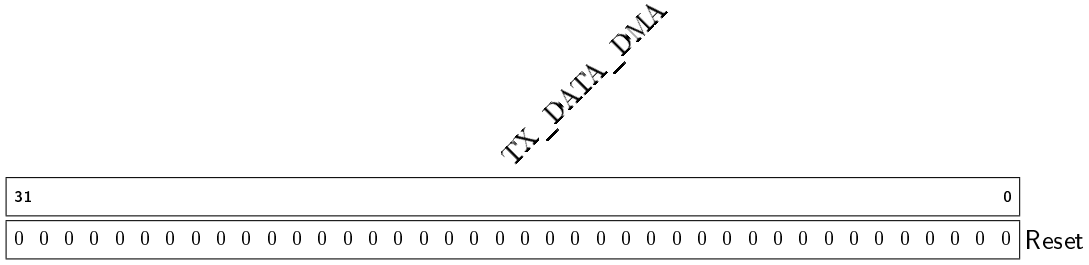
TX_CUTOFF_ERROR_CLEAR (Write-Only) TX cutoff error flag clear. 0 = flag unchanged and 1 = flag cleared.

RX_OVERFLOW_ERROR_CLEAR (Write-Only) RX overflow error flag clear. 0 = flag unchanged and 1 = flag cleared.

RX_CRC_ERROR_CLEAR (Write-Only) RX CRC error flag clear. 0 = flag unchanged and 1 = flag cleared.

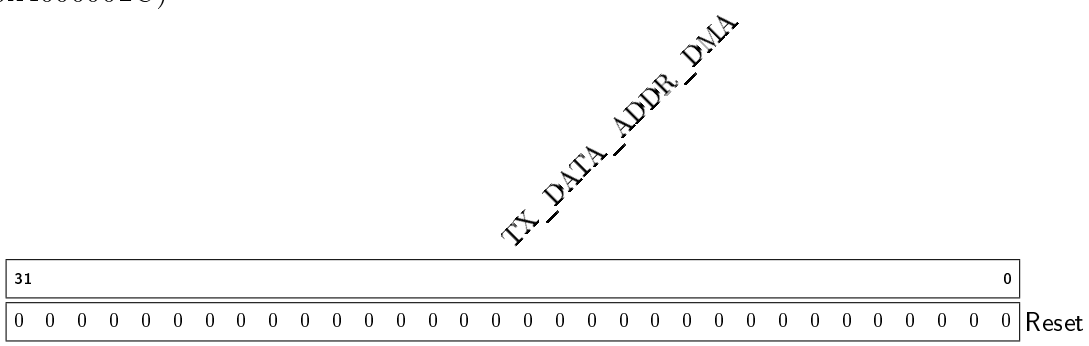
RX_CUTOFF_ERROR_CLEAR (Write-Only) RX cutoff error flag clear. 0 = flag unchanged and 1 = flag cleared.

Register 3.14: RFCONTROLLER_REG__TX_DATA_DMA (0x40000028)



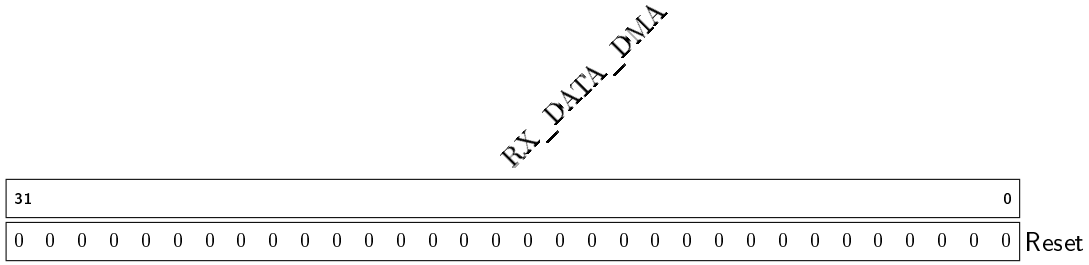
TX_DATA_DMA (Read-only) Address used by the DMA to fetch data for packet transmissions. THIS REGISTER IS EXCLUSIVELY FOR USE BY THE DMA.

Register 3.15: RFCONTROLLER_REG__TX_DATA_ADDR_DMA (0x4000002C)



TX_DATA_ADDR_DMA Packet data for transmission, written by the DMA. Address used by the DMA to fetch data for packet transmissions. THIS REGISTER IS EXCLUSIVELY FOR USE BY THE DMA.

Register 3.16: RFCONTROLLER_REG__RX_DATA_DMA (0x40000030)



RX_DATA_DMA (Read-only) Received packet data, read and stored in the data memory by the DMA. THIS REGISTER IS EXCLUSIVELY FOR USE BY THE DMA.

3.26 tx_fifo2

3.26.1 Description

This module is an asynchronous, asymmetric FIFO designed to store packet data waiting to be transmitted in the `RFcontroller` module. This module is asynchronous since it has two different clock inputs for reading (2MHz `CLK_TX`) and writing (5MHz `HCLK`), and is asymmetric since the read data width (4 bits) and write data width (8 bits) are different. This FIFO stores up to 265 bytes and is large enough to store a transmitted packet at its maximum size. The design is based on the one described in [8], with additional modifications to deal with the asymmetric read and write data widths.

3.26.2 Input/Output Ports

`reset_n` Input reset.

`wr_clk` Input write clock. The write data and write enable signals must be synchronized to this clock. The full flag is synchronized to this clock.

`wr_en` Write enable input. When this input is 1, the data on `wr_data` is written into the FIFO.

`wr_data[7:0]` Write data input. This is the data written into the FIFO.

`wr_full` FIFO full output. This indicates that the FIFO is full and cannot accept any more data. Any attempts to write into a full FIFO will fail and the data will be lost.

`rd_clk` Input read clock. The read data and read enable signals must be synchronized to this clock. The empty flag is synchronized to this clock.

`rd_en` Read enable input. When this input is 1, the next value in the FIFO is read to the `rd_data` output on the next cycle.

`rd_data[3:0]` Read data output. This is the data read from the FIFO.

`rd_empty` FIFO empty output. This indicates that the FIFO is empty and is not able to read additional data. Any attempts to read from an empty FIFO will fail and invalid data will be present on `rd_data`.

3.26.3 Design Details

Original Design

The original implementation in [8] uses an asynchronous read and write pointer comparison technique to reduce the amount of synchronization flip-flops in the design. Another side-effect of this technique is the improved temporal accuracy of the `rd_empty` and `wr_full` outputs.

The traditional asynchronous FIFO design requires a shift register to sample the read pointer with the write clock, and another shift register to sample the write pointer with the read clock. A FIFO using 8-bit pointers and a shift register depth of 3 (since greater depths reduce the chances of synchronization errors), requires 48 flip-flops. There is also at least a 3 cycle delay before the full or empty signals to de-assert. In contrast, the method used in [8] requires only 5 flip-flops and some extra combinational logic.

This method creates an asynchronous empty signal, where its rising edge aligned to the read clock and its falling edge aligned to the write clock. A simple circuit containing two flip-flops solves this issue and synchronizes both edges to the read clock. This results in a `rd_empty` signal that asserts as soon as the FIFO is empty, and de-asserts on the next read clock edge after the FIFO is not empty. The full signal has its rising edge aligned to the write clock and its falling edge aligned to the read clock. The same two-flop circuit synchronizes both edges to the write clock, and the `wr_full` signal asserts as soon as the FIFO is full, and de-asserts on the next write clock edge after the FIFO is not full.

For more information on this FIFO design, see [8]. A copy is found in `scm-digital/doc/`.

Read Data Order

The original implementation in [8] assumes that the read and write data widths are the same. However, the `RFcontroller` modules requires a FIFO with 8-bit writes and 4-bit reads. For every 8 bits written to the FIFO, the lower 4 bits, [3:0], are read out of the FIFO first, and the upper 4 bits, [7:4], are read out of the FIFO second.

Pointer Comparison Modifications

The original implementation in [8] uses Gray code read and write address pointers instead of binary-encoded pointers. Using Gray code reduces the chance of comparison and synchronization errors. However, the implementation in `tx_fifo2` requires asymmetric read and write data widths, resulting in different read and write address pointer sizes. Given that the read data width is half of the write data width, the read pointer requires one more bit over the write pointer.

Consider the n-bit read pointer `rd_addr[n-1:0]` and the n-1-bit write pointer, `wr_addr[n-2:0]`. If the pointers are binary-encoded, then the FIFO is be full after a write when `rd_addr[n-1:1] == wr_addr[n-2:0]`,and empty after a read when `rd_addr[n-1:0] == {wr_addr[n-2:0], 1'b0}`. The empty comparison fails when applied to two Gray code pointers of different sizes. This is because the least significant bit of `rd_addr` is sometimes 0 after two reads, and sometimes 1 after two reads. As a Grey code pointer increments, the least significant bit changes according to the following pattern: 0110 (as opposed to 0101 with a binary pointer). For the pointers in this design, the problem is fixed by changing `{wr_addr[n-2:0], 1'b0}` to `{wr_addr[n-2:0], ^wr_addr}`.

FIFO Memory Modifications

The memory used to store the FIFO data is designed to behave like a synchronous SRAM with a width of 8 bits and a depth of 256 bits. This SRAM is not instantiated; however, the syntax used in the Verilog code infers a synchronous SRAM.

The write data width is 8 bits; one word of the RAM is written during every write, and the write address width of the FIFO matches write address width of the RAM. The read data width is 4 bits; therefore, the most significant bits (`rd_addr[8:1]`) of the read address are used to address the 8-bit word in the RAM containing the desired 4 bits. The least significant bit (`rd_addr[0]`) is used to determine which set of 4 bits in the RAM is the correct output.

Grey code pointers further complicate the read handling. As a Grey code pointer increments, the least significant bit changes according to the following pattern: 0110 (as opposed to 0101 with a binary pointer). Therefore, sometimes `rd_addr[0] == 0` means the lower 4 bits must be read, and sometimes it means the upper 4 bits must be read. This is determined based on the value of `^rd_addr[8:1]`.

Overflow and Underflow Handling

The original implementation in [8] does not detect or handle overflows and underflows. This implementation was modified to allow reads only when the FIFO is not empty (by using `rd_en && !rd_empty` instead of only `rd_en`) and writes only when the FIFO is not full (by using `wr_en && !wr_full` instead of only `wr_full`). The

asynchronous pointer comparisons allow for the `rd_empty` output to assert as soon as the FIFO is empty and the `wr_full` output to assert as soon as the FIFO is full; this is required for the prevention of overflows and underflows.

3.27 rx_fifo

3.27.1 Description

This module is an asynchronous, asymmetric, first word fall through (FWFT) FIFO designed to received packet data in the `RFcontroller` module. This module is asynchronous since it has two different clock inputs for reading (5MHz `HCLK`) and writing (2MHz `CLK_RX`), and is asymmetric since the read data width (8 bits) and write data width (4 bits) are different. This FIFO stores up to 128 bytes, 2 bytes smaller than the maximum size of received packet data. However, the `RFcontroller` module reads data out of the FIFO as soon as it is written, and therefore it is highly unlikely that the FIFO will overflow. The design is based on the one described in [8], with additional modifications to deal with the asymmetric read and write data widths and the addition of first word fall through logic.

3.27.2 Input/Output Ports

`reset_n` Input reset.

`wr_clk` Input write clock. The write data and write enable signals must be synchronized to this clock. The full flag is synchronized to this clock.

`wr_en` Write enable input. When this input is 1, the data on `wr_data` is written into the FIFO.

`wr_data[3:0]` Write data input. This is the data written into the FIFO.

`wr_full` FIFO full output. This indicates that the FIFO is full and cannot accept any more data. Any attempts to write into a full FIFO will fail and the data will be lost.

`rd_clk` Input read clock. The read data and read enable signals must be synchronized to this clock. The empty flag is synchronized to this clock.

`rd_en` Read enable input. When this input is 1, the next value in the FIFO is read to the `rd_data` output on the next cycle.

`rd_data[7:0]` Read data output. This is the data read from the FIFO.

`rd_empty` FIFO empty output. This indicates that the FIFO is empty and is not able to read additional data. Any attempts to read from an empty FIFO will fail and invalid data will be present on `rd_data`.

3.27.3 Design Details

Original Design

The original implementation in [8] uses an asynchronous read and write pointer comparison technique to reduce the amount of synchronization flip-flops in the design. Another side-effect of this technique is the improved temporal accuracy of the `rd_empty` and `wr_full` outputs.

The traditional asynchronous FIFO design requires a shift register to sample the read pointer with the write clock, and another shift register to sample the write pointer with the read clock. A FIFO using 8-bit pointers and a shift register depth of 3 (since greater depths reduce the chances of synchronization errors), requires 48 flip-flops. There is also at least a 3 cycle delay before the full or empty signals to de-assert. In contrast, the method used in [8] requires only 5 flip-flops and some extra combinational logic.

This method creates an asynchronous empty signal, with its rising edge aligned to the read clock and its falling edge aligned to the write clock. A simple circuit containing two flip-flops solves this issue and synchronizes both edges to the read clock. This results in a `rd_empty` signal that asserts as soon as the FIFO is empty, and de-asserts on the next read clock edge after the FIFO is not empty. The full signal has its rising edge aligned to the write clock and its falling edge aligned to the read clock. The same two-flop circuit synchronizes both edges to the write clock, and the `wr_full` signal asserts as soon as the FIFO is full, and de-asserts on the next write clock edge after the FIFO is not full.

For more information on this FIFO design, see [8]. A copy is found in `scm-digital/doc/`.

Write Data Order

The original implementation in [8] assumes that the read and write data widths are the same. However, the `RFcontroller` modules requires a FIFO 4-bit writes and 8-bit reads. For every 8 bits read from the FIFO, the lower 4 bits, [3:0], are written to the FIFO first, and the upper 4 bits, [7:4], are written to the FIFO second.

Pointer Comparison Modifications

The original implementation in [8] uses Gray code read and write address pointers instead of binary-encoded pointers. Using Gray code reduces the chance of comparison and synchronization errors. However, the implementation in `tx_fifo2` requires asymmetric read and write data widths, resulting in different read and write address pointer sizes. Given that the write data width is half of the read data width, the read pointer requires one less bit than the write pointer.

Consider the n-1-bit read pointer `rd_addr[n-2:0]` and the n-bit write pointer, `wr_addr[n-1:0]`. If the pointers are binary-encoded, then the FIFO is be empty after a read when `rd_addr[n-2:0] == wr_addr[n-1:1]`, and full after a write when `{rd_addr[n-2:0], 1'b0} == wr_addr[n-1:0]`. The full comparison fails when applied to two Gray code pointers of different sizes. This is because the least significant bit of `wr_addr` is sometimes 0 after two writes, and sometimes 1 after two writes. As a Grey code pointer increments, the least significant bit changes according to the following pattern: 0110 (as opposed to 0101 with a binary pointer). For the

pointers in this design, the problem is fixed by changing `{rd_addr[n-2:0], 1'b0}` to `{rd_addr[n-2:0], ^rd_addr}`.

FIFO Memory Modifications

The memory used to store the FIFO data is designed to behave like a synchronous SRAM with a width of 8 bits, a depth of 128 bits, and write enable signals for the upper and lower 4 bits of the word line. This SRAM is not instantiated; however, the syntax used in the Verilog code infers a synchronous SRAM.

The read data width is 8 bits; one word in the RAM is read for every read, and the read address width of the FIFO matches read address width in the RAM. The write data width is 4 bits; therefore, the most significant bits (`wr_addr[7:1]`) of the write address are used to address corresponding the 8-bit word in the RAM. The least significant bit (`wr_addr[0]`) is used to determine which set of 4 bits in the RAM to overwrite.

Grey code pointers further complicate the write handling. As a Grey code pointer increments, the least significant bit changes according to the following pattern: 0110 (as opposed to 0101 with a binary pointer). Therefore, sometimes `wr_addr[0] == 0` means the lower 4 bits must be written, and sometimes it means the upper 4 bits must be written. This is determined based on the value of `^wr_addr[7:1]`.

Overflow and Underflow Handling

The original implementation in [8] does not detect or handle overflows and underflows. This implementation was modified to allow reads only when the FIFO is not empty (by using `rd_en && !rd_empty` instead of only `rd_en`) and writes only when the FIFO is not full (by using `wr_en && !wr_full` instead of only `wr_full`). The asynchronous pointer comparisons allow for the `rd_empty` output to assert as soon as the FIFO is empty and the `wr_full` output to assert as soon as the FIFO is full; this is required for the prevention of overflows and underflows.

First Word Fall Through Handling

The original implementation in [8] is designed such that the next word to be read is copied to the `rd_data` output 1 cycle after `rd_en` is asserted. First word fall through (FWFT) FIFOs have the next word copied to the `rd_data` output as soon as the first word is written to the FIFO and immediately after the previous word is read. Therefore, valid data is always on the `rd_data` output (unless the FIFO is empty), and asserting `rd_en` indicates that the controlling module is ready for the next data word.

The FWFT logic is implemented by modifying the read enable signal sent to the FIFO memory. The original implementation uses the `rd_en` input to the module as the read enable for the memory. The modified implementation uses `rd_en || async_empty_n`, where `async_empty_n` is a signal that is high when the FIFO is not empty, and low when the FIFO is empty. This signal is updated as soon as an empty FIFO is written, and this signal is also used to generate `rd_empty`. The result is that the data is transferred to `rd_data` at the same time as `rd_empty` is updated, ensuring the correct behavior.

3.28 spreader

3.28.1 Description

This module is part of the TX state machine in the `RFcontroller` module. This module reads packet data out of the `tx_fifo2` module 4 bits at a time, where each set of 4 bits is called a symbol. This module converts each symbol into to a series of 32-bit chips, also referred to as OQPSK codes, outputted serially to the radio circuit, via the `tx_dout` output of the top module. The process of converting symbols to chips is also called spreading. The radio circuit uses these chips to control the frequency of the transmitted signal. This module also indicates when it has finished transmitting the start-of-frame delimiter (SFD) of the packet, and when it has finished transmitting the entire packet. This is used by the `RFcontroller` module to trigger interrupts to the Cortex-M0 or the `RFTIMER` module.

3.28.2 Input/Output Ports

`clk` Input clock.

`resetn` Input reset. This is be connected to the global reset. It is also recommended that this module be reset between packet transmissions.

`tx_dout[3:0]` Input data from the TX FIFO.

`tx_start` Input from the `RFcontroller` module to trigger the symbol-to-chip conversion and transmission.

tx_fifo_empty Input indicating that the FIFO is empty. This indicates that all packet data is transmitted and the transmission is complete.

tx_rd_en Read enable output to the TX FIFO to request more data.

tx_sfd_sent Output pulse indicating that the last bit of the SFD has been sent.

`chip_dout` Serial output stream of chips sent to the radio circuit. This is connected to the radio circuit via the `tx_dout` output of `RFcontroller` and the top module.

`done` Output pulse indicating that the last bit of the packet has been sent.

3.28.3 Design Details

OQPSK vs. MSK Modulation.

The IEEE 802.15.4 standard [15] defines the symbol-to-chip mapping according to a method of modulation called OQPSK, which assigns one code (a set of 32 chips) to each 4-bit symbol. The method of modulation used in the radio for the Single Chip Mote is called MSK. The MSK modulator generates a radio signal equivalent to one generated by an OQPSK modulator.

This module instantiates the `symbol2chips` submodule to convert symbols into two sets of 16 OQPSK chips, called I and Q. I includes all of the even chips in the code, and Q includes all of the odd chips in the code. If a single code has 32 chips,

labeled $c_0, c_1, c_2, ..., c_{30}, c_{31}$, then the I chips are $c_0, c_2, c_4, ..., c_{28}, c_{30}$ and the Q chips are $c_1, c_3, c_5, ..., c_{29}, c_{31}$. Another way to describe this is to say that $i_n = c_{2n}$ and $q_n = c_{2n+1}$. The I chips and Q chips for OQPSK modulation are combined in a particular way to generate 32 chips for MSK modulation.

Suppose the nth symbol in a packet is equivalent OQPSK chips $i_{0,n}...i_{15,n}$ and $q_{0,n}...q_{15,n}$ and MSK chips $m_{0,n}...m_{15,n}$, then the conversion between OQPSK and MSK is:

$$
\begin{aligned}
m_{0,n} &= i_{0,n} \oplus q_{0,n} \\
m_{1,n} &= !(i_{1,n} \oplus q_{0,n}) \\
m_{2,n} &= i_{1,n} \oplus q_{1,n} \\
m_{3,n} &= !(i_{2,n} \oplus q_{1,n}) \\
m_{4,n} &= i_{2,n} \oplus q_{2,n} \\
&... \\
m_{28,n} &= i_{14,n} \oplus q_{14,n} \\
m_{29,n} &= !(i_{15,n} \oplus q_{14,n}) \\
m_{30,n} &= i_{15,n} \oplus q_{15,n} \\
m_{31,n} &= !(i_{0,n+1} \oplus q_{15,n})
\end{aligned}
$$

Note that the last chip of symbol n depends on symbol $n + 1$. The method to convert between OQPSK and MSK is described in more detail in [25].

State Machine

This module uses a state machine used to read symbols from the TX FIFO, convert it to sets of I chips and Q chips, combine I and Q to generate MSK chips, and send each MSK chip serially to the radio circuit in the correct order. The I chips and Q chips are stored in the 16-bit `I` and `Q` registers, which are updated during the appropriate time by the state machine. The `i_ptr` and `q_ptr` registers store pointers to individual bits within the `I` and `Q` registers, which are also incremented at the appropriate time by the state machine. The output, `chip_dout`, is assigned to `(I[i_ptr] ^Q[q_ptr]) ^inv_chip_dout`, where `inv_chip_dout` indicates that the output must be inverted (this signal is also set by the state machine). The overall procedure for converting an entire packet to MSK chips is described by the following states:

`IDLE` Wait for the `tx_start` signal, then go to the `WAIT` state.

`WAIT` After the `tx_start` signal, wait for the TX FIFO to have data (indicated by `!tx_fifo_empty`), then go to the `FIFO_READ` state. Reset `i_ptr` to `4'b1111` and `q_ptr` to `4'b1110`.

`FIFO_READ` Assert the read enable signal for the TX FIFO, increment `q_ptr` from `4'b1110` to `4'b1111`, and go to the `CHIP_CONVERSION_I` state. The data on the output of the FIFO updates during the following cycle. The FIFO output data is connected to the input of the `symbol2chips` module, which converts the data into I and Q chips.

`CHIP_CONVERSION_I` Store the updated value of the I chips onto the `I` register, increment `i_ptr` from `4'b1111` to `4'b0000`, and go to the `CHIP_CONVERSION_Q` state.

`CHIP_CONVERSION_Q` Store the updated value of the Q chips onto the `Q` register, increment `q_ptr` from `4'b1111` to `4'b0000`, and go to the `SHIFT_WAIT_I` state.

`SHIFT_WAIT_I` Increment `i_ptr`. If `i_ptr` is less than `4'b1110`, go to the `SHIFT_WAIT_Q` state. Otherwise, go to the `FIFO_READ` state if there is still data in the FIFO (`!tx_fifo_empty`) or go to the `FINISH_Q` state if the FIFO is empty.

`SHIFT_WAIT_Q` Increment `q_ptr` and go back to the `SHIFT_WAIT_I` state.

`FINISH_Q` Increment `q_ptr` from `4'b1110` to `4'b1111` and go to the `FINISH` state.

`FINISH` Assert the `done` output and go to the `IDLE` state.

This process is implemented in the state machine shown in Figure 3.18. Note that `i_ptr` and `q_ptr` are not reset to zero. Instead, the pointers are reset in order to ensure that the `CHIP_CONVERSION_I` and `CHIP_CONVERSION_Q` states output the 31st and 32nd chip while also updating the `I` and `Q` registers at the appropriate time. In particular, the `CHIP_CONVERSION_Q` state requires $i_{0,n+1}$ and $q_{15,n}$ to generate the proper value for $m_{31,n}$. This requires that `i_ptr` is `4'b0000`, `q_ptr` is `4'b1111`, register `I` is updated right after the `CHIP_CONVERSION_I` state, and register `Q` is updated right after the `CHIP_CONVERSION_Q` state. Working backwards from there determines the default/reset values for `i_ptr` and `q_ptr`.

Also not shown in Figure 3.18 is how the `tx_sfd_sent` output is generated. The `read_count` register keeps track of the number of reads from the FIFO. The `SFD_DONE_CYCLE` local parameter defines the number of reads from the FIFO until *the last bit of the SFD has been sent.* The IEEE 802.15.4 standard [15] indicates that the beginning of a packet is comprised of 8 copies of the `4'b0000` symbol, followed by the two SFD symbols. Therefore, there are 10 symbols in the FIFO that must be read, converted, and transmitted before the `tx_sfd_sent` output is pulsed. However, the last bit of the SFD is sent just after the 11th symbol is read from the FIFO, during the `CHIP_CONVERSION_Q` state. Therefore, the `SFD_DONE_CYCLE` parameter is set to 11, and the `tx_sfd_sent` output is pulsed when `(read_count == SFD_DONE_CYCLE) && (state == CHIP_CONVERSION_Q)`. The `read_count` register stops incrementing after `read_count == SFD_DONE_CYCLE + 1` to avoid wasting energy.

3.29 symbol2chips

3.29.1 Description

This module contains the combinational logic needed in the `spreader` module to convert 4-bit symbols to two 16-bit sets of chips for radio packet data transmission. Each symbol has one set of chips called I and one set called Q. This module uses the `OQPSK_I_CODE` and `OQPSK_Q_CODE` values defined in `chips.vh` to map each symbol to a series of chips.

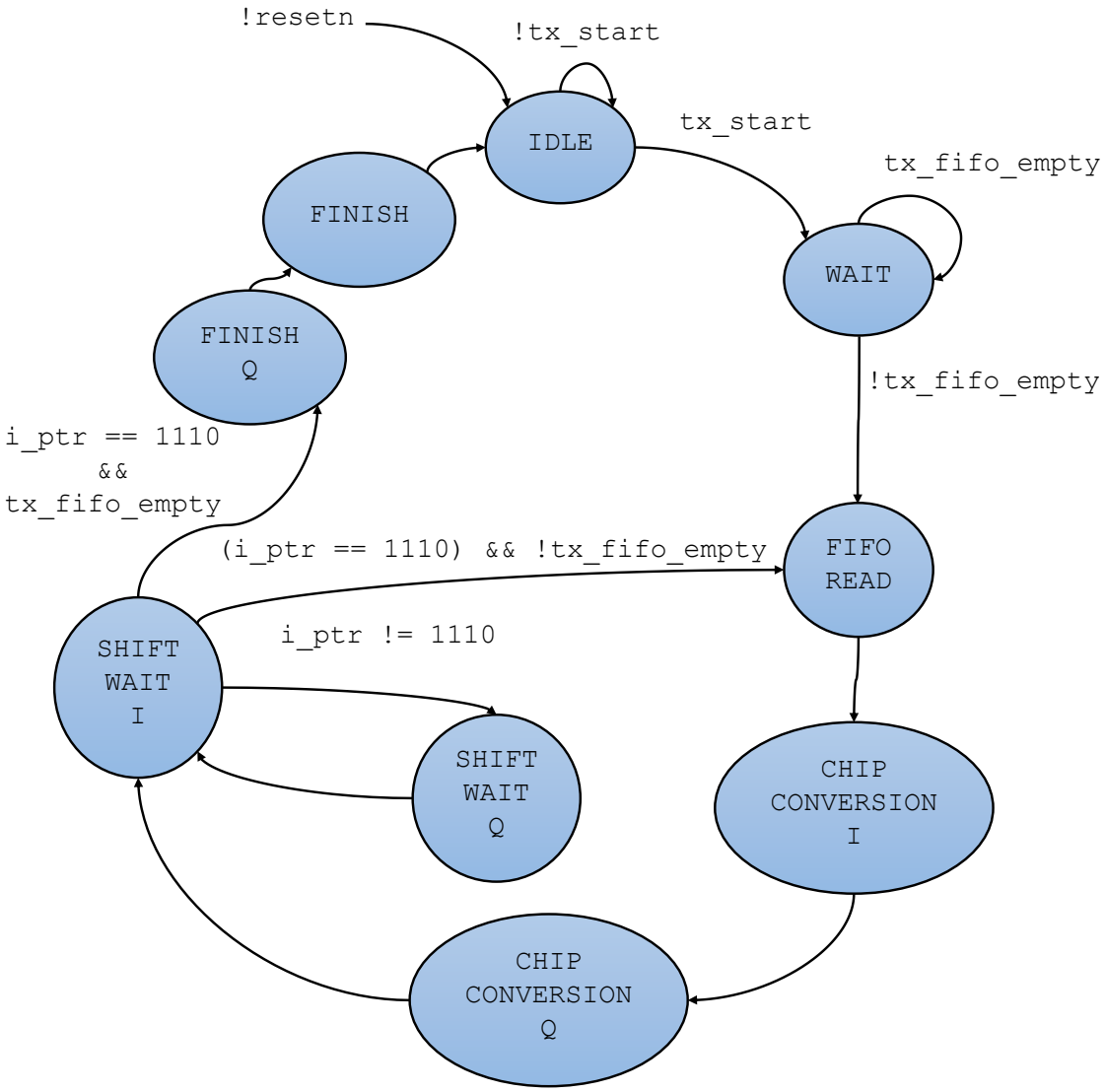


Figure 3.18: Finite State Machine for the `spreader` module

3.29.2 Input/Output Ports

`symbol[3:0]` Input symbol to be converted to chips.

`input_valid` Input indicating that the value on the `symbol` input is valid. When `input_valid` is high, the output is also valid. When `input_valid` is low, the input is ignored and the output corresponds to MSK code for the `4'b0000` symbol.

`I_chips[15:0]` Output containing the `OQPSK_I_CODE` corresponding to the input `symbol`. When `input_valid` is low, the input is ignored and the output corresponds to the chips for the `4'b0000` symbol.

`Q_chips[15:0]` Output containing the `OQPSK_Q_CODE` corresponding to the input `symbol`. When `input_valid` is low, the input is ignored and the output corresponds to the chips for the `4'b0000` symbol.

3.29.3 Design Details

This module implements a case statement used to select the correct OQPSK codes corresponding to the input symbol. The `input_valid` input was added to multiplex the input to the combinational logic inside this module. This prevents any unnecessary switching when the module is not in use.

The IEEE 802.15.4 standard [15] defines the symbol-to-chip mapping for the particular frequency band used by the Single Chip Mote (2450MHz) in section 10.2.4. This section defines a set of 32 chips, $c_0 ... c_{31}$. The set I includes all of the even chips, $c_0, c_2, c_4, ..., c_{28}, c_{30}$, and the set Q includes all of the odd chips, $c_1, c_3, c_5, ..., c_{29}, c_{31}$. Another way to describe this is to say that $i_n = c_{2n}$ and $q_n = c_{2n+1}$. The Verilog file `chips.vh` contains the mapping for each 4-bit symbol to both I and Q.

3.30 corr_despreader

3.30.1 Description

This module is part of the RX state machine in the `RFcontroller` module. This module reads the incoming data from the radio circuit in order to detect a received packet and store the data in the RX FIFO. The incoming data is a series of frequency shifts encoded into a serial binary data stream. These frequency shifts are determined using a demodulation circuit, and the Single Chip Mote uses a method of demodulation called MSK. The bits in the stream are called chips, and sets of 32 chips correspond to 4-bit sets of actual data called symbols. However, since this Single Chip Mote uses MSK demodulation, only the first 31 chips out of every set are used for comparison (see Dr. Osama Khan and Brad Wheeler for an explanation). This module scans the stream of chips to find the start of a packet, then converts the chips to symbols, and stores the symbols in the RX FIFO. This module also indicates when a packet has been detected to the `RFcontroller` module and provides the length of incoming packet to assist the RX state machine.

The process of converting chips to symbols is called despreading. Note that there are only 16 possible symbols, while there are 2^{32} possible combination of chips. Only

16 of those combinations correspond to symbols, and these combinations are referred to as MSK codes. If a packet is transmitted with no errors, then each set of 31 chips matches one of the 16 MSK codes. However, if there is an error in transmission, some of the chips may be changed. Therefore, this module must also take every input set of chips and find the closest matching code. This matching process is called correlation.

3.30.2 Input/Output Ports and Parameters

`clk` Input clock.

`resetn` Input reset.

`rx_din` Serial input stream of chips from the radio circuit. This is connected to the radio circuit via the `rx_din` input of the `RFcontroller` and top modules.

`rx_start` Input from the `RFcontroller` module indicating that this module begin scanning the `rx_din` input to find the beginning of a packet.

`dout[3:0]` Output symbols to be written to the RX FIFO.

`dout_valid` Output indicating that the data on `dout` is valid. This is the write enable input for the RX FIFO.

`packet_detected` Output indicating to the `RFcontroller` module that a packet has been detected. This output also indicates that the data on the `length` output is also valid.

`length[6:0]` Output containing the length of the recently detected packet in bytes.

`THRESHOLD` Parameter used when scanning the input data stream for the beginning of a packet. The beginning of a packet is indicated by a repeating set of symbols/chips called the preamble. This parameter determines how closely a set of chips must match the preamble chips before it is decided that a packet is detected. This parameter is passed into the `correlator` submodule.

3.30.3 Design Details

Correlator

This module instantiates the `correlator` submodule to find the symbol that is the closet match to the current set of chips, as well as quantify the 'closeness' of that match. The 'closeness' of a match is defined as the Hamming Distance, where a smaller Hamming Distance means a closer match, and a Hamming Distance of 0 indicates a perfect match. A minimum Hamming Distance threshold can be used when scanning for the start of a packet to reduce the amount of false-positives. This threshold is defined by the `THRESHOLD` parameter, also passed into the `correlator` submodule. Note that this module accepts 31 chips rather than 32 chips, since the last (most recent) chip cannot be used for correlation.

Packet Detection

The IEEE 802.15.4 standard [15] requires that the beginning of a packet is comprised of 8 copies of the `4'b0000` symbol, known as the preamble, followed by the two start-of-frame delimiter (SFD) symbols. The first SFD symbol (denoted in the code as `STARTSYMBOL0`) is `4'h0101` and the second SFD symbol (denoted in the code as `STARTSYMBOL1`) is `4'h1110`. The combination of the preamble and SFD are used to detect the beginning of a packet.

Finite State Machine

The input chip values are shifted each cycle into a 32-bit shift register. The procedure for detecting and capturing a packet is:

1. Each time a new chip is shifted in, check if the closest matching symbol is `4'b0000`, and the minimum Hamming Distance is below the threshold.
    - If the chips match `4'b0000`, part of the preamble, and the minimum Hamming Distance is below the threshold, wait 32 cycles for new chips to shift in and move on to the next step.
    - If the chips do not match the preamble, or the minimum Hamming Distance is above the threshold, go back to the first step.
2. Check if the set of chips matches the preamble.
    - If the chips match the preamble, wait 32 cycles for new chips to shift in and move on to the next step.
    - If the chips do not match the preamble, go back to the first step.
3. Check if the set of chips matches the preamble or `STARTSYMBOL0`.
    - If the chips match the preamble, wait 32 cycles for new chips to shift in and repeat this step.
    - If the chips match `STARTSYMBOL0`, wait 32 cycles for new chips to shift in and move on to the next step.
    - If the chips do not match the preamble or `STARTSYMBOL0`, go back to the first step.
4. Check if the set of chips matches `STARTSYMBOL1`.
    - If the chips match `STARTSYMBOL1`, then a packet is detected. Wait 32 cycles for new chips to shift in and move on to the next step.
    - If the chips do not match the preamble, go back to the first step.
5. The next two symbols contain the length of the packet in bytes. Store the first symbol, the LSB of the length. Wait 32 cycles.
6. Store the next symbol, the MSB of the length.
7. Use the packet length to determine how many symbols are left in the packet. Continue to wait 32 cycles between each symbol and store each symbol in the RX FIFO until the entire packet has been stored.

This process is implemented in the state machine shown in Figure 3.19. The module stays in the `IDLE` state, where the input from the radio is ignored, until the `rx_start` input is asserted by the `RFcontroller` module. The `SCAN` state is the first step, where the shift register is checked each cycle to see if it matches the preamble. The `PREAMBLE_MATCH1` state is the second step, the `PREAMBLE_MATCH2` state is the third step, and so on. The `max_bit_count` signal in Figure 3.19 indicates when 32 new chips have been shifted into the shift register, and `max_symbol_count` indicates when all of the data in the packet has been copied into the RX FIFO.

Note that this state machine looks for two copies of the preamble symbol and then the beginning of the start symbol. The minimum number of preamble symbols is not adjustable in the current version of the Verilog code. This can be made adjustable by adding a counter and a parameter for the number of preamble symbols to the `PREAMBLE_MATCH2` state. The parameterized counter can check for two or more preamble symbols in a row. To check for one preamble symbol, the `PREAMBLE_MATCH2` state must be removed.

Storing Packet Data Into the FIFO

Each packet uses the two symbols after the SFD to indicate the size of the packet payload, in bytes. The length field is 7 bits wide, since the maximum payload length is 127 bytes. After the length is the packet payload, and after that there are two additional bytes containing the cyclic redundancy check (CRC) value of the packet. Both the payload and the CRC must be stored in the RX FIFO.

The `symbol_count` register stores the number of symbols written into the FIFO. This register increments when the state is `PAYLOAD_CAPTURE` and `max_bit_count == 1`, as this is when 32 new chips have been shifted in. At this point the chips are converted to a symbol and stored into the FIFO. Once `symbol_count` is equal to the number of symbols, minus one, and `max_bit_count == 1`, the last symbol is written to the FIFO and the state transitions back to `IDLE`. The `max_symbol_count` signal in Figure 3.19 is not in the Verilog code, but it represents the code that checks if `symbol_count` is equal to the number of symbols minus one.

Calculating the Number of Symbols

The first symbol after the SFD contains the LSB of the length, `length[3:0]`. The second symbol after the SFD contains the MSB of the length, `length[6:4]`. This length does not include the extra two bytes for the CRC. This means that the number of bytes to copy into the FIFO is `length[6:0] + 2`, and the number of symbols is twice as much.

Given that the value compared to `symbol_count` is the number of symbols minus 1, the following simplifications allow for a small optimization in the size of the register storing this value:

$$NumSymbolsInPacket = 2 \times (length + 2)$$

$$NumSymbolsInPacket - 1 = 2 \times (length + 2) - 1$$

$$NumSymbolsInPacket - 1 = 2 \times (length + 1) + 2 - 1$$

$$NumSymbolsInPacket - 1 = 2 \times (length + 1) + 1$$

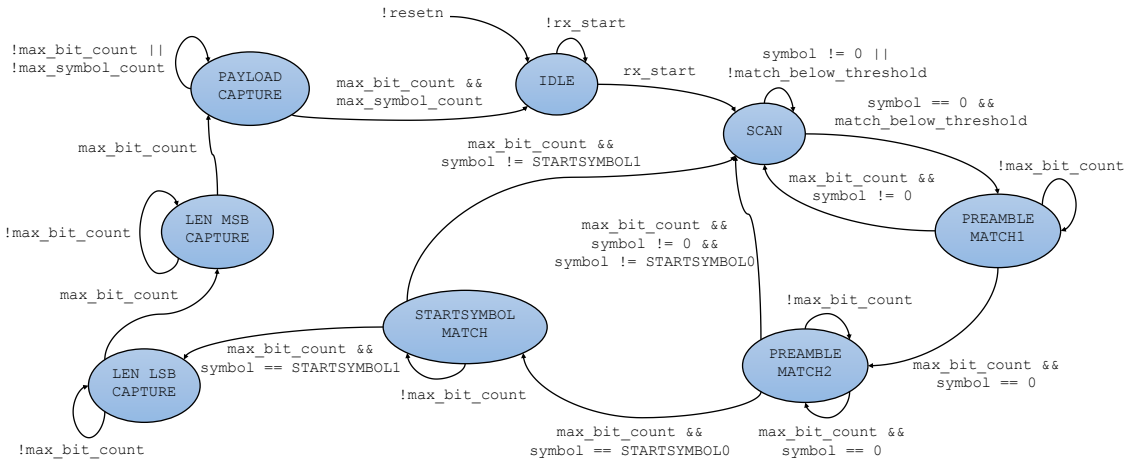


Figure 3.19: Finite State Machine for the `cor_despreader` module

In Verilog, this is the same as adding 1 to `length`, shifting to the left by one, and concatenating a 1 to the end, or `{length+1, 1'b1}`. Since the last bit is always 1, this module instead stores `length+1` in an 8-bit register called `total_symbol_count`, rather than the storing the actual number of symbols in the packet (requiring a 9-bit register). The state transitions from `PAYLOAD_CAPTURE` to `IDLE` when `max_bit_count && (symbol_count == {total_symbol_count, 1'b1})`.

3.31 correlator

3.31.1 Description

This module contains the combinational logic needed in the `corr_despreader` module to find the 4-bit symbol that matches the current set of 31 chips. Each 4-bit symbol has a corresponding set of chips referred to as an MSK code. The Hamming Distance defines how 'close' two binary values are to one another, where a smaller Hamming Distance means a closer match, and a Hamming Distance of 0 means a perfect match. This module calculates the Hamming Distance between the input and each of the 16 possible MSK codes, and returns the symbol corresponding to the code with the minimum Hamming Distance. This module also has the option to check if the minimum Hamming Distance is at or below a threshold specified by a Verilog parameter. This threshold is used when the `corr_despreader` module is scanning for the beginning of a packet.

3.31.2 Input/Output Ports and Parameters

`chips_in[30:0]` 31-bit set of input chips to be correlated with a symbol.

`input_valid` Input indicating that the data on the `chips_in` input is valid. When `input_valid` is high, the output is also valid.

`use_threshold` Input indicating that the `correlator` module must also compare the calculated minimum Hamming Distance to the threshold specified by

the THRESHOLD parameter. The result of this comparison is driven on the match_below_threshold output.

symbol_out[3:0] Output with the closest matching symbol to the chips_in input. When input_valid is low, the input is ignored and the output corresponds to the 4'b0000 symbol.

match_below_threshold Output indicating that the current minimum Hamming Distance is at or below the threshold specified by the THRESHOLD parameter. This output is only valid when use_threshold is high; otherwise, this output is 1.

THRESHOLD Parameter describing the minimum Hamming Distance threshold used when scanning for the beginning of a packet.

3.31.3 Design Details

While the function of this module is seemingly simple, to find the minimum Hamming Distance between the input and a set of 16 codes and return the symbol corresponding to the code with the minimum distance, there is no straightforward method in Verilog to specify this behavior. The Verilog code for this module contains several generate statements to decrease the code size and take advantage of the redundancy in this design. As a result the code can be difficult to understand at first glance without a detailed explanation.

First, note that the MSK codes are all defined in the chips.vh header file. The mapping of symbols to MSK codes was provided by Brad Wheeler, a graduate student researcher working on the radio circuit for the Single Chip Mote. The following code copies these values into an array called MSK_CHIPS, allowing for each individual code to be easily indexed within a generate statement:

```
reg [30:0] MSK_CHIPS[0:15];
always @(*) begin
    MSK_CHIPS[0]  = `MSK_CODE_0;
    MSK_CHIPS[1]  = `MSK_CODE_1;
    MSK_CHIPS[2]  = `MSK_CODE_2;
    MSK_CHIPS[3]  = `MSK_CODE_3;
    MSK_CHIPS[4]  = `MSK_CODE_4;
    MSK_CHIPS[5]  = `MSK_CODE_5;
    MSK_CHIPS[6]  = `MSK_CODE_6;
    MSK_CHIPS[7]  = `MSK_CODE_7;
    MSK_CHIPS[8]  = `MSK_CODE_8;
    MSK_CHIPS[9]  = `MSK_CODE_9;
    MSK_CHIPS[10] = `MSK_CODE_A;
    MSK_CHIPS[11] = `MSK_CODE_B;
    MSK_CHIPS[12] = `MSK_CODE_C;
    MSK_CHIPS[13] = `MSK_CODE_D;
    MSK_CHIPS[14] = `MSK_CODE_E;
    MSK_CHIPS[15] = `MSK_CODE_F;
end
```

The following syntax specifies an array of registers: reg [N-1:0] arr[0:M-1], where the [N-1:0] part indicates that the width of the registers is N bits, and the [0:M-1] part indicates that the number of registers in the array is M. In order to access these registers in Verilog, first the register must be selected, and then the bit(s) within the register are selected:

```
reg [N-1:0] arr [0:M-1];
wire [N-1:0] myreg;
```

```
wire [2:0] mybits;

assign myreg = myarray[3];
assign mybits = myreg[2:0];
```

Several of these register arrays are used for the sections of this code written inside generate statements.

Calculating the Hamming Distance between two binary values is straightforward. The XOR of the two numbers returns another binary number with a 1 for each bit that does not match, and a 0 for every bit that does match. Adding the number of ones in this result yields the Hamming Distance. The `hamming_distance_xor` array holds the 31-bit results of XORing the input with each of the MSK codes. The `hamming_distance_sum` array holds the 5-bit results of adding the individual bits for each register in `hamming_distance_xor`. The values in `hamming_distance_sum` are equal to the Hamming Distance between the input and each of the MSK codes. This value ranges between 0 and 31. The calculation for these two arrays is found in the for loop labeled `calculate_hamming_distance`.

Finding the minimum Hamming Distance, and the corresponding symbol is much more difficult. The method used in this module is similar to performing a binary result on the values stored in the `hamming_distance_sum` array from the bottom-up.

The first step is to compare every two values in `hamming_distance_sum` and find the minimum. This means that `hamming_distance_sum[0]` is compared to `hamming_distance_sum[1]`, `hamming_distance_sum[2]` is compared to `hamming_distance_sum[3]`, and so on. The result of this comparison is eventually used to find the first bit of the `symbol_out` output. The `bit0_comparison_result` register has 8 bits to store the results of this comparison in the following manner:

- If `hamming_distance_sum[0] < hamming_distance_sum[1]`, then set `bit0_comparison_result[0] = 1`.
  Otherwise, set `bit0_comparison_result[0] = 0`.
- If `hamming_distance_sum[2] < hamming_distance_sum[3]`, then set `bit0_comparison_result[1] = 1`.
  Otherwise, set `bit0_comparison_result[1] = 0`.
- Continue following the same pattern until the results of all 8 comparisons are in `bit0_comparison_result`.

At the same time, the minimum Hamming Distances for each comparison are stored in the `bit0_comparison_minimum` arrays to be used for the next set of comparisons:

- If `hamming_distance_sum[0] < hamming_distance_sum[1]`, then set `bit0_comparison_minimums[0] = hamming_distance_sum[0]`. Otherwise, set `bit0_comparison_minimums[0] = hamming_distance_sum[1]`.
- If `hamming_distance_sum[2] < hamming_distance_sum[3]`, then set `bit0_comparison_minimums[1] = hamming_distance_sum[2]`. Otherwise, set `bit0_comparison_minimums[1] = hamming_distance_sum[3]`.
- Continue following the same pattern until the minimum Hamming Distances of all 8 comparisons are in `bit0_comparison_minimums`.

The first set of comparisons, and the assignments to `bit0_comparison_result` and `bit0_comparison_minimums` is found in the for loop labeled `compare_bit0`.

The next step is to compare every two values in `bit0_comparison_minimums`, and store those minimums. This is the same as the process described in the first set of comparisons, only now the `bit0_comparison_minimums` values are being compared, and the results are stored in `bit1_comparison_result` and `bit1_comparison_minimums`. This time there are 4 comparisons in total, and the code is found in the for loop labeled `compare_bit1`.

The third step is to compare every two values in `bit1_comparison_minimums`, and store those minimums. This is the same as the process described in the first set of comparisons, only now the `bit1_comparison_minimums` values are being compared, and the results are stored in `bit2_comparison_result` and `bit2_comparison_minimums`. This time there are 2 comparisons in total, and the code is found in the for loop labeled `compare_bit2`.

The final step is to compare the two values in `bit2_comparison_minimums` and store the result in the `bit3_comparator` register. The following code essentially performs a binary search over the results stored in the `bit3_comparator` and `bitx_comparison_result` registers in order to find the symbol with the minimum distance:

```
assign hamming_distance_minimum_symbol[3] = ~bit3_comparator;
assign hamming_distance_minimum_symbol[2] = ~bit2_comparison_result[
    hamming_distance_minimum_symbol[3]];
assign hamming_distance_minimum_symbol[1] = ~bit1_comparison_result[
    hamming_distance_minimum_symbol[3:2]];
assign hamming_distance_minimum_symbol[0] = ~bit0_comparison_result[
    hamming_distance_minimum_symbol[3:1]];
```

Finally, the following code assigns `symbol_out`, and checks if the minimum distance is at or below the threshold:

```
assign minimum_distance = hamming_distance_sum[hamming_distance_minimum_symbol];
assign match_below_threshold = use_threshold ? (minimum_distance <= THRESHOLD) : 1'
    b1;
assign symbol_out = hamming_distance_minimum_symbol;
```

3.32 bit_sync

3.32.1 Description

This module synchronizes a signal from a slower clock domain to a faster one. In particular, the fast clock samples the signal from the slow clock, and outputs a single-cycle (according to the fast clock) pulse when a rising edge is detected.

This module is used in the `RFcontroller` module to synchronize the `tx_sfd_sent` and `tx_spread_done` signals from the `spreader` module. These signals indicate that the last bit of a packet's SFD and the last bit of a packet has finished transmitting, respectively.

3.32.2 Input/Output Ports

`reset_n` Input reset.

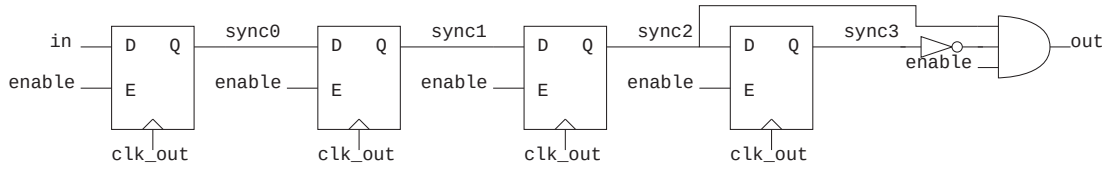


Figure 3.20: Schematic of the `bit_sync` module

`enable` Enable input for the entire synchronizer. Inputs are only be sampled when this input is asserted. The output remains low unless this input is asserted.

`in` Input signal to be sampled. This input must be aligned to a clock slower than `clk_out`.

`clk_out` Input sampling clock. This clock is used to sample the input, `in`. The output pulse, `out`, is aligned to this clock.

`out` Edge detected pulse output. This output is a single-cycle pulse (aligned with `clk_out`) indicating that an edge in the `in` input is detected.

3.32.3 Design Details

This synchronizer is designed using a shift register of depth 4 to sample the input. A long shift register is used to reduce the probability of metastability at the output. The last two bits in the shift register are compared to determine if there is a rising edge in the input. The shift register does not sample new values unless the `enable` input is asserted, and the output remains low unless the `enable` input is asserted. A schematic of this circuit is shown in Figure 3.20.

3.33 bus_sync

3.33.1 Description

This module synchronizes a bus from a slower clock domain to a faster one. In particular, this module behaves somewhat like a FIFO with a depth of 1. The slow clock writes to the `bus_sync` module. The written data is copied into another register using the fast clock, and a valid signal is asserted. The faster clock then asserts its read enable signal to indicate that it read the data, and the valid signal is de-asserted.

This module is used in the `RFcontroller` module to synchronize the length of a packet being received from the `corr_despreader` module. The valid signal of this module is used by the `RFcontroller` both to indicate that the `corr_despreader` module has detected an incoming packet, and it has decoded the length of that packet.

3.33.2 Input/Output Ports and Parameters

`reset_n` Input reset.

`clk` Input data clock. The input data, `din`, and the write enable, `wr_en`, are aligned to this clock.

`sync_clk` Input sampling clock. This clock is used to sample the input data. The output data, `dout`, and the valid output, `valid`, are aligned to this clock. The write enable input, `wr_en`, must also be aligned to this clock.

`sync_en` Enable input for the entire synchronizer. Both `wr_en` and `rd_en` are ignored if this input is low. This input must be stable before attempting to write and remain stable as long as this synchronizer is in use.

`wr_en` Write enable input. This input must be aligned with `clk`. When this input is high, the input data, `din` is stored and is eventually written to the data output, `dout`.

`rd_en` Read enable input. This input must be aligned with `sync_clk`. This input indicates that the data on `dout` has been read. The `valid` output de-asserts after a read.

`din[BUS_LENGTH-1:0]` Input data to be synchronized. This input must be aligned with `clk`.

`dout[BUS_LENGTH-1:0]` Synchronized output data. This output is aligned with `sync_clk`.

`valid` Data valid output. Indicates that the data on `dout` is valid. This output is aligned with `sync_clk`.

`BUS_LENGTH` Parameter describing the width of the data buses.

3.33.3 Design Details

This module ignores all inputs as long as the `sync_en` input is low. When both `sync_en` and `wr_en` are asserted, this module samples the data input, `din`, onto a register using the input clock `clk`. The `wr_en` input itself is also sampled onto a register using `clk`. The registered version of `wr_en` is then sampled into a shift register with a depth 3 using `sync_clk`. This shift register is used to reduce the probability of metastability. The last two bits are compared to determine if there is a rising edge. This rising edge indicates that data was previously sampled from `din`, and that the registered value is currently stable. This rising edge is used to copy the registered `din` data into another register using `sync_clk`. At the same time, the register with the `valid` output is set. This register is cleared by asserting the `rd_en` input. Throughout this entire process the `sync_en` input must be high. A schematic of this circuit is shown in Figure 3.21.

This synchronizer design may fail if there are multiple writes in succession, as there is no check to see if the value from a previous write is read before writing again. However, this module is used in a context where multiple writes are not possible. This module is used to only synchronize one signal between two clock domains on a single occasion, and then is reset.

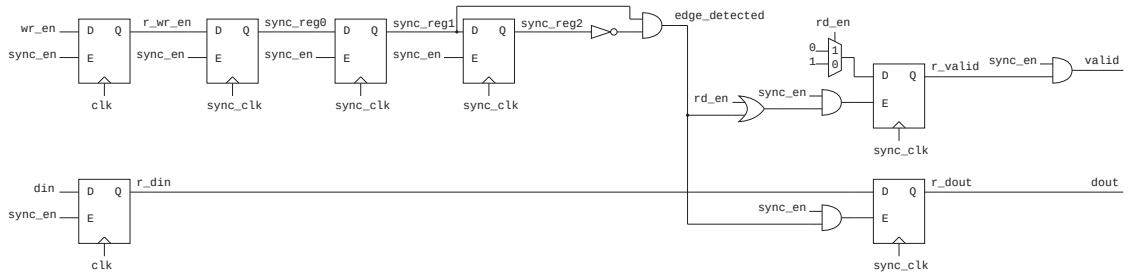


Figure 3.21: Schematic of the `bus_sync` module

3.34 crcParallel

3.34.1 Description

This module is used by the `RFcontroller` module to calculate the cyclic redundancy check (CRC) value of a radio packet conforming to the IEEE 802.15.4 standard. This standard uses CRC to detect any errors in a received packet.

The theory behind the CRC calculation may be relatively confusing for those without any experience in error detection and correction; however, its purpose and use is straightforward. The CRC is the result of a mathematical function over the entire payload data of a packet. On the transmission side, a 16-bit CRC is calculated over the payload (up to 127 bytes) and appended to end of the packet. On the receive side, the 16-bit CRC is calculated over the entire payload *and* the additional CRC bits appended to the end of the packet. If the result on the receive end is not 0, then there is an error in the packet payload and the data should be discarded.

Typical implementations calculate the CRC input 1 bit of the payload at a time. Such a module requires that the payload data be clocked in serially, from LSB to MSB. However, all of the operations in the `RFcontroller` module deal with bytes rather than individual bits, and serializing the data for this calculation slows down the overall packet transmission process. Therefore, this implementation was designed to calculate the CRC using inputs of 1 byte instead of 1 bit. The packet data is clocked in 1 byte at a time, from the first byte in the packet (the first byte transmitted) to the last byte in the packet.

For more information on the CRC calculation, see section 5.2.1.9 of the IEEE 802.15.4 standard [15]. A copy is found in `scm-digital/doc/`.

3.34.2 Input/Output Ports

`HCLK` Input clock.

`HRESETn` Input reset. This input is used for the main system reset.

`local_rst` Input reset from the `RFcontroller`. This input is used when the `RFcontroller` module clears the CRC value between computations. The CRC must be reset before each computation in order to produce the correct result.

`sample_en` Input indicating that the input is valid and the CRC value must update.

`in[7:0]` Data input. This is the packet payload data.

`crc[15:0]` CRC output. This is the CRC calculated over all of the previous inputs since the module was reset. This value is not valid until the entire packet payload is inputted to this module.

3.34.3 Design Details

Typical CRC implementations use a linear feedback shift register (LFSR) to calculate the CRC 1 bit at a time. This is the method used in section 5.2.1.9 of the IEEE 802.15.4 standard [15].

This module instead calculates the CRC using an 8-bit input by following the method described in [30]. The design process involves creating a reference serial implementation (such as an LFSR), and recording the output given a series of particular inputs. These results can be used to determine the output / next state as a function of the 16-bit current state and the 8-bit input.

A testbench was created to compare the results of this module with the serial reference implementation. A large number of random inputs were generated and the outputs were compared for every 16 bits of input. This module matched the serial result in each comparison.

For more information on how to create a parallel CRC calculation using a serial reference, see [30]. A copy is found in `scm-digital/doc/`.

3.35 RFTIMER

3.35.1 Description

This module is a special-purpose timer designed to interface with the `RFcontroller` module. The timer itself is a counter (with a parameterized width) connected to the `CLK_RFTIMER` clock with a frequency of 500kHz. There are 8 compare units (the number of compare units is parameterized) that generate an interrupt to the Cortex-M0 when the counter matches the value stored in the compare unit. The compare units also have the option of sending a trigger to the `RFcontroller` module. There are also 4 capture units (the number of capture units is parameterized) that capture the value of the timer when it receives either a signal from the Cortex-M0 or an interrupt from the `RFcontroller` module.

With the correct combination and configuration of compare and capture units, the Single Chip Mote is able to send packets or listen for incoming packets at specific times without any intervention from the software on the Cortex-M0 beyond the initial setup. This module is also suitable as a timer for purposes other than sending or receiving packets.

3.35.2 Input/Output Ports and Parameters

`HRESETn` Input reset.

`HCLK` System clock input.

`TCLK` Timer clock input.

HSEL Slave select input.

HWRITE Write select input.

HTRANS[1] Transfer type input.

HADDR[31:0] Address input.

HWDATA[31:0] Write data input.

HRDATA[31:0] Read data output.

HREADY Transfer finished input. This input indicates that the previous transfer on the bus has finished and that address phase signals must be latched.

HREADYOUT Transfer finished output.

tx_load_done_in Input from the RFcontroller module to the capture units for the TX_LOAD_DONE interrupt.

tx_sfd_done_in Input from the RFcontroller module to the capture units for the TX_SFD_DONE interrupt.

tx_send_done_in Input from the RFcontroller module to the capture units for the TX_SEND_DONE interrupt.

rx_sfd_done_in Input from the RFcontroller module to the capture units for the RX_SFD_DONE interrupt.

rx_done_in Input from the RFcontroller module to the capture units for the RX_DONE interrupt.

tx_load_trigger_out Output to the RFcontroller module from the compare units for the TX_LOAD trigger.

tx_send_trigger_out Output to the RFcontroller module from the compare units for the TX_SEND trigger.

rx_start_trigger_out Output to the RFcontroller module from the compare units for the RX_START trigger.

rx_stop_trigger_out Output to the RFcontroller module from the compare units for the RX_STOP trigger.

NUM_COMPARE_UNITS Parameter describing the number of compare units.

NUM_CAPTURE_UNITS Parameter describing the number of capture units.

COUNTER_WIDTH Parameter describing the width of the counter used for the timer.

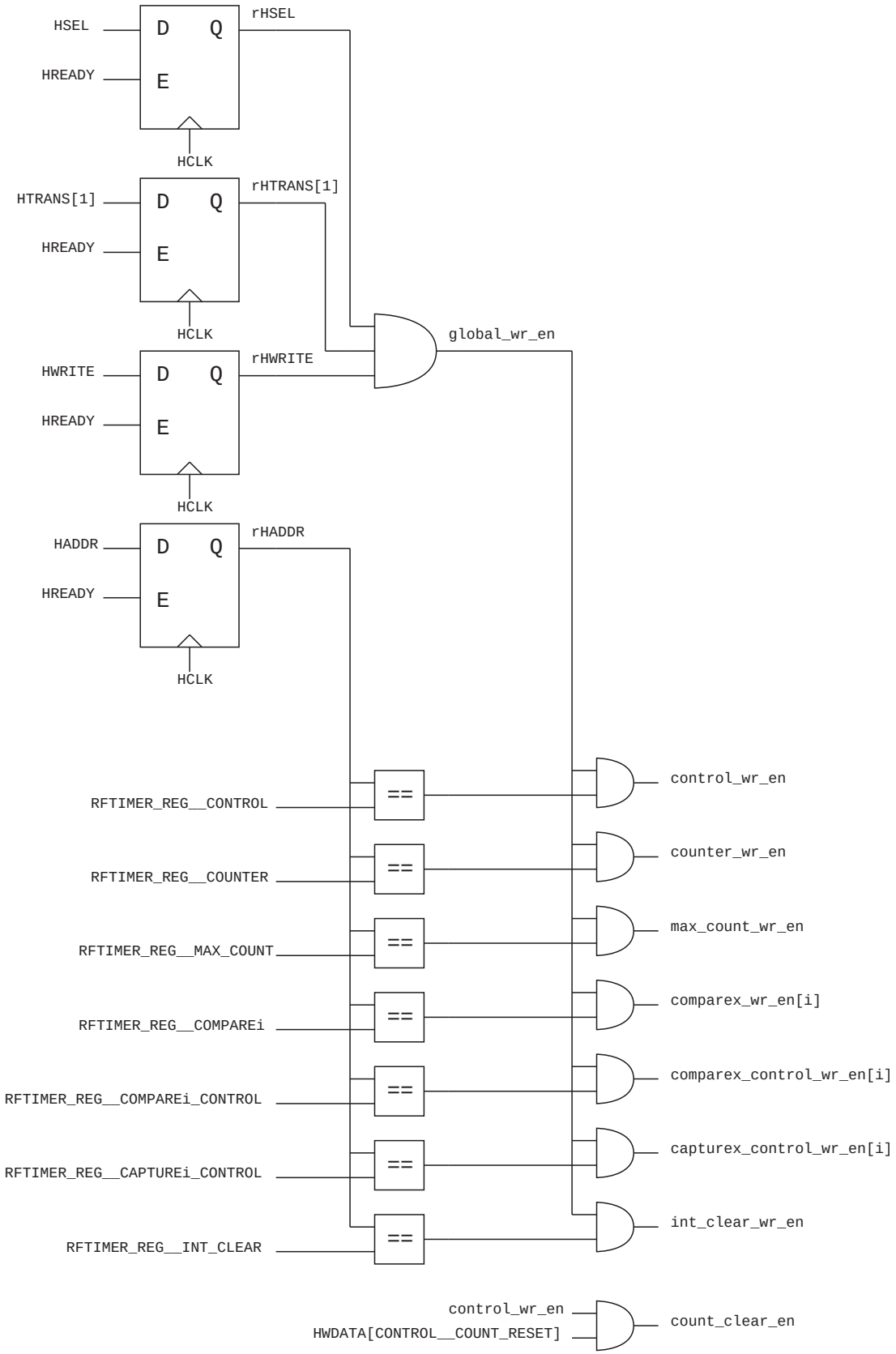


Figure 3.22: Register enable signals from the AHB for the RFTIMER module.

3.35.3 Design Details

Clocking Assumptions

In order for this module to function correctly, the clock frequency of the timer f_{timer} must be a division of the system clock frequency f_{sys} such that $f_{sys} = N \times f_{timer}$, where N is an integer. In the case of the Single Chip Mote digital system on an FPGA, the system clock `HCLK`, has a frequency of 5MHz, and the timer clock, `CLK_RFTIMER` (connected to the `TCLK` input on this module), has a frequency of 500kHz. The rising edge of the timer clock must be aligned to the rising edge of the system clock. In general this is achieved by having the timer clock and the system clock originate from the same oscillator and using feedback to keep their relative phases in alignment. There are no synchronization issues as long as these requirements are met.

Synchronization

The timer operates in the slower timer clock domain, and the AHB bus (that sets the configuration registers) operates in the faster system clock domain. If the relationship between these two clocks is unknown, then the `RFTIMER` module would require synchronizers to pass data between the two clock domains. However, by requiring that the timer clock and system clock are aligned in phase, the following assumptions are valid:

- Registers in the timer clock domain only change their value on the rising edge of the timer clock. Since this coincides with the rising edge of the system clock, the output of those registers are also only changing on the rising edge of the system clock, and are stable at all other times. This means that any logic in the system clock domain can use the output of a register in the timer clock domain without a synchronizer. Therefore, the AHB can directly read any registers in the `RFTIMER` module that are connected to `TCLK`.

- Logic in the timer clock domain relying on configuration registers in the system clock domain must not be connected directly to the output of those registers. It is possible for registers in the system clock domain to change multiple times during a single timer clock cycle, leading to erratic behavior.

- Configuration registers in the system clock domain are safely sampled by registers in the timer clock domain by directly connecting the output of the former to the input of the latter. The data in the configuration register may change many times in a single timer clock cycle, but only the last value is written into the sampling register. The timing tools use the defined phase relationship between the two clock domains to ensure that there are no setup and hold time violations. As a result, the paths between the registers are laid out such that the sampling register does not sample when the input is changing and there is little chance of metastability.

Therefore, it is possible to connect signals from between the two clock domains without much overhead. Connecting signals from the system clock domain to the timer clock domain first requires directly sampling the signals in the timer clock domain and then using those sampled outputs in the timer clock domain logic. On

the other hand, signals from the timer clock domain are directly connected to logic in the system clock domain. More robust synchronization logic is not necessary.

In the Verilog code for this module, all register names with the `_t` suffix indicate that the register is connected to `TCLK`. All register names with the `_h` suffix indicate that the register is connected to `HCLK`. A common pattern repeated throughout this design is the use of a `_h` register written via the AHB and an accompanying `_t` register sampling its value on every rising edge of `TCLK`.

Counter

The timer in the `RFTIMER` module is a counter (with a width of `COUNTER_WIDTH`) that increments by 1 on the rising edge of `TCLK`. The counter value can also be read or written via the `RFTIMER_REG__COUNT` register. If at any time the value in this register is greater than or equal to the maximum counter value, stored in the `RFTIMER_REG__MAX_COUNT` register, the counter will roll over back to 0 on the next rising edge of `TCLK`. This counter is enabled or disabled using the `ENABLE` bit of the `RFTIMER_REG__CONTROL` register, and is reset to 0 using the `COUNT_RESET` bit in the same register.

Figure 3.23 contains a conceptual schematic and the actual Verilog code for the `RFTIMER_REG__COUNT` register (`count_t`), the `RFTIMER_REG__MAX_COUNT` register (`max_count_t`), and the `ENABLE` bit of the `RFTIMER_REG__CONTROL` register (`counter_en_t`). The write enable signals are shown in Figure 3.22. `max_count_t` and `counter_en_t` sample their values from the `max_count_h` and `counter_en_h` registers written by the AHB. The `count_t` register samples its value from `count_h`.

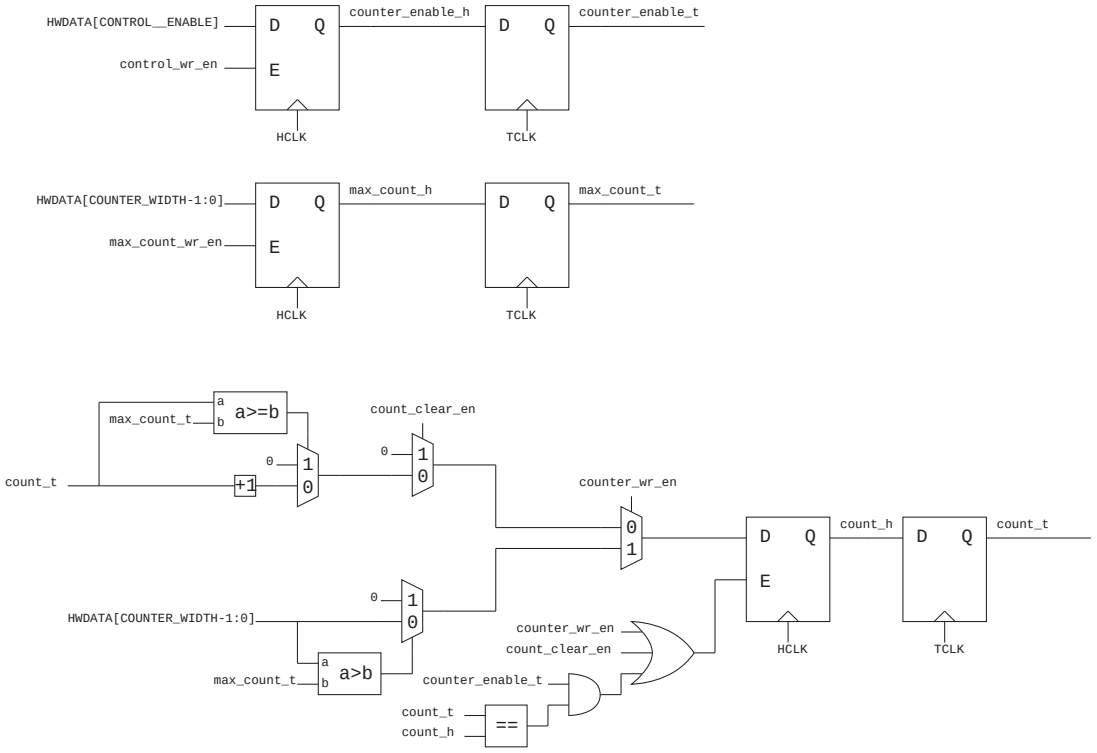
While the logic at the input to `count_h` appears complicated, its function is more straightforward. `count_h` always contains the next value of `count_t`. The next value of `count_t` is either written to a specific value via the AHB, cleared to 0 via the AHB, or set to `count_t + 1` when the counter is enabled.

When `count_h` is overwritten or cleared via the AHB, `count_t` will reflect that change on the next rising edge of `TCLK`. If there are no AHB writes, `count_h` is set to `count_t + 1` whenever `count_h == count_t`. When `count_t` updates, the condition `count_h == cont_t` is true, and therefore `count_h` increments, and again it stores the next value of `count_t`. If the timer is disabled (`counter_wr_en == 0`), then `count_h` will not update when `count_h == count_t` (although it will update when there is an AHB write since AHB writes must still be take effect when the counter is disabled). There is additional logic to ensure that the value being written to `count_h` is not larger than `max_count_t`.

Compare Units

This module has multiple compare units, specified by the `NUM_COMPARE_UNITS` parameter, that store a value to compare against the counter. If the compare value matches the counter, then the compare unit has the option to set a bit in the `RFTIMER_REG__INT` register to trigger the Cortex-M0 interrupt, or send a trigger to the `RFcontroller` module.

Each compare unit has a compare register, `RFTIMER_REG__COMPAREi`, and a control register, `RFTIMER_REG__COMPAREi_CONTROL` (where the i represents the index of the compare unit, ranging from 0 to `NUM_COMPARE_UNITS-1`). The compare unit




```
// Counter Enable Register
always @(posedge HCLK or negedge HRESETn) begin
     if (!HRESETn) counter_enable_h <= 1'b0;
     else if (control_wr_en) counter_enable_h <= HWDATA[CONTROL__ENABLE];
end
always @(posedge TCLK or negedge HRESETn) begin
     if (!HRESETn) counter_enable_t <= 1'b0;
     else counter_enable_t <= counter_enable_h;
end

// Max Count Register
always @(posedge HCLK or negedge HRESETn) begin
     if (!HRESETn) max_count_h <= {COUNTER_WIDTH{1'b1}};
     else if (max_count_wr_en) max_count_h <= HWDATA[COUNTER_WIDTH-1:0];
end
always @(posedge TCLK or negedge HRESETn) begin
     if (!HRESETn) max_count_t <= {COUNTER_WIDTH{1'b1}};
     else max_count_t <= max_count_h;
end

// Counter
always @(posedge HCLK or negedge HRESETn) begin
   if (!HRESETn) count_h <= 0;
   else if (counter_wr_en)
        if (HWDATA[COUNTER_WIDTH-1:0] > max_count_t) count_h <= 0;
        else count_h <= HWDATA[COUNTER_WIDTH-1:0];
   else if (count_clear_en) count_h <= 0;
   else if ((count_h == count_t) && counter_enable_t)
        if (count_t >= max_count_t) count_h <= 0;
        else count_h <= count_t + 1;
end
always @(posedge TCLK or negedge HRESETn) begin
   if (!HRESETn) count_t <= 0;
   else count_t <= count_h;
end
```

Figure 3.23: Schematic and Verilog code for the `counter` register, the `max_count` register, and the `counter_enable` register for the `RFTIMER` module.

is implemented in the compare_unit submodule, with connections to the RFTIMER module for the AHB interface and the interrupts.

In order to parameterize the module for different numbers of compare units, each compare_unit submodule is instantiated inside a generate statement. The following buses and arrays of buses are used to connect to the compare_unit modules within the generate statement based on their index i:

comparex_wr_en Bit i in this bus connects to the write enable signal for the RFTIMER_REG__COMPAREi register.

comparex_control_wr_en Bit i in this bus connects to the write enable signal for the RFTIMER_REG__COMPAREi_CONTROL register.

comparex_interrupt_en Bit i in this bus connects to the interrupt enable output for compare unit i, used for the Cortex-M0 interrupt.

comparex_match Bit i in this bus connects to the compare match output for compare unit i, used for the Cortex-M0 interrupt.

compare Bus i in this array of buses connects to the output of the RFTIMER_REG__COMPAREi register.

compare_control Bus i in this array of buses connects to the output of the RFTIMER_REG__COMPAREi_CONTROL register.

tx_load_trigger_bus Bit i in this bus connects to the TX_LOAD trigger output from compare unit i.

tx_send_trigger_bus Bit i in this bus connects to the TX_SEND trigger output from compare unit i.

rx_start_trigger_bus Bit i in this bus connects to the TX_START trigger output from compare unit i.

rx_stop_trigger_bus Bit i in this bus connects to the RX_STOP trigger output from compare unit i.

For more information on the implementation of the compare units, see section 3.36.

Capture Units

This module has multiple capture units, specified by the NUM_CAPTURE_UNITS parameter, that store the value of the counter into a register when triggered by the Cortex-M0 via the AHB, or when triggered by an interrupt from the RFcontroller module. This module also has the option to set a bit in the RFTIMER_REG__INT register to trigger the Cortex-M0 interrupt after a capture.

Each capture unit has a capture register, RFTIMER_REG__CAPTUREi, and a control register, RFTIMER_REG__CAPTUREi_CONTROL (where the i represents the index of the capture unit, ranging from 0 to NUM_CAPTURE_UNITS-1). The capture unit is implemented in the capture_unit submodule, with connections to the RFTIMER module for the AHB interface and the interrupts.

In order to parameterize the module for different numbers of capture units, each `capture_unit` submodule is instantiated inside a generate statement. The following buses and arrays of buses are used to connect to the `capture_unit` modules within the generate statement based on their index i:

`capturex_control_wr_en` Bit i in this bus connects to the write enable signal for the `RFTIMER_REG__CAPTUREi_CONTROL` register.

`capturex_interrupt_en` Bit i in this bus connects to the interrupt enable output for capture unit i, used for the Cortex-M0 interrupt.

`capturex_trigger` Bit i in this bus connects to the capture trigger output for capture unit i, used for the Cortex-M0 interrupt.

`capture` Bus i in this array of buses connects to the output of the `RFTIMER_REG__CAPTUREi` register.

`capture_control` Bus i in this array of buses connects to the output of the `RFTIMER_REG__CAPTUREi_CONTROL` register.

For more information on the implementation of the capture units, see section 3.37.

Interrupts from the `RFcontroller` Module

When configured to do so, the `RFcontroller` module sends single-cycle (synchronized with `HCLK`) pulses to the `RFTIMER` module to trigger a capture on any of the capture units. These pulses come from the `tx_load_done_in`, `tx_sfd_done_in`, `tx_send_done_in`, `rx_sfd_done_in`, and `rx_done_in` inputs to the `RFTIMER` module, and are connected directly to each capture unit. The capture units are only triggered when a particular input is enabled in the `RFTIMER_REG__CAPTUREi_CONTROL` register. For more information on the events that trigger these pulses, see section 3.25.

Triggers to the `RFcontroller` Module

When configured to do so, the `RFTIMER` module sends single-cycle (synchronized with `TCLK`) pulses during a compare match to the `RFcontroller` module. These pulses trigger the state machines controlling the sending and receiving of packets. The compare units only generate a pulse when a particular output is enabled in the `RFTIMER_REG__COMPAREi_CONTROL` register. Each compare unit has four outputs for the four types of triggers: `tx_load_trigger`, `tx_send_trigger`, `rx_start_trigger`, `rx_stop_trigger`. The outputs to the `RFcontroller` module from the `RFTIMER` module are the bitwise OR of the outputs from each compare unit.

Cortex-M0 Interrupt

This module has one interrupt to the Cortex-M0, through the `rftimer_irq` output. This output is the bitwise OR of the bits in the `RFTIMER_REG__INT` register, synchronous with `HCLK`. Each bit in the `RFTIMER_REG__INT` corresponds to an interrupt

from a compare or capture unit, and is only be set if interrupts are globally enabled by setting the INTERRUPT_ENABLE bit of the RFTIMER_REG__CONTROL register. Each bit in this register is cleared by setting the same bit in the RFTIMER_REG__INT_CLEAR register.

The RFTIMER_REG__INT register is a concatenation of three separate registers: comaprex_int, capturex_int, and capturex_overflow. The comparex_int register has 1 bit for each compare unit, and the capturex_int and capturex_overflow registers have 1 bit for each capture unit.

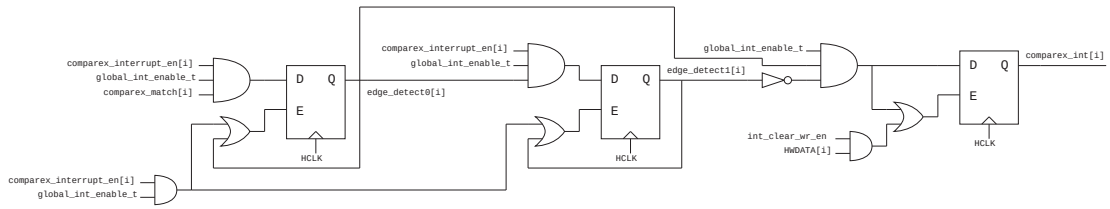
A bit in the comparex_int register is set when the following conditions are true: there is a compare match, interrupts are globally enabled, and the INTERRUPT_ENABLE bit of the RFTIMER_REG__COMPAREi_CONTROL (assigned to comparex_interrupt_en[i]) is set. A compare match is indicated by the comparex_match[i] signal, a single-cycle pulse synchronous with TCLK. In order to prevent this pulse from writing to the interrupt register multiple times (since the comparex_int register is synchronous with HCLK), an edge detection circuit is used to sample the comparex_match[i] signal and set comparex_int[i] on the rising edge. A conceptual schematic of this edge detection circuit and the comparex_int[i] register is shown in Figure 3.24 along with the actual Verilog code. The two edge_detect registers are designed to clear themselves if the interrupt is disabled.

A bit in the capturex_int register is set when the following conditions are true: a capture is triggered, interrupts are globally enabled, and the INTERRUPT_ENABLE bit of the RFTIMER_REG__CAPTUREi_CONTROL (assigned to capturex_interrupt_en[i]) is set. A capture trigger is indicated by the capturex_trigger[i] signal, a single-cycle pulse synchronous with HCLK. If the capturex_int[i] bit is not cleared before the next capture event, the corresponding bit in the capturex_overflow register is set instead. A conceptual schematic of the capturex_int[i] and capturex_overflow[i] registers is shown in Figure 3.25, along with the actual Verilog code.

3.35.4 Register Interface

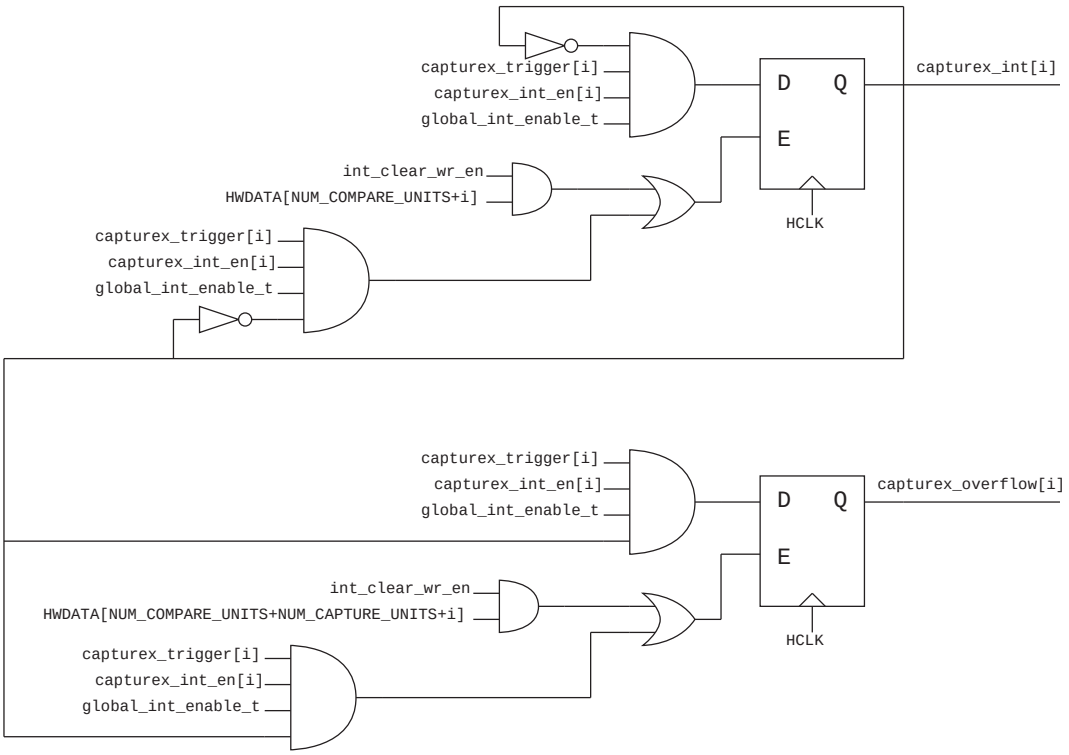
Control Register

The counter is controlled using the RFTIMER_REG__CONTROL register. This register has three bits, ENABLE, INTERRUPT_ENABLE, and COUNT_RESET. Setting the ENABLE bit of this register causes the counter to increment during each timer clock cycle that the ENABLE bit is set to 1. The counter does not increment if the ENABLE bit is set to 0, and continues to increment from the current value of RFTIMER_REG__COUNT when re-enabled. The INTERRUPT_ENABLE bit is a global enable signal for all of the compare/capture interrupts. Setting the COUNT_RESET bit resets the value of the counter back to 0 on the next timer clock (TCLK) cycle. While it is also possible to reset the timer by writing a 0 to the RFTIMER_REG__COUNT register, the COUNT_RESET bit allows the software to reset and enable the timer in a single register write access. If multiple changes are made to the RFTIMER_REG__CONTROL register during a single timer clock (TCLK) cycle, only the last change takes effect on the next rising edge of the timer clock.




```
// Compare Interrupts
generate
for (i = 0; i < NUM_COMPARE_UNITS; i = i + 1) begin : compare_interrupts
    always @(posedge HCLK or negedge HRESETn) begin
        if (!HRESETn) edge_detect0[i] <= 1'b0;
        else if (global_int_enable_t && comparex_interrupt_en[i]) edge_detect0[i]
            <= comparex_match[i];
        else if (edge_detect0[i]) edge_detect0[i] <= 1'b0;
    end
    always @(posedge HCLK or negedge HRESETn) begin
        if (!HRESETn) edge_detect1[i] <= 1'b0;
        else if (global_int_enable_t && comparex_interrupt_en[i]) edge_detect1[i]
            <= edge_detect0[i];
        else if (edge_detect1[i]) edge_detect1[i] <= 1'b0;
    end
    always @(posedge HCLK or negedge HRESETn) begin
        if (!HRESETn) comparex_int[i] <= 1'b0;
        else if (global_int_enable_t && edge_detect0[i] && !edge_detect1[i])
            comparex_int[i] <= 1'b1;
        else if (int_clear_wr_en && HWDATA[i]) comparex_int[i] <= 1'b0;
    end
end
endgenerate
```

Figure 3.24: Schematic and Verilog code for the `comparex_int[i]` register for the `RFTIMER` module.




```
// Capture Interrupts
generate
for (i = 0; i < NUM_CAPTURE_UNITS; i = i + 1) begin : capture_interrupts
    always @(posedge HCLK or negedge HRESETn) begin
        if (!HRESETn) capturex_int[i] <= 1'b0;
        else if (global_int_enable_t && capturex_trigger[i] &&
            capturex_interrupt_en[i] && !capturex_int[i]) capturex_int[i] <= 1'b1;
        else if (int_clear_wr_en && HWDATA[NUM_COMPARE_UNITS+i]) capturex_int[i] <=
             1'b0;
    end
    always @(posedge HCLK or negedge HRESETn) begin
        if (!HRESETn) capturex_overflow[i] <= 1'b0;
        else if (global_int_enable_t && capturex_trigger[i] &&
            capturex_interrupt_en[i] && capturex_int[i]) capturex_overflow[i] <= 1'
            b1;
        else if (int_clear_wr_en && HWDATA[NUM_COMPARE_UNITS+NUM_CAPTURE_UNITS+i])
            capturex_overflow[i] <= 1'b0;
    end
end
endgenerate
```

Figure 3.25: Schematic and Verilog code for the `capturex_int[i]` and `capturex_overflow[i]` registers for the `RFTIMER` module.

Counter and Max Count Registers

The counter value is stored on the RFTIMER_REG__COUNTER register, with a width equal to the COUNTER_WIDTH parameter. The RFTIMER_REG__MAX_COUNT register contains the maximum value of the counter before it rolls over to 0. When the timer is enabled, the counter value increments by 1 with each rising edge of the timer clock (TCLK). The RFTIMER_REG__COUNTER register can also be written by the software, including when the timer is enabled. When the counter reaches the value in the RFTIMER_REG__MAX_COUNT register, it rolls over back to 0 on the next rising edge. If RFTIMER_REG__MAX_COUNT is changed when the counter value is greater than or equal to the new RFTIMER_REG__MAX_COUNT value, then the counter rolls over to 0 on the next rising edge. If multiple changes are made to the RFTIMER_REG__COUNTER and RFTIMER_REG__MAX_COUNT registers during a single timer clock (TCLK) cycle, only the last change takes effect on the next rising edge of the timer clock.

Compare Unit i Registers

The RFTIMER_REG__COMPAREi register, with width equal to the COUNTER_WIDTH parameter, contains the value that is compared to the counter in compare unit i. Each compare unit, when configured to do so, generates an interrupt to the Cortex-M0 on the next rising edge of the system clock (HCLK) after the counter value matches the value stored in the RFTIMER_REG__COMPAREi register. Each compare unit also has the option of sending a trigger to the RFcontroller module with a single-cycle pulse (according to TCLK) when the counter matches the value stored in the RFTIMER_REG__COMPAREi register.

The RFTIMER_REG__COMPAREi_CONTROL register contains six bits to enable the compare unit and configure its interrupts and triggers. The ENABLE bit must be set in order for the compare unit to generate an interrupt or trigger. The other five bits enable or disable the individual interrupts and triggers.

The compare unit is capable of generating five different types of interrupts or triggers in any combination. The first is an interrupt to the Cortex-M0, enabled by setting the INTERRUPT_ENABLE bit of the RFTIMER_REG__COMPAREi_CONTROL register. This interrupt causes the corresponding bit in the RFTIMER_REG__INT register to be set. The other four are triggers for the RFcontroller module. These triggers are enabled by setting the TX_LOAD_ENABLE, TX_SEND_ENABLE, RX_START_ENABLE and RX_STOP_ENABLE bits in the RFTIMER_REG__COMPAREi_CONTROL register. These triggers activate the state machines in the RFcontroller module to load the TX FIFO with the packet data, send the data in the TX FIFO, turn on the radio to listen for packets, and turn off the radio after listening for packets, respectively. The four triggers are intended to be mutually exclusive, though this is not enforced in the compare unit logic. For more information on these triggers and how they interact with the RFcontroller module, see section 3.25.

Changes can be made to both the RFTIMER_REG__COMPAREi and RFTIMER_REG__COMPAREi_CONTROL registers when the counter is enabled. If multiple changes are made during a single timer clock cycle, only the last change takes effect on the next rising edge of the timer clock.

Capture Unit i Registers

The RFTIMER_REG__CAPTUREi register, with width equal to the COUNTER_WIDTH parameter, stores the value of the counter register when of the enabled inputs to the capture unit is asserted. The write is enabled on the next rising edge of the system clock (HCLK) after the input is asserted, and therefore the input must be asserted for at least 1 HCLK cycle.

The RFTIMER_REG__CAPTUREi_CONTROL register contains nine bits to configure the inputs and interrupts to the capture unit. If the INTERRUPT_ENABLE bit is set, a capture event triggers an interrupt to the Cortex-M0, by setting the corresponding bit in the RFTIMER_REG__INT register on the next rising edge of the system clock (HCLK). If the INTERRUPT_ENABLE bit is set and the corresponding bit in RFTIMER_REG__INT register is not cleared before the next trigger/capture event, the corresponding overflow bit is also set in the RFTIMER_REG__INT register. If there is an overflow then the value in the RFTIMER_REG__CAPTUREi register is overwritten. If the INTERRUPT_ENABLE is not set, then the value in the RFTIMER_REG__CAPTUREi register is overwritten, and there is no indication to the Cortex-M0 of a capture event or an overflow.

There are six possible inputs for triggering each capture unit. These inputs can be enabled by setting the INPUT_SEL_SOFTWARE, INPUT_SEL_TX_LOAD_DONE, INPUT_SEL_TX_SFD_DONE, INPUT_SEL_TX_SEND_DONE, INPUT_SEL_RX_SFD_DONE, and INPUT_SEL_RX_DONE bits in the RFTIMER_REG__CAPTUREi_CONTROL register. The first input is from the Cortex-M0, and is triggered when the software sets the CAPTURE_NOW bit of the RFTIMER_REG__CAPTUREi_CONTROL register. The other five inputs are from the RFcontroller module. These inputs are single-cycle (using the system clock HCLK) pulses that indicate when loading the TX FIFO from data memory is done, transmitting the SFD of a packet is done, transmitting an entire packet is done, the SFD of a packet has been received, and an entire packet has been received and stored in data memory, respectively. For more information on these inputs from the RFcontroller module and their significance, see section 3.25.

Setting the CLEAR bit of the RFTIMER_REG__CAPTUREi_CONTROL register resets the value in the RFTIMER_REG__CAPTUREi register back to 0 on the next rising edge of the system clock (HCLK). If the software sets the CLEAR bit during the same system clock (HCLK) cycle that one of the inputs is triggered, then the trigger overrides the clear, and the value of the counter is copied into the RFTIMER_REG__CAPTUREi register.

Changes can be made to the RFTIMER_REG__CAPTUREi_CONTROL register when the counter is enabled. If multiple changes are made during a single timer clock cycle, only the last change takes effect on the next rising edge of the timer clock.

Interrupt and Interrupt Clear Registers

The RFTIMER module has one interrupt to the Cortex-M0. This interrupt is the bitwise OR of all of the bits in the RFTIMER_REG__INT register, where each bit corresponds to one interrupt source in the RFTIMER module. This interrupt is also globally enabled or disabled using the INTERRUPT_ENABLE bit of the RFTIMER_REG__CONTROL register.

The RFTIMER_REG__INT register contains 16 bits, eight of them corresponding to the interrupts of the compare units, four of them corresponding to the interrupts

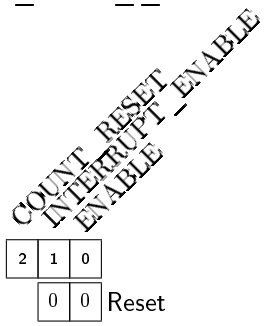
of the capture units, and four of them corresponding to the overflow bits of the capture units. Each bit in RFTIMER_REG__INT is set automatically by the RFTIMER module in the event of an interrupt from its corresponding capture/compare unit. Each bit in the RFTIMER_REG__INT is cleared by setting the corresponding bit in the RFTIMER_REG__INT_CLEAR register.

If the interrupt service routine does not disable the counter after an interrupt, then the timer may generate other interrupts as the service routine is running. If there are bits in RFTIMER_REG__INT that are not cleared before the interrupt service routine returns, then the interrupt signal will remain high, and the Cortex-M0 will execute the interrupt service routine again until all bits in RFTIMER_REG__INT are cleared.

It is recommended that the interrupt service routine store the value of RFTIMER-_REG__INT, perform any necessary actions, and then write that stored value to RFTIMER_REG__INT. This ensures that any additional interrupts as the service routine is running are not accidentally cleared, and the interrupt service routine is called again to handle with the new interrupts.

Register Descriptions

Register 3.17: RFTIMER_REG__CONTROL (0x42000000)

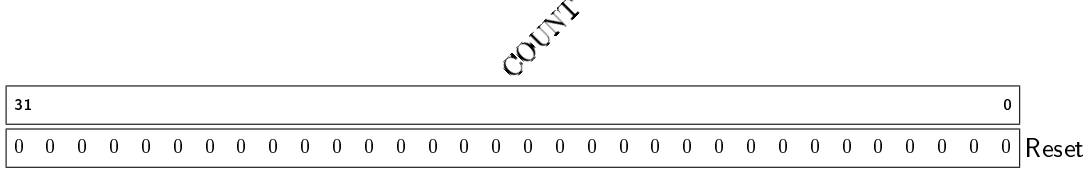


ENABLE Counter enable. The value in the RFTIMER_REG__COUNTER increments when enabled. 0 = disabled and 1 = enabled.

INTERRUPT_ENABLE Global interrupt enable. 0 = all interrupts disabled and 1 = interrupts enabled.

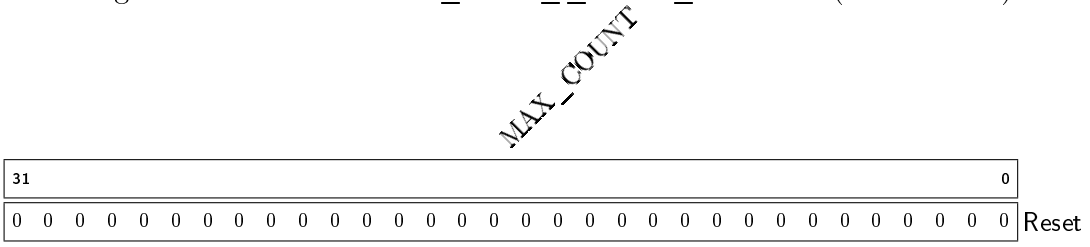
COUNT_RESET (Write-only) Clears the RFTIMER_REG__COUNTER register. 0 = no clear and 1 = clear.

Register 3.18: RFTIMER_REG__COUNTER (0x42000004)



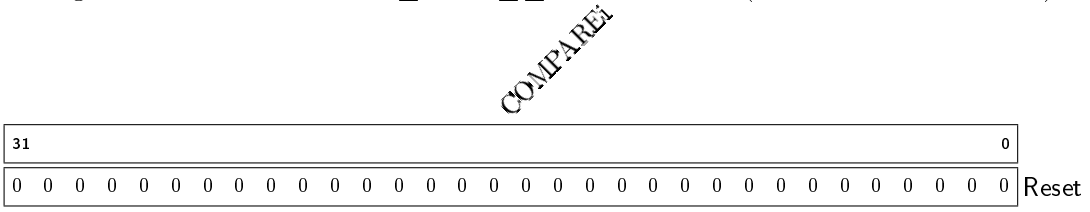
COUNT The counter for the timer.

Register 3.19: RFTIMER_REG__MAX_COUNT (0x42000008)



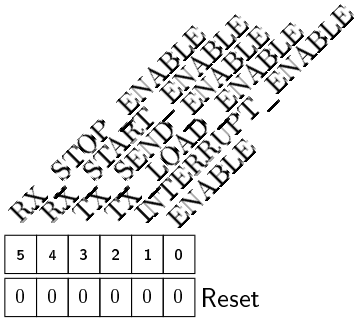
MAX_COUNT Holds the maximum value for the counter register, `RFTIMER_REG__COUNTER`.

Register 3.20: RFTIMER_REG__COMPAREi (0x42000010 + i*0x04)



COMPAREi Holds the data for comparison to the counter register, `RFTIMER_REG__COUNTER`.

Register 3.21: RFTIMER_REG__COMPAREi_CONTROL (0x42000030 + i*0x04)



ENABLE Compare unit enable. The value in the `RFTIMER_REG__COMPAREi` is compared to the value in `RFTIMER_REG__COUNTER` when enabled. 0 = compare unit disabled and 1 = compare unit enabled.

INTERRUPT_ENABLE Interrupt enable. 0 = interrupt disabled and 1 = interrupt enabled.

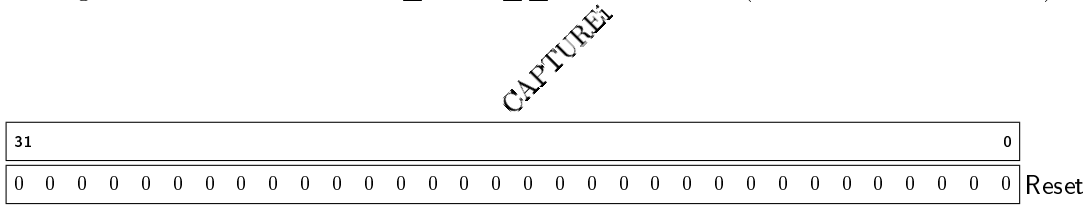
TX_LOAD_ENABLE TX_LOAD output trigger enable. 0 = trigger output disabled and 1 = trigger output enabled.

TX_SEND_ENABLE TX_SEND output trigger enable. 0 = trigger output disabled and 1 = trigger output enabled.

RX_START_ENABLE RX_START output trigger enable. 0 = trigger output disabled and 1 = trigger output enabled.

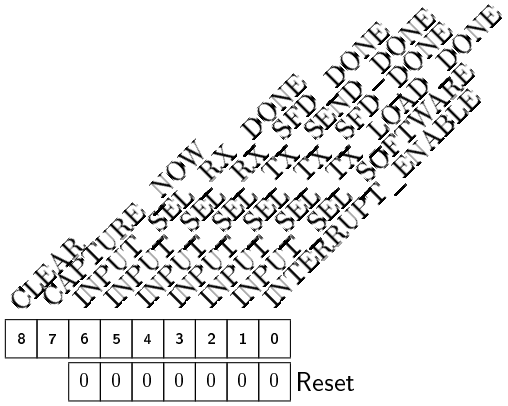
RX_STOP_ENABLE RX_STOP output trigger enable. 0 = trigger output disabled and 1 = trigger output enabled.

Register 3.22: RFTIMER_REG__CAPTUREi (0x42000050 + i*0x04)



CAPTUREi The counter register, `RFTIMER_REG__COUNTER`, is copied onto this register/field when a capture is triggered.

Register 3.23: RFTIMER_REG__CAPTUREi_CONTROL (0x42000060 + i*0x04)



INTERRUPT_ENABLE Interrupt enable. 0 = interrupt disabled and 1 = interrupt enabled.

INPUT_SEL_SOFTWARE Software capture input select. 0 = input disabled and 1 = input enabled.

INPUT_SEL_TX_LOAD_DONE TX_LOAD_DONE capture input select. 0 = input disabled and 1 = input enabled.

INPUT_SEL_TX_SFD_DONE TX_SFD_DONE capture input select. 0 = input disabled and 1 = input enabled.

INPUT_SEL_TX_SEND_DONE TX_SEND_DONE capture input select. 0 = input disabled and 1 = input enabled.

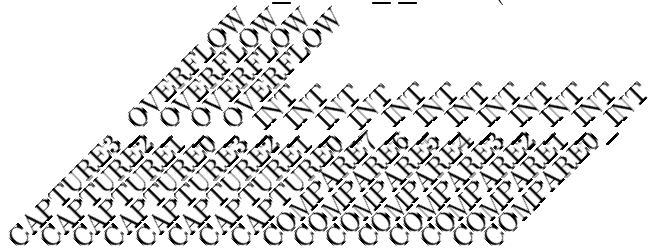
INPUT_SEL_RX_SFD_DONE RX_SFD_DONE capture input select. 0 = input disabled and 1 = input enabled.

INPUT_SEL_RX_DONE RX_DONE capture input select. 0 = input disabled and 1 = input enabled.

CAPTURE_NOW (Write-only) Triggers a capture event immediately. 0 = no capture and 1 = immediate capture.

CLEAR (Write-only) Clears the `RFTIMER_REG__CAPTUREi` register. 0 = no clear and 1 = clear.

Register 3.24: RFTIMER_REG__INT (0x42000070)



COMPARE0_INT Compare unit 0 interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

COMPARE1_INT Compare unit 1 interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

COMPARE2_INT Compare unit 2 interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

COMPARE3_INT Compare unit 3 interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

COMPARE4_INT Compare unit 4 interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

COMPARE5_INT Compare unit 5 interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

COMPARE6_INT Compare unit 6 interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

COMPARE7_INT Compare unit 7 interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

CAPTURE0_INT Capture unit 0 interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

CAPTURE1_INT Capture unit 1 interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

CAPTURE2_INT Capture unit 2 interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

CAPTURE3_INT Capture unit 3 interrupt flag. 0 = no interrupt pending and 1 = interrupt pending.

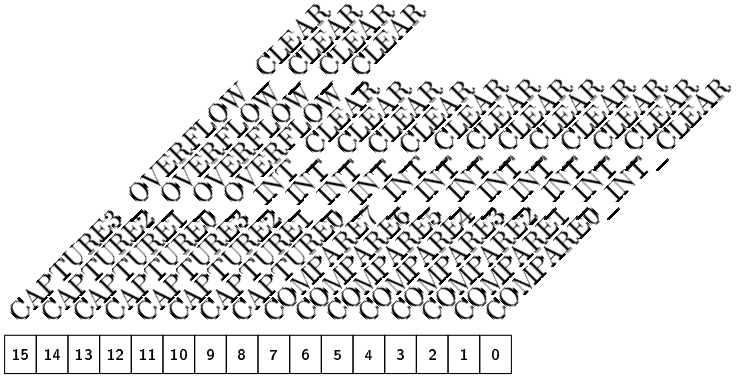
CAPTURE0_OVERFLOW Capture unit 0 overflow flag. 0 = no capture overflow occurred and 1 = capture overflow occurred.

CAPTURE1_OVERFLOW Capture unit 1 overflow flag. 0 = no capture overflow occurred and 1 = capture overflow occurred.

CAPTURE2_OVERFLOW Capture unit 2 overflow flag. 0 = no capture overflow occurred and 1 = capture overflow occurred.

CAPTURE3_OVERFLOW Capture unit 3 overflow flag. 0 = no capture overflow occurred and 1 = capture overflow occurred.

Register 3.25: RFTIMER_REG__INT_CLEAR (0x42000074)



COMPARE0_INT_CLEAR (Write-only) Compare unit 0 interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

COMPARE1_INT_CLEAR (Write-only) Compare unit 1 interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

COMPARE2_INT_CLEAR (Write-only) Compare unit 2 interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

COMPARE3_INT_CLEAR (Write-only) Compare unit 3 interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

COMPARE4_INT_CLEAR (Write-only) Compare unit 4 interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

COMPARE5_INT_CLEAR (Write-only) Compare unit 5 interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

COMPARE6_INT_CLEAR (Write-only) Compare unit 6 interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

COMPARE7_INT_CLEAR (Write-only) Compare unit 7 interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

CAPTURE0_INT_CLEAR (Write-only) Capture unit 0 interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

CAPTURE1_INT_CLEAR (Write-only) Capture unit 1 interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

CAPTURE2_INT_CLEAR (Write-only) Capture unit 2 interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

CAPTURE3_INT_CLEAR (Write-only) Capture unit 3 interrupt flag clear. 0 = flag unchanged and 1 = flag cleared.

CAPTURE0_OVERFLOW_CLEAR (Write-only) Capture unit 0 overflow flag clear. 0 = flag unchanged and 1 = flag cleared.

CAPTURE1_OVERFLOW_CLEAR (Write-only) Capture unit 1 overflow flag clear. 0 = flag unchanged and 1 = flag cleared.

CAPTURE2_OVERFLOW_CLEAR (Write-only) Capture unit 2 overflow flag clear. 0 = flag unchanged and 1 = flag cleared.

CAPTURE3_OVERFLOW_CLEAR (Write-only) Capture unit 3 overflow flag clear. 0 = flag unchanged and 1 = flag cleared.

3.36 compare_unit

3.36.1 Description

This module is part of the RFTIMER module, and implements the functionality for one of its compare units. This module stores a compare value and generates interrupts when the value of the counter from the RFTIMER module matches the compare value. One of the interrupts is for the Cortex-M0, and the other four are triggers for the state machines inside the RFcontroller module. The interrupt and trigger outputs from this module are synchronous with the timer clock (TCLK). For more details on the purpose and function of this module within the RFTIMER design, see section 3.35.

3.36.2 Input/Output Ports and Parameters

HRESETn Input reset.

HCLK System clock input.

TCLK Timer clock input.

HWDATA[COUNTER_WIDTH-1:0] Write data input. Used for AHB writes to the compare and compare control registers.

count[COUNTER_WIDTH-1:0] Counter input from the RFTIMER module.

compare_wr_en Write enable input for the compare register.

compare_control_wr_en Write enable input for the compare control register.

tx_load_trigger Output to the RFcontroller module for the TX_LOAD trigger.

tx_send_trigger Output to the RFcontroller module for the TX_SEND trigger.

rx_start_trigger Output to the RFcontroller module for the RX_START trigger.

rx_stop_trigger Output to the RFcontroller module for the RX_STOP trigger.

compare[COUNTER_WIDTH-1:0] Compare register output. Used for AHB reads from the compare register.

compare_control[5:0] Compare control register output. Used for AHB reads from the compare control register.

compare_match Compare match output. Used for the interrupt to the Cortex-M0.

interrupt_en Interrupt enable output from the compare control register. Used for the interrupt to the Cortex-M0.

COUNTER_WIDTH Parameter describing the width of the counter used for the timer.

3.36.3 Design Details

As with the RFTIMER module, all register names with the _t suffix indicate that the register is connected to TCLK. All register names with the _h suffix indicate that the register is connected to HCLK. A common pattern repeated throughout this design is the use of a _h register written via the AHB and an accompanying _t register sampling its value on every rising edge of TCLK.

This module has a pair of _t and _h registers for each bit in the compare control register: ENABLE, INTERRUPT_ENABLE, TX_LOAD_ENABLE, TX_SEND_ENABLE, RX_START_ENABLE, and RX_STOP_ENABLE. The _t registers for the compare control register are concatenated together to make the compare_control output. The compare_t/compare_h pair stores the compare register value, and compare_t is connected to the compare output.

The compare_match output is set to count == compare_t as long as the compare_enable_t register is high. Otherwise, the output stays low. Since both count and compare_t are synchronous with TCLK, the compare_match output is also synchronous with TCLK.

Each trigger output is the bitwise AND of the trigger's enable register and compare_match. The enable registers and compare_match are synchronous with TCLK, and therefore the trigger outputs are also synchronous with TCLK.

A schematic of the compare register, compare control registers, and trigger outputs is shown in Figure 3.26.

3.37 capture_unit

3.37.1 Description

This module is part of the RFTIMER module, and implements the functionality for one of its capture units. This module monitors several interrupts, and stores the value of the counter from the RFTIMER module when one of those interrupts is asserted. One of these interrupts comes from the Cortex-M0 via the AHB, and the other five come from the RFcontroller module. For more details on the purpose and function of this module within the RFTIMER design, see section 3.35.

3.37.2 Input/Output Ports and Parameters

HRESETn Input reset.

HCLK System clock input.

TCLK Timer clock input.

HWDATA[8:0] Write data input. Used for AHB writes to the capture control register.

count[COUNTER_WIDTH-1:0] Counter input from the RFTIMER module.

capture_control_wr_en Write enable input for the capture control register.

tx_load_done_in Input from the RFcontroller module for the TX_LOAD_DONE interrupt.

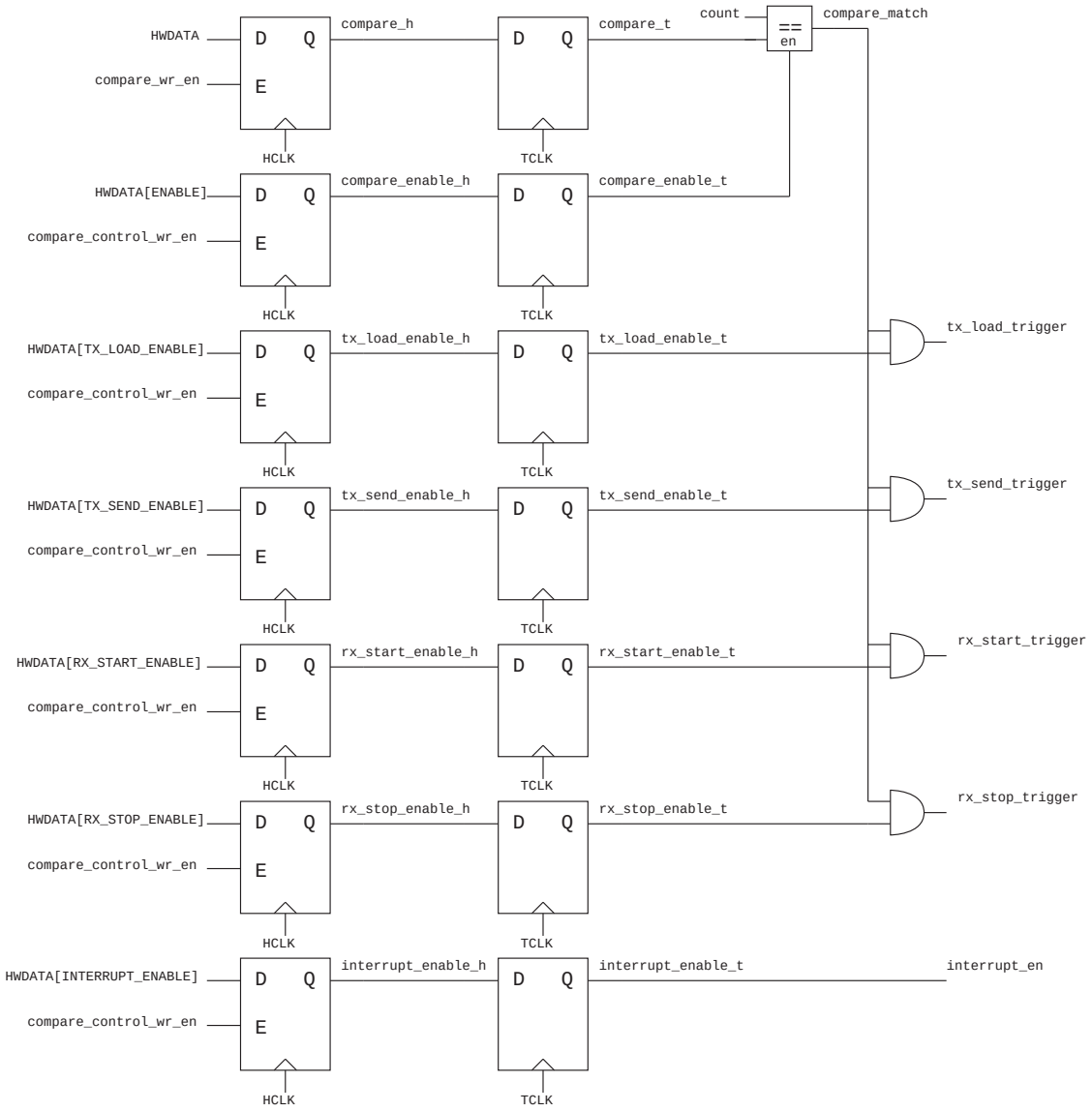


Figure 3.26: Schematic for the `compare_unit` module.

tx_sfd_done_in Input from the RFcontroller module for the TX_SFD_DONE interrupt.

tx_send_done_in Input from the RFcontroller module for the TX_SEND_DONE interrupt.

rx_sfd_done_in Input from the RFcontroller module for the RX_SFD_DONE interrupt.

rx_done_in Input from the RFcontroller module for the RX_DONE interrupt.

capture_trigger Capture trigger output. Used for the interrupt to the Cortex-M0.

capture[COUNTER_WIDTH-1:0] Capture register output. Used for AHB reads from the capture register.

capture_control[6:0] Capture control register output. Used for AHB reads from the capture control register.

interrupt_en Interrupt enable output from the capture control register. Used for the interrupt to the Cortex-M0.

COUNTER_WIDTH Parameter describing the width of the counter used for the timer.

3.37.3 Design Details

As with the RFTIMER module, all register names with the _t suffix indicate that the register is connected to TCLK. All register names with the _h suffix indicate that the register is connected to HCLK. A common pattern repeated throughout this design is the use of a _h register written via the AHB and an accompanying _t register sampling its value on every rising edge of TCLK.

This module has a pair of _t and _h registers for each read-write bit in the capture control register: INTERRUPT_ENABLE, INPUT_SEL_SOFTWARE, INPUT_SEL_TX-_LOAD_DONE, INPUT_SEL_TX_SFD_DONE, INPUT_SEL_TX_SEND_DONE, INPUT_SEL_RX-_SFD_DONE, and INPUT_SEL_RX_DONE. The _t registers for the capture control register are concatenated together to make the capture_control output. The CAPTURE-_NOW and CLEAR bits are write enable signals for the capture register, and are not part of the capture_control output, as they are write-only bits in the register.

The capture_trigger signal is the bitwise OR of all the possible capture interrupts after their bitwise AND with the corresponding input select signals. The capture_h register is updated with the count register value when capture_trigger is high on the rising edge of HCLK. The capture_h register is also cleared when the CLEAR bit of the control register is written and capture_clear_en is high. If both capture_trigger and capture_clear_en are asserted, the trigger takes precedence and the count value is copied into capture_h.

The capture register, capture_h, uses a synchronous enable to store the counter value. The synchronous enable imposes the requirement that the interrupt input is high for at least 1 clock cycle. The capture register is attached to the system clock (HCLK) and all input pulses must be asserted for at least 1 system clock cycle. An asynchronous alternative is to attach the interrupt inputs directly to the clock input of the capture_h register. However, this method carries the risk of capturing

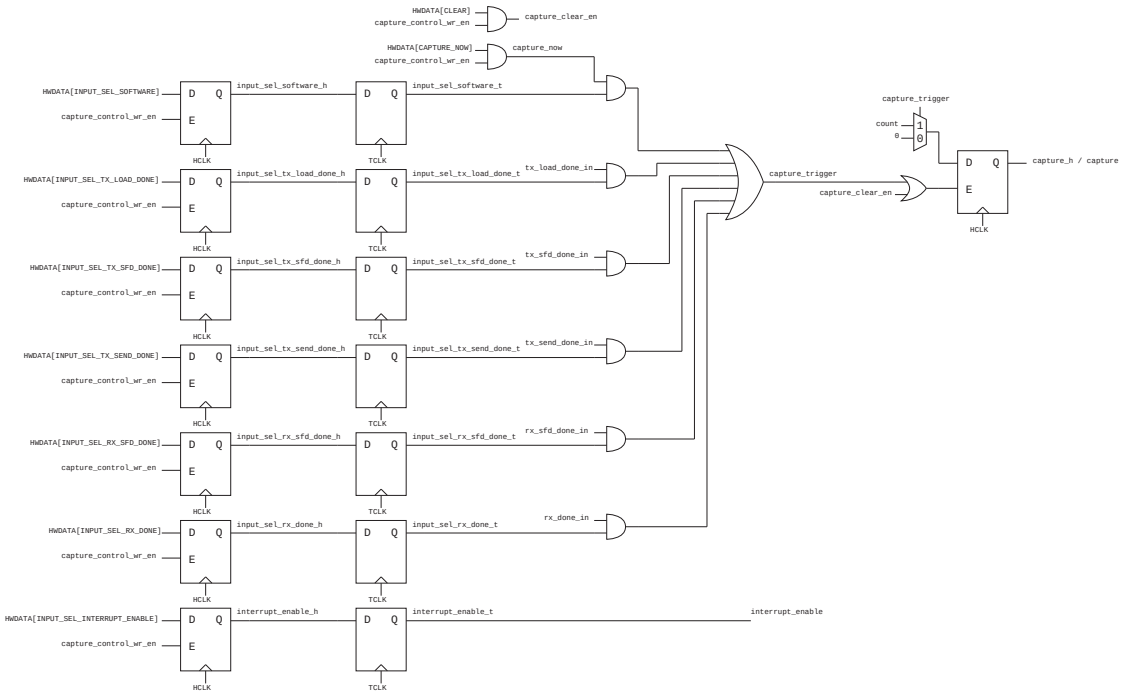


Figure 3.27: Schematic for the `capture_unit` module.

the counter as the value is changing, leading to metastability. This method is also not recommended on FPGAs as it requires connecting several non-clock signals to dedicated clock nets. Given the intended use of this module, to capture only when directed by the Cortex-M0 or through the `RFcontroller` module, both synchronous with the system clock, the former method is acceptable.

A schematic of the capture register and capture control registers is shown in Figure 3.27.

3.38 AHB2APB

3.38.1 Description

This module is designed by Bigazzi to act as a bridge connecting the AHB and the APB buses. This module is the master of the APB bus, and creates all of the APB master signals sent to the `APBMUX` and all APB slaves.

3.38.2 Input/Output Ports

`HCLK` Input clock. Used on both AHB and APB interfaces.

`HRESETn` Input reset. Used on both AHB and APB interfaces.

`HADDR[31:16]` AHB input address. Since the APB is a 16-bit bus, only the 16 upper bits of `HADDR` are needed. The rest are omitted.

`HTRANS[1]` AHB transfer type input.

HWRITE AHB write select input.

HWDATA[15:0] AHB write data input. Since the APB is a 16-bit bus, only the 16 lower bits of HWDATA used. The rest are omitted.

HSEL AHB slave select input.

HREADY AHB transfer finished input. This input indicates that the previous transfer on the bus has finished and that address phase signals must be latched.

HRDATA[31:0] AHB read data output.

HREADYOUT AHB transfer finished output.

PADDR[15:0] APB address output.

PENABLE APB access phase enable output.

PWRITE APB write select output.

PWDATA[15:0] APB write data output.

PRDATA[15:0] APB read data input.

PREADY APB transfer finished input.

3.38.3 Design Details

This module operates in three states, ST_IDLE, ST_SETUP, ST_ACCESS. During the ST_IDLE state, the bridge waits for an AHB transfer addressing one of the APB peripherals. Once a transfer is detected, the bridge latches the AHB address phase signals, and changes to the ST_SETUP state. This state corresponds to the setup phase of the APB transfer. Meanwhile, the AHB master is stalled by setting HREADYOUT to 0. After the ST_SETUP state is the ST_ACCESS state, corresponding to the access phase of the APB transfer. The bridge stays in this state until the slave indicates the state is complete using PREADY. All of the APB slave signals are also routed to the AHB master, as the APB access phase is the same as the AHB data phase. Once the transfer is complete, the bridge transitions to the ST_SETUP phase if there is another AHB transfer, or transitions to the ST_IDLE state if there are no more AHB transfers.

3.39 APBMUX

3.39.1 Description

This module is designed by Bigazzi to determine which APB slave is being accessed and route the correct set of slave signals to the APB master. It also decodes HADDR[31:24] to generate a PSEL signal for each slave. This module currently supports four APB slaves; however, with modifications this module may support any number of APB slaves.

3.39.2 Input/Output Ports

`PADDR[15:8]` Address input. Only the 8 upper bits are necessary and the other bits are omitted.

`PRDATA0[15:0]` Read data input from the first APB slave. In the Single Chip Mote digital system this is the `APBADC_V2` module.

`PREADY0` Transfer finished input from the first APB slave. In the Single Chip Mote digital system this is the `APBADC_V2` module.

`PRDATA1[15:0]` Read data input from the second APB slave. In the Single Chip Mote digital system this is the `APBUART` module.

`PREADY1` Transfer finished input from the second APB slave. In the Single Chip Mote digital system this is the `APBUART` module.

`PRDATA2[15:0]` Read data input from the third APB slave. In the Single Chip Mote digital system this is the `APB_ANALOG_CFG` module.

`PREADY2` Transfer finished input from the third APB slave. In the Single Chip Mote digital system this is the `APB_ANALOG_CFG` module.

`PRDATA3[15:0]` Read data input from the fourth APB slave. In the Single Chip Mote digital system this is the `APBGPIO` module.

`PREADY3` Transfer finished input from the fourth APB slave. In the Single Chip Mote digital system this is the `APBGPIO` module.

`PRDATA[15:0]` Read data output to the APB master.

`PREADY` Transfer finished output to the APB master.

`PSEL0` Slave select output for the first APB slave. In the Single Chip Mote digital system this is the `APBADC_V2` module.

`PSEL1` Slave select output for the second APB slave. In the Single Chip Mote digital system this is the `APBUART` module.

`PSEL2` Slave select output for the third APB slave. In the Single Chip Mote digital system this is the `APB_ANALOG_CFG` module.

`PSEL3` Slave select output for the fourth APB slave. In the Single Chip Mote digital system this is the `APBGPIO` module.

3.39.3 Design Details

This module only contains combinational logic to set the `PSEL` outputs and choose the correct slave inputs for `PRDATA` and `PREADY` using the 8 upper bits of `HADDR`. This is done using a case statement with the prefixes defined in `REGISTERS.vh`.

3.39.4 Adding Another APB Slave

Adding a new APB slave in this module requires the following steps:

1. Define an address prefix for the slave in `REGISTERS.vh`.
2. Add another `PSEL` output.
3. Add another `PRDATA` input.
4. Add another `PREADY` input
5. Add another case to the case statement using the new address prefix. Assign the new `PSEL` output. Assign `PRDATA` and `PREADY`.
6. Connect the new `PSEL` output to the new slave/peripheral in the top module, `uCONTROLLER`.
7. Connect the new `PRDATA` and `PREADY` inputs to the new slave/peripheral in the top module, `uCONTROLLER`.

3.40 APBUART

3.40.1 Description

This module is a slightly modified version of the `AHBUART` module provided in the DesignStart kit. Bigazzi adapted the original code to be used on the APB bus instead of the AHB, and parameterized the section of the code used to send data at the appropriate baud rate.

The `APBUART` module implements a 3-wire serial interface. The data is transmitted in individual data frames containing one start bit, 8 data bits, 1 stop bit, and no extra parity bits. This module also does not implement any flow control. The baud rate is determined based on the `UARTBAUDGEN` parameter in `SYS_PROP.vh`, and is currently set to 19200. The Nexys 4 DDR board has support for RTS/CTS flow control. However, this module does not have any flow control logic since the original `AHBUART` module was created for the Nexys 3 board (which does not support flow control).

To send a byte, write to the `UART_REG__TX_DATA` register. To read received data, read from the `UART_REG__RX_DATA` register. When the module receives one or more bytes of data, the interrupt to the Cortex-M0 is asserted. This interrupt remains active until all received data is read.

3.40.2 Input/Output Ports and Parameters

`HCLK` Input clock.

`HRESETn` Input reset.

`PADDR[15:0]` Address input.

`PWDATA[7:0]` Write data input. Only the 8 lower bits are used because this UART interface has 8 data bits.

PENABLE Access phase enable input.

PSEL Slave select input.

PWRITE Write select input.

PRDATA[15:0] Read data output. The upper 8 bits are always 0 since this UART interface has 8 data bits.

PREADYOUT Transfer finished output.

RsRx The receive input for UART.

RsTx The transmit output for UART.

uart_irq Interrupt output to the Cortex-M0.

CLK_FREQ Parameter describing the frequency of HCLK in Hz. This is used with the UARTBAUDGEN parameter to send data at the correct baud rate and sample the input data at the correct rate.

UARTBAUDGEN Parameter used to describe the target baud rate. This is used with the CLK_FREQ parameter to send data at the correct baud rate and sample the input data at the correct rate.

3.40.3 Design Details

The main section of this module modified by Bigazzi is the APB interface, previously the AHB interface of the AHBUART module in the DesignStart kit. This module instantiates four submodules, also provided as part of the DesignStart kit:

BAUDGEN This submodule is used to generate a one-cycle tick at a specified baud rate. This is accomplished with a counter used to generate a one-cycle tick when the counter reaches its maximum value. The maximum value is parameterized such that the frequency of ticks matches the baud rate. This module requires the UARTBAUDGEN and CLK_FREQ parameters in order to determine the maximum value of the counter.

FIFO This submodule (instantiated twice in APBUART) is for the FIFOs used to store outgoing Tx data and incoming Rx data.

UART_RX This module contains the logic and state machine used to sample the RsRx input, and write the data into the Rx FIFO. Unfortunately, this module is not the most ideal design, as it does not oversample the input.

UART_TX This module contains the logic and state machine used to toggle the RsTx output and send the data stored in the Tx FIFO.

All APB write transfers to this module, regardless of the address, store data into the Tx FIFO. If this FIFO has data, then the logic in UART_TX reads the data out of the FIFO and sends it through RsTx. Meanwhile, the UART_RX module samples the level on RsRx to listen for incoming data. This module then stores the data in the Rx FIFO. If the Rx FIFO has data, then the uart_irq interrupt to the Cortex-M0 is asserted. All APB read transfers from this module, regardless of address, read data from the Rx FIFO. Once this FIFO is empty, the interrupt is de-asserted.

3.40.4 Register Interface

UART TX Data

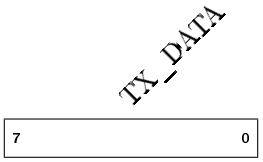
The `UART_REG__TX_DATA` register is an 8-bit write-only memory-mapped register used to write one symbol to be transmitted over UART to the TX FIFO. Any APB write to the `APBUART` module is a write to this register and to the TX FIFO. There is no feedback to indicate that this FIFO is full; any APB writes to the FIFO when it is full are ignored. It is up to the software on the Cortex-M0 to ensure that it does not write to the FIFO faster than the data is transmitted over UART.

UART RX Data

The `UART_REG__RX_DATA` register is an 8-bit read-only memory-mapped register that reads one symbol of received UART data from the RX FIFO. Any APB read from the `APBUART` module is a read from this register and from the RX FIFO. The `uart_irq` indicates that there is data in this FIFO. APB reads from the FIFO when it is empty return invalid data.

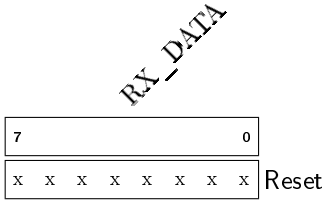
Register Descriptions

Register 3.26: UART_REG__TX_DATA (0x51000000)



TX_DATA (Write-only) Next UART symbol to write to the TX FIFO and transmit. Ignored if the FIFO is full.

Register 3.27: UART_REG__RX_DATA (0x51040000)



RX_DATA (Read-only) Next received UART symbol in the RX FIFO. Data is invalid if the FIFO is empty.

3.41 APBADC_V2

3.41.1 Description

This module is the digital interface to the analog-to-digital converter (ADC) designed by David Burnett for the Single Chip Mote. This module controls a series of inputs that activate and control the ADC; these inputs are toggled using an internal

state machine. The 10-bit result of the ADC is saved onto a register after the conversion is complete. This module also asserts an interrupt to the Cortex-M0 when a conversion is complete. The Verilog for this module is also designed by David Burnett.

3.41.2 Input/Output Ports

`HCLK` Input clock.

`HRESETn` Input reset.

`PSEL` Slave select input.

`PENABLE` Access phase enable input.

`PWRITE` Write select input.

`PWDATA[0:0]` Write data input. This is only 1 bit since the only writeable register has 1 bit.

`PADDR[15:0]` Address input.

`PRDATA[15:0]` Read data output.

`PREADYOUT` Transfer finished output.

`adc_int` Interrupt output to the Cortex-M0.

`adc_din[9:0]` Digital input from the ADC.

`adc_done` Input from the ADC indicating that the conversion is complete.

`adc_reset` Output control signal to the ADC.

`adc_clken` Output control signal to the ADC.

`adc_load` Output control signal to the ADC.

`adc_cdig` Output control signal to the ADC.

`adc_cvin` Output control signal to the ADC.

`adc_cvref` Output control signal to the ADC.

3.41.3 Design Details

Writing a 1 to the `ADC_REG__START` register sets the `conversion_started` register that triggers the internal state machine. Once the state machine is finished, the `done` signal is asserted for one cycle. The `done` signal is used as the interrupt output to the Cortex-M0, `adc_int`. For more details on the exact function of the state machine, see David Burnett. The higher-level use of this module, through the APB interface, is the same regardless of ADC architecture or state machine details, and is explained in the following section.

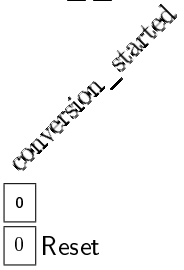
3.41.4 Register Interface

Writing a 1 to the 1-bit `ADC_REG__START` register triggers the beginning of the conversion. This register stays set until the conversion is complete, regardless of any other APB writes to the `ADC_REG__START` register. Reading this register indicates whether or not a conversion is in progress.

The converted data is read through the read-only 10-bit `ADC_REG__DATA` register. This register is updated at the end of every conversion.

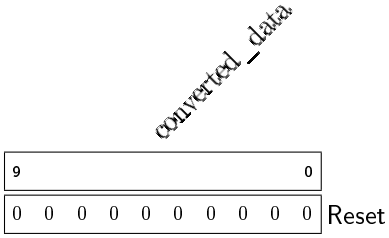
Register Descriptions

Register 3.28: ADC_REG__START (0x50000000)



conversion_started Reading this bit indicates if a conversion is in progress. Writing a 1 to this bit begins a conversion if one is not already in progress. Writing a 0 to this bit has no effect.

Register 3.29: ADC_REG__DATA (0x50040000)



converted_data Converted ADC value.

3.42 APB_ANALOG_CFG

3.42.1 Description

This module contains a parameterizable amount of 16-bit programmable registers used to configure any analog circuits on the Single Chip Mote. These registers are connected to outputs in the top module. On an FPGA these are connected to actual output pins, but on an ASIC they are not connected to any external pins. Instead they are connected directly to the analog circuit requiring configuration. The number of 16-bit registers is parameterized in order to accommodate an unknown number of required configuration outputs. All configuration registers are exactly 16 bits wide.

3.42.2 Input/Output Ports

`HCLK` Input clock.

`HRESETn` Input reset.

`PSEL` Slave select input.

`PENABLE` Access phase enable input.

`PWRITE` Write select input.

`PWDATA[15:0]` Write data input.

`PADDR[15:0]` Address input.

`PRDATA[15:0]` Read data output.

`PREADYOUT` Transfer finished output.

`ANALOG_CFG[(NUM_REGISTERS)-1:0]` Configuration register output. All configuration registers are concatenated into one bus, with the registers in order from the lowest address (in the lowest 16 bits) to the highest address (in the highest 16 bits), with the bits of each register in order of LSB to MSB.

`NUM_REGISTERS` Parameter describing the total number of 16-bit configuration registers.

3.42.3 Design Details

The module creates an array of 16-bits registers with depth equal to `NUM_REGISTERS`. The following generate statement is used to handle APB writes to each of the registers using the `NUM_REGISTERS` parameter:

```
genvar i;
generate
for (i = 0; i < NUM_REGISTERS; i = i + 1) begin : analog_cfg_reg_gen

always @ (posedge HCLK or negedge HRESETn) begin
    if (!HRESETn) analog_cfg_reg[i] <= 16'h0000;
    else if(PSEL & PWRITE & PENABLE)
        if (PADDR[15:0] == (`APB_BASE__ANALOG_CFG + (i*16'h04))) analog_cfg_reg[i]
            <= PWDATA;
end

end
endgenerate
```

The following for loop is used to create the multiplexer used to assign `PRDATA` during an APB read:

```
integer j;
always @(*) begin
    rPRDATA = 16'h0000;
    for (j = 0; j < NUM_REGISTERS; j = j + 1) begin : apb_read
        if (PADDR[15:0] == (`APB_BASE__ANALOG_CFG + (j*16'h04))) rPRDATA =
            analog_cfg_reg[j];
    end
end
```

The following for loop is used to concatenate all of the configuration registers into the single `ANALOG_CFG` output:

```
integer k;
always @(*) begin
    rANALOG_CFG = 0;
    for (k = 0; k < NUM_REGISTERS; k = k + 1) begin : r_analog_cfg
        rANALOG_CFG[k*16 +: 16] = analog_cfg_reg[k];
end
end
```

The `[k*16 +: 16]` syntax is called index part-select, where the first term, `k*16`, is the bit offset (the LSB), and the second term, `16`, is the width added to the offset to determine the MSB. This is equivalent to the following expression: `[((k+1)*16)-1:k*16]`; however this expression is illegal in Verilog.

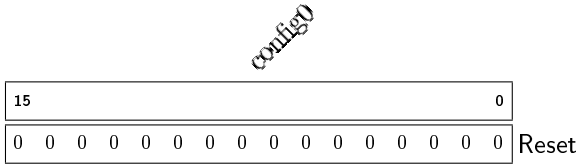
3.42.4 Register Interface

Analog Config i Register

The `ANALOG_CFG_REG__i` register is a 16-bit register corresponding to the ith analog configuration register, where i ranges from 0 to `NUM_REGISTERS-1`. The 16 bits in this register are connected to the `ANALOG_CFG[16*(i+1)-1:16*i]` output signals. Writing a 1 to a bit in this register drives the corresponding output high, and writing a 0 to a bit in this register drives the corresponding output low. Reading this register returns the current state of the `ANALOG_CFG[16*(i+1)-1:16*i]` outputs.

Register Descriptions

Register 3.30: ANALOG_CFG_REG__0 (0x52000000)



config0 Configuration register 0 output voltage. For each bit, 0 = low/ground and 1 = high/vdd.

3.43 APBGPIO

3.43.1 Description

This module is an interface for up-to 16 general-purpose digital inputs and up-to 16 general purpose digital outputs. The inputs may be used for buttons and switches on the FPGA board, or any kind of digital input on an ASIC. The outputs may be used to drive LEDs on the FPGA board, or any kind of digital output on an ASIC. The number of inputs and outputs are parameterized in order to accommodate an unknown number of pins available on the FPGA or ASIC. The current design includes inputs and 4 outputs.

3.43.2 Input/Output Ports and Parameters

`HCLK` Input clock.

`HRESETn` Input reset.

`PSEL` Slave select input.

`PADDR[15:0]` Address input.

`PENABLE` Access phase enable input.

`PWRITE` Write select input.

`PWDATA[15:0]` Write data input.

`PRDATA[15:0]` Read data output.

`PREADYOUT` Transfer finished output.

`gp_in[NUM_INPUTS-1:0]` General-purpose digital input.

`gp_out[NUM_OUTPUTS-1:0]` General-purpose digital output.

`NUM_INPUTS` Parameter defining the number of digital inputs.

`NUM_OUTPUTS` Parameter defining the number of digital outputs.

3.43.3 Design Details

The number of inputs and outputs are parameterizable because the number of available pins on the ASIC version of this design is currently unknown. The maximum number of inputs and outputs is 16 bits each to allow for all inputs to be read using one APB transfer and all outputs to be written using one APB transfer. It is highly unlikely that there will be a need for more inputs or outputs given the limited number of pins available on ASIC designs.

The `gp_in` inputs are sampled during the setup phase of an APB read transfer, indicated by the following boolean expression: `PSEL & ~PWRITE & ~PENABLE`. This way the sampled input is ready during the first cycle of the access phase. On the other hand, the new values are written to `gp_out` during the first cycle of the access phase. In both cases the APB transfers take only two cycles.

3.43.4 Register Interface

General Purpose Input Register

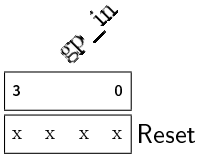
The `APBGPIO_REG__INPUT` register is a read-only register with a width of `NUM_INPUTS`. Each bit in this register corresponds to one of the general-purpose digital inputs. Each bit reads 0 when the input is low and 1 when the input is high.

General Purpose Output Register

The `APBGPIO_REG__OUTPUT` register is a read-write register with a width of `NUM_OUTPUTS`. Each bit in this register corresponds to one of the general-purpose outputs. Writing a 1 to a bit in this register drives the corresponding output high, and writing a 0 to a bit in this register drives the corresponding output low. Reading this register returns the current state of the outputs.

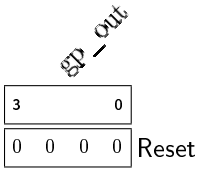
Register Descriptions

Register 3.31: APBGPIO_REG__INPUT (0x53000000)



gp_in (Read-Only) Input voltage of the general-purpose inputs 0-3. For each bit, 0 = low/-ground and 1 = high/vdd.

Register 3.32: APBGPIO_REG__OUTPUT (0x53040000)



gp_out Output voltage for the general-purpose outputs 0-3. For each bit, 0 = low/ground and 1 = high/vdd.

3.44 chipscope_debug

`chipscope_debug.cdc` is ChipScope definition and connection file. ChipScope is a tool from Xilinx used to sample signals from within a design as it is running on the FPGA. The module that samples these signals is referred to as an integrated logic analyzer (ILA). This tool is typically used for debugging purposes.

In order to make it easier to connect signals to the ILA, the Synthesis process needs to be modified to preserve module hierarchy. When hierarchy is not preserved, signals and nets are combined and renamed, making them much more difficult to find when selecting nets to connect to the ILA. To preserve hierarchy, go to the Synthesis process properties, and change the Keep Hierarchy option to Yes.

To connect signals to the ILA, first synthesize the design. Then, double-click the chipscope_debug.cdc file in the Design Hierarchy panel. If there is no ChipScope definition and connection file in the design, create one using the New Source option in the Project menu. Then open the CDC file.

Follow the prompts and directions to create a new ILA unit, if one does not exist. There are three tabs with settings to configure the ILA unit. These tabs are Trigger Parameters, Capture Parameters, and Net Connections. The Trigger

Parameters tab is used to configure how many signals to monitor for triggers. There can be multiple trigger units for multiple triggers. The Capture Parameters tab is used to determine how many data signals to sample (these can be the same as the triggers or different), and how many samples to take after a trigger. The Net Connections tab is used to connect nets in the design to the trigger and data ports of the ILA, as well as specify the sampling clock. Beware that the ILA module uses block RAM resources on the FPGA, and it is possible to exceed the amount of resources available.

After all of the ILA inputs have been connected, the design can be implemented and a bitstream file generated. The ChipScope Pro software can be used to load the bitstream onto the FPGA and access the ILA unit inside the design.

For a more complete tutorial on using ChipScope, see the following tutorial from Xilinx: Using Xilinx ChipScope Pro ILA Core with Project Navigator to Debug FPGA Applications [16]. A copy is found in `scm-digital/doc`.

3.45 Deprecated Modules

This section contains information on deprecated Verilog modules or modules generated in Coregen for Single Chip Mote on the Artix-7 and uRobotDigitalController on the Spartan-6. Some of these modules are found in the ISE project files for both the Artix-7 and Spartan-6 versions, some are only in the Spartan-6 project, and some of them have been removed from the project files. The Verilog code remains in the `scm-digital/src/hw/` folder.

The following sections describe the origin, function, and purpose of these modules, the reason why they are considered deprecated, and any possible uses for these modules in the future. This section does not go into detail on how these modules accomplish (or do not accomplish) their intended function.

3.45.1 clk_div22

This module was created by Bigazzi using a copy of the code generated by Coregen for a clock divider. This module was used in uRobotDigitalController to divide the 100MHz clock to 5MHz using FPGA clocking primitives. This module was deprecated because it was replaced by the `PON` module, which accomplishes the same function while also handling resets. This module is most likely not be needed for future designs.

3.45.2 pb_debounce

This module was provided by ARM as part of the ARM Cortex-M0 DesignStart kit (see section 2.2 for more information) in order to act as a push-button debouncer for the buttons on the Nexys 3 board. This module was removed from the Single Chip Mote and uRobotDigitalController projects because the FPGA button inputs (aside from the reset button which has its own debouncing module) have been removed from the design. This removal was done to make the design closer to the ASIC version of the Single Chip Mote, which does not have many external IOs and most likely will not use any buttons. This module can be used for any future designs on FPGAs if buttons are needed.

3.45.3 DMA

This is the original `DMA` module designed by Bigazzi, with further modifications added before its eventual deprecation. This module was deprecated because it was designed to interface with an older version of the RFcontroller and ADC and requires many changes to make it compatible with the new designs. This module is most likely not be needed in the future, and any updates to the DMA should be done to `DMA_V2` or a new DMA module should be created.

3.45.4 AHBTIMER

This module was designed by Bigazzi to be a timer for the Single Chip Mote. This module was deprecated because it was found unnecessary when the `RFTIMER` module was used instead, and keeping both modules was seen as excessive. However, the `RFTIMER` module was designed to be specifically used for the RFcontroller, and this timer may be better suited for other microcontroller applications. Therefore this module can be used in future designs if another, more generalized, timer is needed.

3.45.5 AHB2LED

This module was provided by ARM as part of the ARM Cortex-M0 DesignStart kit (see section 2.2 for more information) in order to act as an interface to the LEDs on the Nexys 3 board. This module was removed from the design because LEDs have been removed from the design. This removal was done to make the design closer to the ASIC version of the Single Chip Mote, which does not have many external IOs and most likely will not have any programmable LED outputs. One of the outputs from the `AHBGPIO` module can be used as a programmable LED output instead. This module can be used on future designs with FPGAs if LEDs are needed.

3.45.6 AHB2MEM_V2

This module was provided by ARM as part of the ARM Cortex-M0 DesignStart kit (see section 2.2 for more information) in order to act as an interface to the data memory on the FPGA. This module was slightly modified by Bigazzi to use a synchronous read in order to use the block RAM on the FPGA. This module was depreciated because it was replaced with the `AHBDMEM` module, containing almost the exact same code except for the use of an instantiated RAM instead of an inferred RAM. This was done to bring the design closer to the ASIC version, which also uses instantiated RAM rather than inferred RAM. This module is most likely not be needed for any future designs.

3.45.7 AHB2SRAMFLSH

This module was provided by ARM as part of the ARM Cortex-M0 DesignStart kit (see section 2.2 for more information) in order to act as an interface to the instruction memory on the FPGA. This instruction memory was stored on an external RAM rather than an internal RAM, to make it easier to load the software for the ARM Cortex-M0 DesignStart kit using Digilent Adept. This module was depricated because fetching instructions from the external RAM was much slower than using

the internal RAM, thus necessitating the creation of the AHBIMEM module along with the bootloader. Also, this external RAM is only found on the Nexys 3 board and not the Nexys 4 DDR. Therefore, this module is most likely not be needed for any future designs.

3.45.8 AHB2SRAMFLSH_V2

This module is a modified version of AHB2SRAMFLSH meant to speed up instruction fetches from the external RAM. This module was deprecated for the same reason as AHB2SRAMFLSH and is most likely not be needed for any future designs.

3.45.9 AHB2SRAMFLSH_V3

This module is a modified version of AHB2SRAMFLSH_V2. The external RAM interface is used to fetch all instructions from the external RAM and store them into internal RAM. This module was a preliminary bootloader that loaded the data from the external RAM instead of the 3 Wire Bus (see chapter 5 for more information on the actual bootloader). This module was deprecated for the same reason as AHB2SRAMFLSH and AHB2SRAMFLSH_V2 and is most likely not be needed for any future designs.

3.45.10 AHBROM

This module is a heavily modified version of AHB2MEM_V2 used to demonstrate that instruction data could be read from an instantiated ROM (rather than from internal or external RAM). This module was depricated because the AHBIMEM module was created to hold the instruction ROM, instruction RAM, and all bootloading logic. This module is most likely not be needed for any future designs.

3.45.11 AHB_MASTER_MUX

This module was designed by Bigazzi in order to act as a bus arbiter for the AHB, allowing for multiple masters to control one AHB bus. This module was deprecated because it was shown that the module did not function as intended and it was difficult to find the cause of this problem. The module was replaced with AHBLiteArbiter_V2, and is most likely not be needed for any future designs.

3.45.12 startSymbolDetect

This module was created as a precursor to the correlator for the RFcontroller. Prior to adding the spreading and despreading functions, the data bits were sent without any encoding, and this module was used to detect the packet start symbol in a stream of received bits. This module was deprecated because the correlator module was created to detect the packet start symbol in a stream of encoded bits. This module is most likely not be needed for any future designs.

3.45.13 APBADC

This module was designed by Bigazzi as a controller for a SAR ADC circuit he designed on a breadboard. This module was depreciated because this particular ADC circuit is no longer in use and a new ADC was designed for the Single Chip Mote. A new controller, `APBADC_V2` was written for this new ADC. This module is most likely not be needed for any future designs.

3.45.14 APBTSCHTimer

This module was designed by Bigazzi to be a timer for the Single Chip Mote. This module was deprecated because it was found unnecessary when the `RFTIMER` was used instead, and keeping both modules was seen as excessive. However, the `RFTIMER` was designed to be specifically used for the RFcontroller, and this timer may be better suited for other microcontroller applications. Therefore this module can be used in future designs if another, more generalized, timer is needed.

3.45.15 APB_PWM_simple

This module was designed by Bigazzi to be a programmable PWM output from the Single Chip Mote. This module was deprecated because a PWM output is currently not needed and was seen as excessive. However, this module may be useful in future iterations of the Single Chip Mote for microcontroller applications.

3.45.16 APBDO

This module was designed to be a simple digitally-controlled output from the Nexys 3 board. This module was removed from the design because the outputs from the `APBGPIO` module perform the same function, and the `APBGPIO` module is parameterizable. This module is most likely not be needed for any future designs.

3.45.17 APBLED

This module was designed to be an interface to the LEDs on the Nexys 3 board. This module was removed from the design because LEDs have been removed from the design. This removal was done to make the design closer to the ASIC version of the Single Chip Mote, which does not have many external IOs and most likely will not have any programmable LED outputs. One of the outputs from the `AHBGPIO` module can be used as a programmable LED output instead. This module can be used on future designs with FPGAs if LEDs are needed.

3.45.18 APBSW

This module was designed to read the input from the switches on the Nexys 3 board. This module was removed from the design because switch inputs have been removed from the design. This removal was done to make the design closer to the ASIC version of the Single Chip Mote, which does not have many external IOs and most likely does not have any switch inputs. One of the inputs to the `AHBGPIO`

module can be used as an input for a switch instead. This module can be used on future designs with FPGAs if switches are needed.

Chapter 4

Single Chip Mote Software

This chapter provides an overview of the Single Chip Mote digital system software and software development tools. While the majority of the work on the Single Chip Mote is focused on hardware design, the overall goal of this project is to work in collaboration with software developers to design a platform ideal for Internet of Things (IoT) applications. In particular, the Single Chip Mote team is collaborating with the developers of OpenWSN [26], an open-source protocol stack designed for IoT. With their advanced embedded systems experience, the developers of OpenWSN can provide feedback for hardware improvements while they port OpenWSN onto the Single Chip Mote. Through the use of FPGAs, hardware changes can quickly be verified, fine-tuned, and integrated into the software. The information in this chapter contains the basics for writing software that uses the digital system peripherals in order to facilitate application development.

4.1 Keil Project Settings

Keil projects have already been created for the main software running on the Cortex-M0 and the basic firmware (chapter 5) used to load the main software onto the Single Chip Mote digital system. However, it may be necessary in the future to create new projects for different applications or variations of the Single Chip Mote hardware. This section contains the information needed to create a new project in Keil uVision 5 for the ARM Cortex-M0 on the Single Chip Mote digital system.

4.1.1 New Project and Device Selection

To create a new project, open Keil uVision 5 and select the New uVision Project... option in the Project menu. This will bring up the Device Selection, displaying all of the ARM cores supported by the IDE. The device for the Single Chip Mote digital system is found under ARM > ARM Cortex M0 > ARMCM0, as shown in Figure 4.1. Select ARMCM0 and then select Next.

The Manage Runtime Environment window in Figure 4.2 displays options for including additional software drivers from ARM. The default settings are sufficient for the Single Chip Mote. Select OK to finish creating the project.

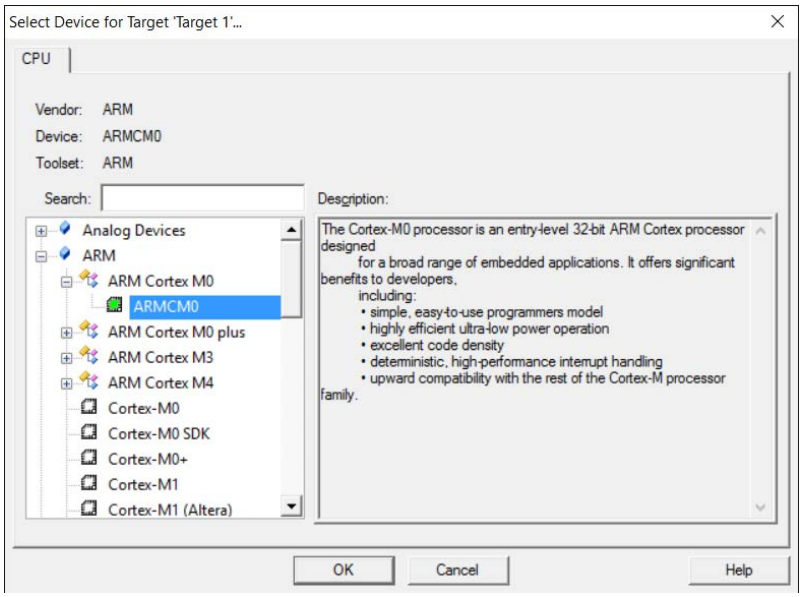


Figure 4.1: Device selection window in Keil

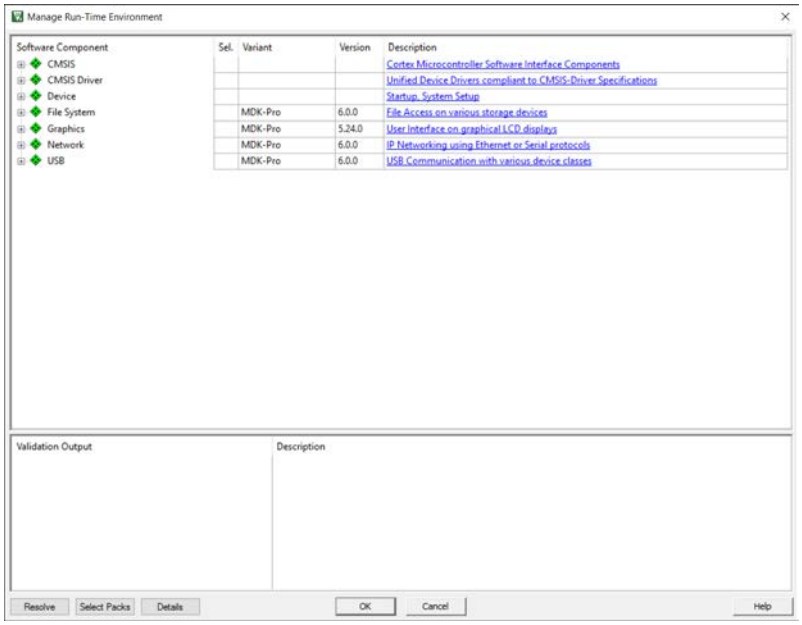


Figure 4.2: Manage Runtime Environment window in Keil

4.1.2 Target Options

The Target Options window contains the settings that describe the memory resources on the device, the compiler and linker options, and the debug options. To access the Target Options window, right-click Target 1 in the Project panel, and select Options for Target 'Target 1' (see Figure 4.3).

The Target Options window has 10 tabs:

Device This tab is used to select or change the CPU for the project. This is similar to the Device Selection window in Figure 4.1.

Target This tab, shown in Figure 4.4, specifies the CPU clock frequency (in the box labeled Xtal (MHz)), operating system, and memory regions. For the Single Chip Mote digital system, the frequency is 5MHz, and there is no operating system. There is one read-only memory area, corresponding to the instruction memory, and one read-write memory, corresponding to the data memory. The instruction memory is specified as on-chip memory with a start address of `0x0`. The size field indicates the size of the memory in bytes, where 64kB is represented in hex as `0x10000`. The startup option indicates that the code is loaded from this memory on startup. The data memory is specified as on-chip memory with a start address of `0x20000000`. The size is also 64kB (represented in hex as `0x10000`).

Output This tab, shown in Figure 4.5, specifies the name of the build output (in the box labeled Name of Executable), and the type of output (executable or library). The Single Chip Mote requires an executable, and the output is called `code.axf`. The check boxes labeled Debug Information, Create HEX File, Browse Information, and Create Batch File are not required for the Single Chip Mote.

Listing This tab contains compiler and linker options. The default settings for this tab are sufficient.

User This tab, shown in Figure 4.6, provides the option to run programs before compilation or before/after builds. For the Single Chip Mote, one program must be run after the build to convert `code.axf` to a binary file, `code.bin`. This is done by running the following command: `fromelf -bin code.axf -o code.bin`. Another program can optionally be run to convert `code.axf` to a disassembled text file. The disassembled file lists the assembly code in `code.axf` in a human-readable form, and may be useful for debugging purposes. This is done by running the following command: `fromelf -cvf code.axf -o disasm.txt`.

C/C++ This tab contains additional compiler options. The default settings for this tab are sufficient.

Asm This tab contains the assembler options. It is recommended that the Thumb Mode option is checked for the Single Chip Mote. Thumb Mode uses 16-bit instructions instead of 32-bit instructions when possible, with the overall effect of reduced code size.

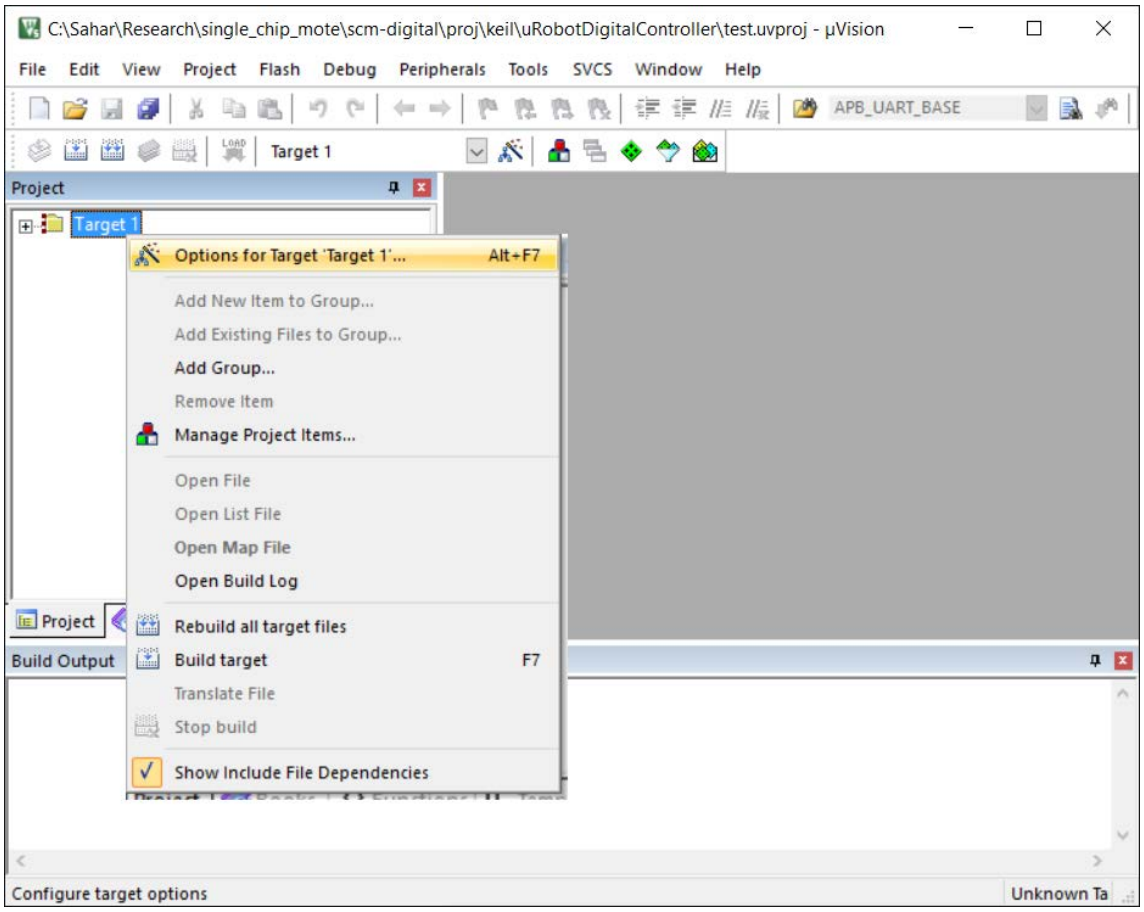


Figure 4.3: Opening the Target Options window

Linker This tab, shown in Figure 4.7, contains additional linker options. It is recommended that the Use Memory Layout from Target Dialog box is checked. This means that the memory regions defined in the Target tab are used to generate a scatter file (called `code.sct`) that describes the available memory regions for the linker. If this box is unchecked, then a scatter file must be provided in the Scatter File box.

Debug This tab contains the debug options. Since the ARM Cortex-M0 DesignStart processor does not have debug capabilities, the settings in this tab make no difference.

Utilities This tab contains the options for flashing the software onto a device using Keil. This is currently not possible with the Single Chip Mote, and therefore the settings in this tab make no difference.

Overall, the default settings in the Target Options window are sufficient for basic applications on the Single Chip Mote. Further fine-tuning and customization is recommended in the future for more optimized code generation.

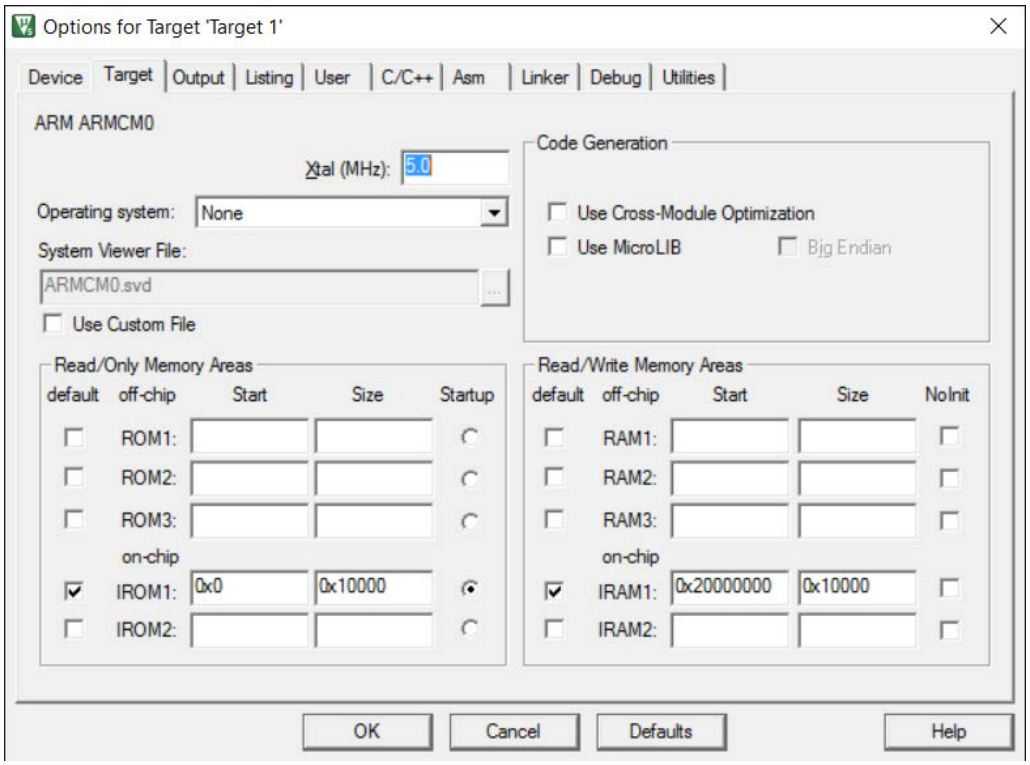


Figure 4.4: Target tab in the Target Options window

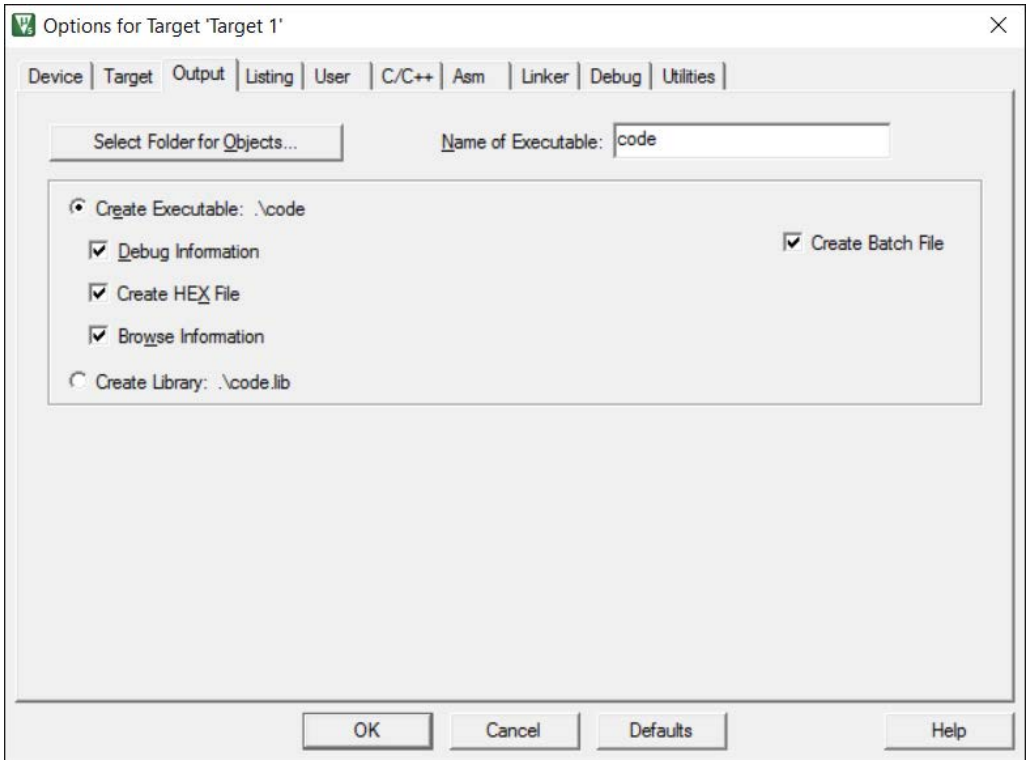


Figure 4.5: Output tab in the Target Options window

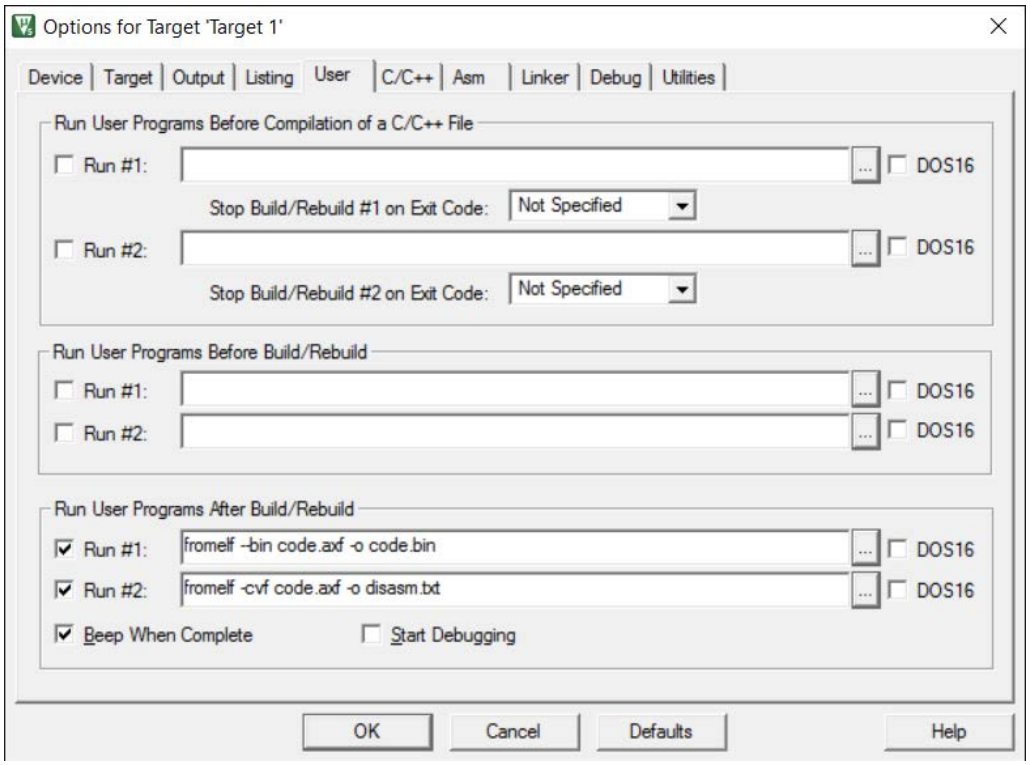


Figure 4.6: User tab in the Target Options window

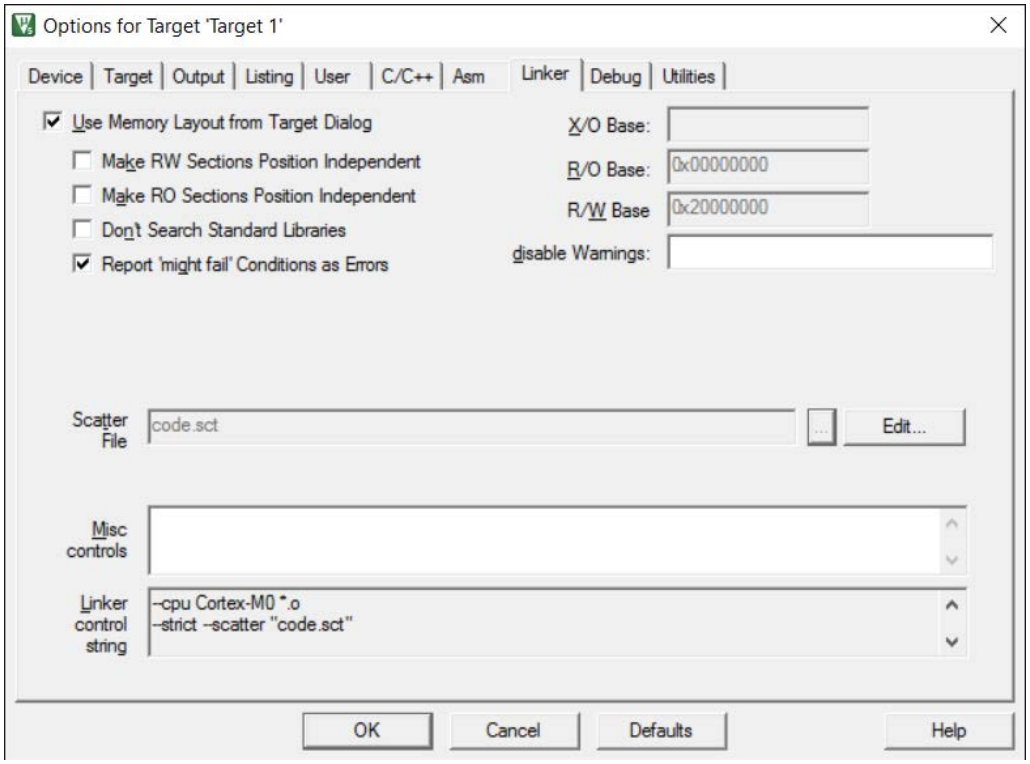


Figure 4.7: Linker tab in the Target Options window

4.1.3 Scatter File Settings

A scatter file describes the memory regions available to a microcontroller for the linker. Keil can generate a scatter file based on settings provided in the Target Options window. Below is an example of the scatter file automatically generated by Keil for the Single Chip Mote. For more information on scatter file see Chapter 4 of the ARM compiler toolchain Version 5.0 Linker Reference [3].

```
; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************

LR_IROM1 0x00000000 0x00010000  {    ; load region size_region
    ER_IROM1 0x00000000 0x00010000  {  ; load address = execution address
        *.o (RESET, +First)
        *(InRoot$$Sections)
        .ANY (+RO)
    }
    RW_IRAM1 0x20000000 0x00010000  {  ; RW data
        .ANY (+RW +ZI)0.9
    }
}
```

4.2 Required Assembly, Header, and C Files

The Cortex-M0 DesignStart processor on the Single Chip Mote digital system is programmed using a combination of C, C++, and ARM assembly code in Keil. Each project in Keil requires some basic assembly code for initial startup (found in `cm0dsasm.s`), a C header file with the addresses of the memory-mapped registers on the Single Chip Mote (found in `Memory_Map.h`), a file with C code to implement `printf` over UART (found in `retarget.c`), and finally any files with the `main` function and any other code required for the application.

4.2.1 cm0dsasm.s

This file is based on an example provided in the ARM Cortex-M0 DesignStart kit. This file contains ARM assembly code required by the Cortex-M0 in order to handle startup, resets, and interrupts properly.

The first section of this code (shown below) defines the size and properties of the stack and heap. The size of the stack and heap must add up to be less than or equal to the available data memory. Note that not all data is stored on the stack or heap; global variables are stored in a special section of memory. If the program has any global variables, the size of the stack and heap must be less than the data memory in order to leave room for those variables.

```
Stack_Size      EQU     0x0800                          ; 4KB of STACK

                AREA    STACK, NOINIT, READWRITE, ALIGN=4
Stack_Mem       SPACE   Stack_Size
__initial_sp


Heap_Size       EQU     0x0400                          ; 2KB of HEAP

                AREA    HEAP, NOINIT, READWRITE, ALIGN=4
__heap_base
Heap_Mem        SPACE   Heap_Size
__heap_limit
```

The second section of this code (shown below) defines the vector table. The vector table is a series of addresses defined at the beginning of a program that point to reset handlers, exception handlers, and interrupt handlers. Each line beginning with DCD initializes one word of memory with the value written after DCD:

```
; Vector Table Mapped to Address 0 at Reset

                    PRESERVE8
                    THUMB

                        AREA    RESET, DATA, READONLY
                        EXPORT  __Vectors

__Vectors               DCD     __initial_sp
                        DCD     Reset_Handler
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0

; External Interrupts

                        DCD     UART_Handler
                        DCD     0
                        DCD     0
                        DCD     ADC_Handler
                        DCD     0
                        DCD     0
                        DCD     RF_Handler
                        DCD     RFTIMER_Handler
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
```

The first 16 entries in the vector table (and their addresses in the vector table) are: initial stack pointer value (0x00), Reset handler address (0x04), NonMaskable Interrupt handler address (0x08), HardFault handler address (0x0C), Reserved (0x10-0x28), SVCall (0x2C), Reserved (0x30-0x34), PendSV (0x38), and SysTick (0x3C). If one of the exception handlers is not implemented, then the address is specified as 0. For more information, see the Cortex-M0 Devices Generic User Guide [10].

The next 16 entries in the vector table are the addresses of the interrupt handlers for the 16 available external interrupts. In this case an external interrupt defines an interrupt generated outside of the Cortex-M0, including from peripherals on the Single Chip Mote. Each address in this section of the vector table corresponds to the one of the interrupt connections to the Cortex-M0 in the Single Chip Mote digital system. The following Verilog code defines a bus containing all of the interrupt connections to the Cortex-M0:

```
assign IRQ = {8'b00000000,RFTIMER_IRQ,RF_IRQ,1'b0,1'b0,ADC_IRQ,1'b0,1'b0,UART_IRQ};
```

Note that unused interrupts are assigned to 0. Their corresponding address in the vector table are also zero. The `UART_IRQ` signal, assigned to `IRQ[0]`, is connected to the `APBUART` module. The first address in the vector table corresponds to `IRQ[0]` and `UART_IRQ`. The first address in the vector table is `UART_Handler`, which is a label referring to a section of code in a later part of `cm0dsasm.s`.

The next section of code contains assembly code for the various exception and interrupt handlers found in the vector table. The reset handler enables the interrupts and then jumps to the `main` function defined in C code:

```
Reset_Handler   PROC
        GLOBAL Reset_Handler
        ENTRY

        LDR     R1, =0xE000E100           ;Interrupt Set Enable Register
        LDR     R0, =0xFF                 ;<- REMEMBER TO ENABLE THE INTERRUPTS!!
        STR     R0, [R1]

        IMPORT  __main
        LDR     R0, =__main
        BX      R0                        ;Branch to __main
                ENDP
```

The other interrupt handlers first mask other interrupts, call the interrupt handler written in C code, and unmask the interrupts before returning:

```
UART_Handler    PROC
        EXPORT UART_Handler
        IMPORT UART_ISR

        PUSH    {R0,LR}

        MOVS     R0, #1 ;          ;MASK all interrupts
        MSR PRIMASK, R0 ;

        BL UART_ISR


        MOVS    R0, #0          ;ENABLE all interrupts
        MSR PRIMASK, R0

        POP     {R0,PC}
                ENDP
```

All interrupt handlers should follow this basic format. In the future it may be necessary to write additional exception handlers in assembly for unused exceptions such as the HardFault Interrupt or NonMaskable Interrupt.

The last part of this file initializes the stack and heap:

```
; User Initial Stack & Heap
                IF      :DEF:__MICROLIB
                EXPORT  __initial_sp
                EXPORT  __heap_base
                EXPORT  __heap_limit
                ELSE
                IMPORT  __use_two_region_memory
                EXPORT  __user_initial_stackheap
__user_initial_stackheap

                LDR     R0, =  Heap_Mem
                LDR     R1, =(Stack_Mem + Stack_Size)
                LDR     R2, = (Heap_Mem +  Heap_Size)
                LDR     R3, = Stack_Mem
                BX      LR

                ALIGN

                ENDIF
```

4.2.2 Memory_Map.h

This file is a C header file containing the addresses of the memory-mapped registers on the Single Chip Mote. The beginning of this file defines the base addresses of each peripheral. The addresses for each register are defined as an offset of the base address for the corresponding peripheral. The information in this file must match the information in the `REGISTERS.vh` Verilog header file found in `scm-digital/src/hw/artix-7/uRobotDigitalController/globalHeaders`. Note that each base address is defined as a hexadecimal number, whereas each register is defined as a dereferenced pointer to an unsigned integer:

```
#define APB_GPIO_BASE         0x53000000
#define GPIO_REG__INPUT       *(unsigned int*)(APB_GPIO_BASE + 0x000000)
#define GPIO_REG__OUTPUT      *(unsigned int*)(APB_GPIO_BASE + 0x040000)
```

This is used so that the registers can be directly read or written using the following syntax:

```
unsigned int i;
i = GPIO_REG__INPUT;
GPIO_REG__OUTPUT = 0x0000000F;
```

These defined registers should be used in any code that writes to or reads from the Single Chip Mote memory-mapped peripherals.

4.2.3 retarget.c

This file is based on an example provided in the ARM Cortex-M0 DesignStart kit. In most ARM-based designs, this file redefines low-level IO routines to work with the hardware on the microcontroller. This file contains the C code needed to implement the `printf` function using UART, by defining `stdout` to point to the UART peripheral, and defining the `fputc` function to write each character to the UART peripheral:

```
struct __FILE {
    unsigned char * ptr;
};

FILE __stdout = {(unsigned char *) APB_UART_BASE};

int fputc(int ch, FILE *f)
{
    return(uart_out(ch));
}

int uart_out(int ch)
{
    unsigned char* UARTPtr;
    UARTPtr = (unsigned char*)APB_UART_BASE;
    *UARTPtr = (char)ch;
    return(ch);
}
```

Note that writing to any of the addresses allocated to the UART peripheral sends the written data. For example, the following code sends the characters “abc” over UART:

```
(unsigned char *)APB_UART_BASE = (char)"a";
(unsigned char *)APB_UART_BASE = (char)"b";
(unsigned char *)APB_UART_BASE = (char)"c";
```

The code in `retarget.c` can also implement the `scanf` function using UART. However, this has not been implemented or tested and will require changes to the `APBUART` hardware module.

4.2.4 main.c

This file contains the C code for the `main` function called by the reset handler. This is where the application software for the Cortex-M0 is written.

4.3 Memory Mapped Peripherals

This section contains the details on writing software that uses the Single Chip Mote peripherals. For each peripheral it is recommended that the accompanying section in chapter 3 is read in order to understand how it operates. Chapter 3 also contains details on the register interface for each peripheral.

4.3.1 Radio Timer

The radio timer is a 500kHz timer that can be used for sending and receiving radio packets without involvement from the Cortex-M0. This timer can also be used as a generic timer without involving the radio. This timer has 8 compare units and 4 capture units. The number of compare and capture units are design parameters. The timer is implemented using a counter that increments every 2 microseconds when enabled. The compare unit generates an interrupt (or sends a trigger to the radio) when the counter reaches a specified value. The capture unit stores the value of the counter when it is activated by the Cortex-M0 or the radio controller.

Before writing software that uses the radio timer on the Single Chip Mote, it is recommended that section 3.35 describing the `RFTIMER` hardware module is read in order to understand how this peripheral operates. Section 3.35.4 has information on the register interface for the radio timer.

Timer Operation

When the timer is enabled, the counter value, stored in the `RFTIMER_REG__COUNT` register, increments by 1 with each rising edge of the timer clock. `RFTIMER_REG__COUNT` can also be written by the software, even when the timer is enabled (as with all other timer registers). When the counter reaches the value in the `RFTIMER_REG__MAX_COUNT` register, it rolls over back to zero. If `RFTIMER_REG__MAX_COUNT` is changed while the counter value is greater than or equal to the new `RFTIMER_REG__MAX_COUNT`, then the counter rolls over to zero. If multiple changes are made to `RFTIMER_REG__COUNT` and `RFTIMER_REG__MAX_COUNT` during a single timer clock cycle, only the last change takes effect on the next rising edge of the timer clock.

Timer Control

The `RFTIMER_REG__CONTROL` register has bits to enable the timer, enable interrupts to the Cortex-M0, and reset the counter. If multiple changes are made to `RFTIMER_REG__CONTROL` during a single timer clock cycle, only the last change takes effect on the next rising edge of the timer clock.

Using a Compare Unit

To use a compare unit, first write the value that is compared with the timer to the RFTIMER_REG__COMPAREx register (where x designates the particular compare unit). Then write to the RFTIMER_REG__COMPAREx_CONTROL register to enable the Cortex-M0 interrupt or any triggers to the radio on a compare match. For more information on the radio triggers, see sections 4.3.2, 3.25, and 3.35.

Using a Capture Unit

To use a capture unit, write to the RFTIMER_REG__CAPTUREx_CONTROL register (where x designates the particular capture unit) to select the inputs to the capture unit and enable the Cortex-M0 interrupt. Any of the selected inputs can trigger the capture unit. In particular, writing a 1 to the CAPTURE_NOW bit of the RFTIMER_REG__CAPTUREx_CONTROL register, when the Cortex-M0 input is enabled, triggers a capture. For more information on the inputs to the capture units, see sections 3.25, and 3.35.

Triggering a capture copies the counter value into the RFTIMER_REG__CAPTUREx register. The counter value stays in the RFTIMER_REG__CAPTUREx register until the next trigger, where it is overwritten. The RFTIMER_REG__CAPTUREx register can also be cleared by writing a 1 to the CLEAR bit of the RFTIMER_REG__CAPTUREx_CONTROL register.

Interrupts

The radio timer has one interrupt to the Cortex-M0. This interrupt is the bitwise OR of all of the bits in the RFTIMER_REG__INT register, which indicates the source of the interrupt within the radio timer. Each compare unit has one bit in the RFTIMER_REG__INT register for an interrupt on a compare match. Each capture unit has two bits in the RFTIMER_REG__INT register for an interrupt and an overflow flag when a capture is triggered. Each bit in the RFTIMER_REG__INT register must be cleared (via the RFTIMER_REG__INT_CLEAR register) in the interrupt service routine in order to prevent another interrupt. If any bits in the RFTIMER_REG__INT register are 1 when the interrupt service routine returns, it will be executed again. If the interrupt service routine does not disable the counter after the interrupt, more bits in the RFTIMER_REG__INT register may be set to 1 while the interrupt service routine is running.

Register Write Delays

As noted in previous sections, there are several memory-mapped registers that can be written by the Cortex-M0 at any time that will only update on the rising edge of the timer clock. This means that reading a register immediately after it is written may not return the same value that was written. The system clock is 5MHz while the timer clock is 500kHz, 10 times slower. Therefore, most of the time the following comparison will return 0:

```
RFTIMER_REG__COMPARE0 = 0x33;
if (RFTIMER_REG__COMPARE0 == 0x33) {
    return 1;
} else {
    return 0;
}
```

However, the following code, which adds a delay after the register is written, will return 1:

```
int i;
RFTIMER_REG__COMPARE0 = 0x33;
for (i = 0; i < 10; i++);
if (RFTIMER_REG__COMPARE0 == 0x33) {
    return 1;
} else {
    return 0;
}
```

Example Code

The following example code creates a small string to send over a radio packet, and tells the radio controller to start copying packet data into memory. Immediately after the code starts the timer, and uses compare unit 0 to tell the radio controller to send the packet after 1 second and trigger a Cortex-M0 interrupt. The code uses capture unit 1 to capture when the last bit of the start of frame delimiter (SFD) of the packet is sent and trigger a Cortex-M0 interrupt. The code also uses capture unit 2 to capture when the last bit of the packet is sent and trigger a Cortex-M0 interrupt. Both the compare and capture units used in this code have their Cortex-M0 interrupts enabled. This code also demonstrates one possible implementation of the interrupt service routine for the radio timer.

```
#include <stdio.h>
#include <rt_misc.h>
#include <stdlib.h>
#include "Memory_map.h"

int flag;

int main(void) {

    char send_packet[127];
    char packet_length;
    char i;

    flag = 0;

    // Compose message
    sprintf(send_packet, "This is a test packet");

    // Set radio controller registers
    RFCONTROLLER_REG__TX_DATA_ADDR = &send_packet[0];
    RFCONTROLLER_REG__TX_PACK_LEN = 21;

    // Set radio timer registers
    RFTIMER_REG__COMPARE0 = 500000; // Set compare unit 0 value to 1s
    RFTIMER_REG__COMPARE0_CONTROL = 0xB;  // Binary value 001011
                                          // Enable compare unit
                                          // Enable interrupt
                                          // Enable the TX_SEND output to radio
    RFTIMER_REG__CAPTURE1_CONTROL = 0x9;  // Binary value 000001001
                                          // Enable interrupt
                                          // Enable TX_SFD_DONE input
    RFTIMER_REG__CAPTURE2_CONTROL = 0x11; // Binary value 000010001
                                          // Enable interrupt
                                          // Enable TX_SEND_DONE input

    // Start sending packet with TX_LOAD
    RFCONTROLLER_REG__CONTROL = 0x01;
    // Start the timer, enable interrupts, and reset counter
    RFTIMER_REG__CONTROL = 0x7;

    // Wait until packet is finished sending and
```

```
    // the radio timer prints all values
    while (flag == 0) {}
    printf("done\n");
}

void RFTIMER_ISR() {

    // Read the interrupt register
    unsigned int interrupt = RFTIMER_REG__INT;

    // Respond to different interrupt sources
    if (interrupt & 0x00000001) printf("Telling radio to send packet with TX_SEND\n
        ");
    if (interrupt & 0x00000200) printf("TX SFD DONE at 0x%x\n",
        RFTIMER_REG__CAPTURE1);
    if (interrupt & 0x00000400) { printf("TX SEND DONE at 0x%x\n",
        RFTIMER_REG__CAPTURE2); flag = 1; }

    // Clear the interrupt register
    RFTIMER_REG__INT_CLEAR = interrupt;
}
```

4.3.2 Radio Controller and DMA

The radio controller and the DMA work together to send and receive packets. Most of the details behind sending and receiving packets are handled in hardware; the software only needs to set a few register values and then tell the radio controller to begin sending or listening for a packet.

Before writing software that uses the radio controller and the DMA on the Single Chip Mote, it is recommended that section 3.25 describing the `RFcontroller` hardware module and section 3.24 describing the `DMA_V2` module is read in order to understand how these peripherals operate. Section 3.25.4 has information on the register interface for the radio controller and section 3.24.4 has information on the register interface for the DMA.

Radio Controller Initialization

It is recommended that the radio controller's interrupt registers are set during any initialization code that runs right after booting. Interrupt registers can also be set before sending or receiving a packet. The registers which configure the interrupt are `RFCONTROLLER_REG__INT_CONFIG` and `RFCONTROLLER_REG__ERROR_CONFIG`.

Given that there are multiple sources for an interrupt from the radio controller, the interrupt service routine should be designed react appropriately to any of the enabled interrupt sources.

Packet Storage in Memory

Packet data for transmission can come from any place in the data memory, and data from a received packet can be stored in any place in the data memory. The radio controller and DMA only require that a continuous, sequential, word-aligned section of memory is dedicated for transmitted or received packets, and that the section of memory is large enough to fit an entire packet (127 bytes for send, 130 bytes for receive). It is not sufficient to declare an array of bytes inside of the function that uses the radio controller and provide the first address to the radio controller and/or DMA. This is because a packet may be sent or received long after

that function returns, and the memory on the stack may be re-used for another function. Therefore, the available options are: declare an array of bytes inside a function and ensure that the function returns after the packet is sent or received, allocate an array of bytes on the heap and pass the pointer to any functions that use the radio controller, or create an array of bytes as a global variable accessible to any function that use the radio controller.

Sending a Packet

1. All packet data should be stored in a continuous, sequential, word-aligned section of data memory. This can be done by creating an array of chars with length equal to that of the packet data in bytes.

2. Write the starting address of the packet data to the `RFCONTROLLER_REG__TX_DATA_ADDR` register.

3. Write the length of the packet, in bytes, to the `RFCONTROLLER_REG__PACK_LEN` register.

4. Set the `TX_LOAD` bit of the `RFCONTROLLER_REG__CONTROL` register, or use the radio timer to send a `TX_LOAD` signal/trigger. This will begin copying the packet data from memory.

5. Wait for the packet data to finish copying. This can be indicated either by the `TX_LOAD_DONE` interrupt, or by monitoring/polling the `TX_STATE` bits of the `RFCONTROLLER_REG__STATUS` register.

6. Set the `TX_SEND` bit of the `RFCONTROLLER_REG__CONTROL` register, or use the radio timer to send a `TX_SEND` signal/trigger. This will begin sending the packet data.

7. Wait for the packet to finish sending. This can be indicated by the `TX_SEND_DONE` interrupt, or by monitoring/polling the `TX_STATE` bits of the `RFCONTROLLER_REG__STATUS` register.

Simulation shows that it takes at most $85.6\mu s$ to copy the packet data from memory to the radio controller, from the time when the `TX_LOAD` signal is sent to the time when the last byte of data is copied (indicated by the `TX_LOAD_DONE` interrupt), assuming the largest possible packet has a payload of 127 bytes. Simulations also show that it takes anywhere between $161.5\mu s$ and $162\mu s$ from the time when the `TX_SEND` trigger is sent to the time when the last bit of the start-of-frame delimiter is sent (indicated by the `TX_SFD_DONE` interrupt). Simulations also show that it takes at most $4322.5\mu s$ to send a packet, from the time when the `TX_SEND` trigger is sent to the time when the last bit of the packet is sent (indicated by the `TX_SEND_DONE` interrupt), assuming the largest possible packet has a payload of 127 bytes.

Receiving a Packet

1. Software should set aside a continuous, word-aligned section of data memory large enough to store an entire packet. The first byte of this section will contain the packet length, followed by up to 127 bytes of packet data. The two bytes

following the packet data are the CRC. This requires a maximum of 130 bytes. This can be done by creating an array of chars with length equal to 130 bytes.

2. Write the starting address of this section of memory into the DMA_REG__RF-_RX_ADDR register.

3. Set the RX_START bit of the RFCONTROLLER_REG__CONTROL register, or use the radio timer to send an RX_START signal/trigger. This will enable listening for packets.

4. Wait for a packet to be detected. This can be indicated by the RX_SFD_DONE interrupt, or by monitoring/polling the RX_STATE bits of the RFCONTROLLER_REG-__STATUS register.

5. If a packet is detected, wait for the data decoded and copied into memory. This can be indicated by the RX_DONE interrupt, or by monitoring/polling the RX_STATE bits of the RFCONTROLLER_REG__STATUS register.

6. If a packet is not detected, the RX mode can be exited by setting the RX_STOP bit of the RFCONTROLLER_REG__STATUS register.

Example Code

The following example code initializes the radio controller and DMA registers, sends a packet, and waits for a response. This code also demonstrates one possible implementation of the interrupt service routine for the radio controller.

```
#include <stdio.h>
#include <rt_misc.h>
#include <stdlib.h>
#include "Memory_map.h"

int flag1;
int flag2;
int flag3;

int main(void) {

    char send_packet[127];
    char recv_packet[130];
    char packet_length;
    char i;

    // Set radio controller and DMA registers
    RFCONTROLLER_REG__TX_DATA_ADDR = &send_packet[0];
    DMA_REG__RF_RX_ADDR = &recv_packet[0];
    RFCONTROLLER_REG__INT_CONFIG = 0x01D;      // Binary value 00000_00000_11111
                                               // all interrupts enabled
                                               // no rftimer pulses
                                               // no interrupts masked
    RFCONTROLLER_REG__ERROR_CONFIG = 0x01F;    // Binary value 00000_11111
                                               // all errors enabled
                                               // no errors masked

    // Set state
    flag1 = 0;
    flag2 = 0;
    flag3 = 0;
    sprintf(send_packet, "This is a test packet");
    RFCONTROLLER_REG__TX_PACK_LEN = 21;

    // Start sending packet with TX_LOAD
    RFCONTROLLER_REG__CONTROL = 0x01;
```

```
    // Wait for TX_LOAD_DONE
    while (flag1 == 0) {}

    // Start sending packet with TX_START
    RFCONTROLLER_REG__CONTROL = 0x02;

    // Wait for packet to finish sending
    while (flag2 == 0) {}

    // Start listening for a packet with RX_START
    RFCONTROLLER_REG__CONTROL = 0x04;

    // Wait for a response
    while (flag3 == 0) {}

    // Get the length of received packet
    packet_length = recv_packet[0];

    // Print out the received packet
    for (i=0; i < packet_length; i++) {
        printf("%c",recv_packet[i+1]);
    }
    printf("\n");

    return 0;
}

void RF_ISR() {

    // Read the interrupt and error registers
    unsigned int interrupt = RFCONTROLLER_REG__INT;
    unsigned int error     = RFCONTROLLER_REG__ERROR;

    // Respond to different interrupt sources
    if (interrupt & 0x00000001) { printf("TX LOAD DONE\n"); flag1 = 1; }
    if (interrupt & 0x00000002) printf("TX SFD DONE\n");
    if (interrupt & 0x00000004) { printf("TX SEND DONE\n"); flag2 = 1; }
    if (interrupt & 0x00000008) printf("RX SFD DONE\n");
    if (interrupt & 0x00000010) { printf("RX DONE\n"); flag3 = 1; }

    // Respond to different error sources
    if (error != 0) {
        if (error & 0x00000001) printf("TX OVERFLOW ERROR\n");
        if (error & 0x00000002) printf("TX CUTOFF ERROR\n");
        if (error & 0x00000004) printf("RX OVERFLOW ERROR\n");
        if (error & 0x00000008) printf("RX CRC ERROR\n");
        if (error & 0x00000010) printf("RX CUTOFF ERROR\n");
        // Clear the error register
        RFCONTROLLER_REG__ERROR_CLEAR = error;
    }

    // Clear the interrupt register
    RFCONTROLLER_REG__INT_CLEAR = interrupt;
}
```

4.3.3 UART

The UART peripheral allows for data to be transferred between the Single Chip Mote and a computer using a 3-wire RS-232 interface (more commonly known as a serial port). The current hardware implementation accepts 8 data bits, and adds 1 start bit and 1 stop bit to create a data frame. The hardware does not support extra parity bits or flow control. The baud rate is a design parameter and is currently set to 19200.

Since each data frame contains 8 data bits, it is useful to think of each UART data frame as transmitting one ASCII character the size of a `char`. The current

test software accepts short 3-letter commands (delimited with a newline character) over UART and uses the `printf` function to send strings of characters to the serial terminal on the computer in a human-readable form. That being said, the UART interface can send and receive any series of 8-bit data frames; the software does not have to interpret these sets of 8 bits as ASCII characters and instead can use their numerical values.

Before writing software that uses the UART peripheral on the Single Chip Mote, it is recommended that section 5.7.1 describing the `UART` hardware module is read in order to understand how this peripheral operates. Section 3.40.4 has information on the UART register interface.

The UART peripheral has two hardware FIFOs to store the transmitted and received data. All writes to this peripheral (regardless of the actual address) copy the lower 8-bits into the FIFO for transmission. Too many writes in succession can overflow the FIFO and there is currently no indication to the software when the FIFO is full. All reads from this peripheral (regardless of the actual address) read 8-bits of data out of the receive FIFO, even when there is no valid data in the FIFO. The hardware interrupt from the UART module indicates that there is data in the receive FIFO. Therefore, the UART interrupt service routine is executed when there is data in the FIFO. However, there is no indication of how much data is in the FIFO, and therefore the interrupt service routine must read only one value from the FIFO. The interrupt service routine will continue to be executed until the receive FIFO is empty.

While the behavior described above is vulnerable to a variety of errors, this behavior is part of the example hardware provided in the ARM Cortex-M0 DesignStart kit. Further hardware modifications are required to make this module more robust and provide additional feedback to the software.

The simplest way to transmit a single 8-bit value over UART is to write directly to the UART peripheral:

```
APBUART_REG__TX_DATA = 0x32;
APBUART_REG__TX_DATA = 0xBEEF; // Only the lower 8 bits are transmitted
```

The simplest way to transmit a string over UART is to use `printf` (assuming that `retarget.c` is written properly):

```
printf("Is this the real life? Is this just fantasy? Caught in a landslide, no
    escape from reality.");
```

The ideal method for dealing with received data is to write the interrupt service for the UART peripheral assuming that it reads one 8-bit value at a time. The value can then be stored in a buffer to be processed by code outside of the interrupt service routine. Alternatively, the interrupt service routine can also read the data in the buffer and perform simple actions:

```
void UART_ISR(){

    static char buff[4] = {0x0, 0x0, 0x0, 0x0};
    char inChar;

    inChar = UART_REG__RX_DATA;
    buff[3] = buff[2];
    buff[2] = buff[1];
    buff[1] = buff[0];
    buff[0] = inChar;

    // Sends TX_LOAD signal to radio controller
```

```
    if ( (buff[3]=='l') && (buff[2]=='o') && (buff[1]=='d') && (buff[0]=='\n') ) {
        RFCONTROLLER_REG__CONTROL = 0x1;
    // Sends TX_SEND signal to radio controller
    } else if ( (buff[3]=='s') && (buff[2]=='n') && (buff[1]=='d') && (buff[0]=='\n
        ') ) {
        RFCONTROLLER_REG__CONTROL = 0x2;
    // Sends RX_START signal to radio controller
    } else if ( (buff[3]=='r') && (buff[2]=='c') && (buff[1]=='v') && (buff[0]=='\n
        ') ) {
        DMA_REG__RF_RX_ADDR = &recv_packet[0];
        RFCONTROLLER_REG__CONTROL = 0x4;
    // Sends RX_STOP signal to radio controller
    } else if ( (buff[3]=='e') && (buff[2]=='n') && (buff[1]=='d') && (buff[0]=='\n
        ') ) {
        RFCONTROLLER_REG__CONTROL = 0x8;
    // Sends RF_RESET signal to radio controller
    } else if ( (buff[3]=='r') && (buff[2]=='s') && (buff[1]=='t') && (buff[0]=='\n
        ') ) {
        RFCONTROLLER_REG__CONTROL = 0x10;
    // Unknown command
    } else if (inChar=='\n'){
        printf("unknown command\n");
    }
}
```

4.3.4 ADC Controller

The analog-to-digital converter (ADC) takes an analog voltage at one of the input pins to the Single Chip Mote and converts it to a 10-bit value. Most of the details behind the operation of the ADC are taken care of in ADC controller hardware; the software only needs to initiate a conversion and wait for the interrupt indicating that the conversion is complete. Alternatively, the interrupt can be disabled (through the interrupt set-enable register of the Cortex-M0), and the ADC controller can be polled by the software to check when the conversion is complete.

Before writing software that uses the ADC on the Single Chip Mote, it is recommended that section 3.41 describing the `APBADC_V2` hardware module is read in order to understand how this peripheral operates. Section 3.41.4 has information on the register interface for the ADC.

The following code demonstrates how to start a conversion and wait for the interrupt to indicate that the conversion is complete:

```
#include <stdio.h>
#include <rt_misc.h>
#include <stdlib.h>
#include "Memory_map.h"

int flag;

int main(void) {

    flag = 0;

    // Starting conversion
    ADC_REG__START = 0x1;

    // Wait until the conversion is finished
    while (flag == 0) {}
    printf("done\n");
}

void ADC_ISR() {

    printf("Conversion complete: 0x%x\n", ADC_REG__DATA);
    flag = 1;
```

```
}
```

The following code demonstrates how to start a conversion and check when the conversion is complete using polling:

```
// Starting conversion
ADC_REG__START = 0x1;

// Wait until the conversion is finished
while (ADC_REG__START == 1) {}
printf("Conversion complete: 0x%x\n", ADC_REG__DATA);
```

4.3.5 Analog Configuration Registers

The analog configuration registers are a set of programmable 16-bit read-write registers used to modify and fine-tine the analog and radio circuits. These registers are not needed for the FPGA version of the Single Chip Mote, since the analog and radio circuits are programmed separately. However, the ASIC version of the Single Chip Mote will directly connect the outputs of these registers to the analog and radio circuits. Each register is 16 bits wide and the number of registers is a design parameter.

The following code demonstrates how to write to and read from one of the analog configuration registers:

```
unsigned int i;
unsigned int j;

ANALOG_CFG_REG__0 = 0xBEEF;

i = 0xFFFFFFFF;
ANALOG_CFG_REG__0 = i; // Only the lower 16 bits are written
j = ANALOG_CFG_REG__0; // Should be equal to 0x0000FFFF
                       // Upper 16 bits are always zero
```

Note that unsigned integers (32-bits) are used since `ANALOG_CFG_REG__0` is defined in `Memory_Map.h` as a dereferenced pointer to an unsigned integer. It may be useful to change the definition of `ANALOG_CFG_REG__0` if a 16-bit data type is preferred.

4.3.6 General-Purpose Input and Output Registers

The read-only general-purpose input register is used to read the values of the digital inputs to the Single Chip Mote. The number of inputs is a design parameter, with a maximum value of 16. On the FPGA version of the Single Chip Mote, these inputs are connected to either switches or input pins on the FPGA board. On the ASIC version of the Single Chip Mote, these inputs are connected to input pads on the chip.

The read-write general-purpose output register is used to write values to the digital outputs from the Single Chip Mote. The number of outputs is a design parameter, with a maximum value of 16. On the FPGA version of the Single Chip Mote, these outputs are connected to either LEDs or output pins on the FPGA board. On the ASIC version of the Single Chip Mote, these outputs are connected to output pads on the chip.

The following code demonstrates how to read from the input register, and read from or write to the output register:

```
unsigned int i;
unsigned int j;

i = GPIO_REG__INPUT; // Assuming there are n inputs, the
                     // lower n bits contain the input
                     // data, and thehe upper 16-n bits
                     // are zero.

GPIO_REG__OUTPUT = 0x123; // Assuming there are m outputs, the
                          // lower m bits are used and the
                          // upper 16-m bits are ignored
j = GPIO_REG__OUTPUT;     // Assuming there are m outputs, the
                          // lower m bits contain the output
                          // values and the upper 16-m bits
                          // are zero.
```

4.4 Current Demo Software

The software currently used to test and demonstrate the Single Chip Mote digital system on an FPGA is found in the following Keil project: `scm-digital/proj/keil/uRobotDigitalController/code.uvprojx`. This software responds to several different commands sent to the Single Chip Mote via UART:

`cpy <string>\n` Copy `<string>` to `send_packet` buffer, which contains the data to be transmitted via the radio.

`lod\n` Tell the radio controller to copy the data in `send_packet` for transmission using the `TX_LOAD` bit in the `RFCONTROLLER_REG__CONTROL` register.

`snd\n` Tell the radio controller to transmit a packet using the `TX_SEND` bit in the `RFCONTROLLER_REG__CONTROL` register.

`rcv\n` Tell the radio controller to listen for incoming packets using the `RX_START` bit in the `RFCONTROLLER_REG__CONTROL` register.

`end\n` Tell the radio controller to stop listening for incoming packets using the `RX_STOP` bit in the `RFCONTROLLER_REG__CONTROL` register.

`rst\n` Reset the radio controller using the `RF_RESET` bit in the `RFCONTROLLER_REG__CONTROL` register.

`sta\n` Print the value of the radio controller status register, `RFCONTROLLER_REG__STATUS`.

`adc\n` Initiate an ADC conversion using the `ADC_REG__START` register.

`atx\n` Auto-TX: Wait 0.5s, then send a `TX_LOAD` signal to the radio controller using the radio timer. Wait 0.5s, then send a `TX_SEND` signal to the radio from the radio timer. Also use the radio timer to capture when the last bit of the SFD is sent and when the last bit of the packet is sent.

`arx\n` Auto-RX: Wait 0.5s, then send a `RX_START` signal to the radio controller using the radio timer. Also use the radio timer to capture when the SFD is detected and when the packet is copied into memory.

`rrt\n` Resets the radio timer compare and capture units. This command should be run after `atx` or `arx` since the compare and capture units continue to run after the command is complete.

`cm0dsasm.s`, `retarget.c`, `Memory_map.h`

The contents of these files match their descriptions in previous sections of this chapter. To recap:

- `cm0dsasm.s` contains the assembly code that describes the vector table, reset handler, and interrupt handlers.
- `retarget.c` contains the C code used to implement the `printf` function using UART.
- `Memory_map.h` contains define statements for all of the peripheral memory-mapped registers.

`rf_global_vars.h`

This file contains the declarations for the `send_packet` and `recv_packet` buffers:

```
// Set aside sections of address space for the packet
char send_packet[127] __attribute__ ((aligned (4)));
char recv_packet[130] __attribute__ ((aligned (4)));
```

These buffers are defined as global variables (and are not stored on the stack or the heap). The `aligned(4)` attribute ensures that the first address of these buffers are word-aligned, which is required for the radio controller and DMA to function correctly. This file is included in any other file that contains a function using `send_packet` or `recv_packet`, where these two variables are defined using the `extern` keyword:

```
#include "rf_global_vars.h"

extern char send_packet[127];
extern char recv_packet[130];
```

This file is included in `main.c` and `Int_Handlers.h`. Using globally defined buffers for storing packet data ensures that the addresses dedicated to these buffers are not overwritten once a the function using them returns.

`main.c`

This file contains the main function that executes after reset:

```
int main(void) {
    int t;
    int j;

    printf("\nWelcome to the uRobot Digital Controller\n\n");

    //SYSTEM INITIALIZATION
    RFCONTROLLER_REG__TX_DATA_ADDR = &send_packet[0];
    RFCONTROLLER_REG__INT_CONFIG = 0x3FF;   // Enable all interrupts and pulses to
        radio timer
    RFCONTROLLER_REG__ERROR_CONFIG = 0x1F;  // Enable all errors
    RFTIMER_REG__MAX_COUNT = 0xFFFFFFFF;
    RFTIMER_REG__CONTROL = 0x7;
```

```
    printf("Initialization complete\n");

    while(1)  {
        j = GPIO_REG__INPUT;
        for(t=0;t<100;t++);
        GPIO_REG__OUTPUT = j;
    }
}
```

The main function first initializes the radio controller by setting the address for transmitted packet data, and then sets up the interrupt and error configuration registers. After that the function goes into an infinite loop. This function loops forever without doing anything useful since the commands are sent via UART and each command is executed in the UART interrupt service routine. Inside of the loop, the general-purpose digital inputs are sampled, and then copied onto the general-purpose digital outputs. On an FPGA the inputs are connected to switches and the outputs are connected to LEDs, and so the loop code makes the LEDs match the switches. If the switches are changed and the LEDs do not change, then the microprocessor may be stuck in some unrecoverable state.

`Int_Handlers.h`

This file contains the interrupt service routines for the UART, ADC, radio controller, and radio timer. These service routines are executed each time these peripherals trigger an interrupt.

The UART interrupt service routine is called when a single 8-bit value is transmitted to the Single Chip Mote via UART. This demo software interprets each 8-bit value as a single ASCII character, and a series of 4 ASCII characters can indicate a particular command. The UART interrupt service routine has a static buffer to store the last 4 received characters. If those 4 characters match a command, that command is executed. Most commands are a series of three letters ending with a newline character (for example, `rcv\n` is the command to send the `RX_START` signal to the radio controller). The only exception is the copy command, which is used to copy a string into the `send_packet` buffer. This command is denoted by the characters 'c', 'p', and 'y', followed by a space. When "`cpy `" is detected, this sets a static variable called `waiting_for_end_of_copy`, which indicates that each subsequent character written via UART should be copied into `send_buffer`, until another newline character is written.

```
void UART_ISR(){
    static char i=0;
    static char buff[4] = {0x0, 0x0, 0x0, 0x0};
    static char waiting_for_end_of_copy = 0;
    char inChar;

    inChar = UART_REG__RX_DATA;
    buff[3] = buff[2];
    buff[2] = buff[1];
    buff[1] = buff[0];
    buff[0] = inChar;

    // If we are still waiting for the end of a load command
    if (waiting_for_end_of_copy) {
        if (inChar=='\n'){
            int j=0;
            printf("copying string of size %u to send_packet: ", i);
            for (j=0; j < i; j++) {
                printf("%c",send_packet[j]);
```

```
            }
            printf("\n");
            RFCONTROLLER_REG__TX_PACK_LEN = i;
            i = 0;
            waiting_for_end_of_copy = 0;
        } else if (i < 127) {
            send_packet[i] = inChar;
            i++;
        } else {
            printf("Input exceeds maximum packet size\n");
        }
    } else { //If waiting for a command
        // Copies string from UART to send_packet
        if ( (buff[3]=='c') && (buff[2]=='p') && (buff[1]=='y') && (buff[0]==' ') )
             {
            waiting_for_end_of_copy = 1;
        // Sends TX_LOAD signal to radio controller
        } else if ( (buff[3]=='l') && (buff[2]=='o') && (buff[1]=='d') && (buff
            [0]=='\n') ) {
            RFCONTROLLER_REG__CONTROL = 0x1;
            printf("TX LOAD\n");
        // Sends TX_SEND signal to radio controller
        } else if ( (buff[3]=='s') && (buff[2]=='n') && (buff[1]=='d') && (buff
            [0]=='\n') ) {
            RFCONTROLLER_REG__CONTROL = 0x2;
            printf("TX SEND\n");
        // Sends RX_START signal to radio controller
        } else if ( (buff[3]=='r') && (buff[2]=='c') && (buff[1]=='v') && (buff
            [0]=='\n') ) {
            printf("Recieving\n");
            DMA_REG__RF_RX_ADDR = &recv_packet[0];
            RFCONTROLLER_REG__CONTROL = 0x4;
        // Sends RX_STOP signal to radio controller
        } else if ( (buff[3]=='e') && (buff[2]=='n') && (buff[1]=='d') && (buff
            [0]=='\n') ) {
            RFCONTROLLER_REG__CONTROL = 0x8;
            printf("RX STOP\n");
        // Sends RF_RESET signal to radio controller
        } else if ( (buff[3]=='r') && (buff[2]=='s') && (buff[1]=='t') && (buff
            [0]=='\n') ) {
            RFCONTROLLER_REG__CONTROL = 0x10;
            printf("RF RESET\n");
        // Returns the status register of the radio controller
        } else if ( (buff[3]=='s') && (buff[2]=='t') && (buff[1]=='a') && (buff
            [0]=='\n') ) {
            int status = RFCONTROLLER_REG__STATUS;
            printf("status register is 0x%x\n", status);
        // Initiates an ADC conversion
        } else if ( (buff[3]=='a') && (buff[2]=='d') && (buff[1]=='c') && (buff
            [0]=='\n') ) {
            ADC_REG__START = 0x1;
            printf("starting ADC conversion\n");
        // Uses the radio timer to send TX_LOAD in 0.5s, TX_SEND in 1s, capture
            when SFD is sent and capture when packet is sent
        } else if ( (buff[3]=='a') && (buff[2]=='t') && (buff[1]=='x') && (buff
            [0]=='\n') ) {
            unsigned int t = RFTIMER_REG__COUNTER + 0x3D090;
            RFTIMER_REG__COMPARE0 = t;
            RFTIMER_REG__COMPARE1 = t + 0x3D090;
            printf("%x\n", RFTIMER_REG__COMPARE0);
            printf("%x\n", RFTIMER_REG__COMPARE1);
            RFTIMER_REG__COMPARE0_CONTROL = 0x5;
            RFTIMER_REG__COMPARE1_CONTROL = 0x9;
            RFTIMER_REG__CAPTURE0_CONTROL = 0x9;
            RFTIMER_REG__CAPTURE1_CONTROL = 0x11;
            printf("Auto TX\n");
        // Uses the radio timer to send RX_START in 0.5s, capture when SFD is
            received and capture when packet is received
        } else if ( (buff[3]=='a') && (buff[2]=='r') && (buff[1]=='x') && (buff
            [0]=='\n') ) {
            RFTIMER_REG__COMPARE0 = RFTIMER_REG__COUNTER + 0x3D090;
            RFTIMER_REG__COMPARE0_CONTROL = 0x11;
            RFTIMER_REG__CAPTURE0_CONTROL = 0x21;
```

```
            RFTIMER_REG__CAPTURE1_CONTROL = 0x41;
            DMA_REG__RF_RX_ADDR = &recv_packet[0];
            printf("Auto RX\n");
        // Reset the radio timer compare and capture units
        } else if ( (buff[3]=='r') && (buff[2]=='r') && (buff[1]=='t') && (buff
            [0]=='\n') ) {
            RFTIMER_REG__COMPARE0_CONTROL = 0x0;
            RFTIMER_REG__COMPARE1_CONTROL = 0x0;
            RFTIMER_REG__CAPTURE0_CONTROL = 0x0;
            RFTIMER_REG__CAPTURE1_CONTROL = 0x0;
            printf("Radio timer reset\n");
        // Unknown command
        } else if (inChar=='\n'){
            printf("unknown command\n");
        }
    }
}
```

The ADC interrupt service routine is called when a conversion is complete. The interrupt service routine prints out the conversion result:

```
void ADC_ISR() {
    printf("Conversion complete: 0x%x\n", ADC_REG__DATA);
}
```

The radio controller interrupt service routine prints the reason for the interrupt based on the `RFCONTROLLER_REG__INT` and `RFCONTROLLER_REG__ERROR` registers and then clears these registers. If the interrupt was triggered after a packet was received, the packet data is also printed:

```
void RF_ISR() {

    unsigned int interrupt = RFCONTROLLER_REG__INT;
    unsigned int error     = RFCONTROLLER_REG__ERROR;

    if (interrupt & 0x00000001) printf("TX LOAD DONE\n");
    if (interrupt & 0x00000002) printf("TX SFD DONE\n");
    if (interrupt & 0x00000004) printf("TX SEND DONE\n");
    if (interrupt & 0x00000008) printf("RX SFD DONE\n");
    if (interrupt & 0x00000010) {
        int i;
        char num_bytes_rec = recv_packet[0];
        char *current_byte_rec = recv_packet+1;
        printf("RX DONE\n");
        printf("Received packet of size %d: ", num_bytes_rec);
        for (i=0; i < num_bytes_rec; i++) {
            printf("%c",current_byte_rec[i]);
        }
        printf("\n");
    }

    if (error == 0) {
        if (error & 0x00000001) printf("TX OVERFLOW ERROR\n");
        if (error & 0x00000002) printf("TX CUTOFF ERROR\n");
        if (error & 0x00000004) printf("RX OVERFLOW ERROR\n");
        if (error & 0x00000008) printf("RX CRC ERROR\n");
        if (error & 0x00000010) printf("RX CUTOFF ERROR\n");
        RFCONTROLLER_REG__ERROR_CLEAR = error;
    }
    RFCONTROLLER_REG__INT_CLEAR = interrupt;
}
```

The radio timer interrupt service routine reads the `RFTIMER_REG__INT` register to find the source of the interrupt. If the interrupt is due to a compare unit, it prints which compare unit triggered the interrupt. If the interrupt is due to a capture unit, it prints which capture unit triggered the interrupt and the captured timer value. It then clears the `RFTIMER_REG__INT` register:

```
void RFTIMER_ISR() {

    unsigned int interrupt = RFTIMER_REG__INT;

    if (interrupt & 0x00000001) printf("COMPARE0 MATCH\n");
    if (interrupt & 0x00000002) printf("COMPARE1 MATCH\n");
    if (interrupt & 0x00000004) printf("COMPARE2 MATCH\n");
    if (interrupt & 0x00000008) printf("COMPARE3 MATCH\n");
    if (interrupt & 0x00000010) printf("COMPARE4 MATCH\n");
    if (interrupt & 0x00000020) printf("COMPARE5 MATCH\n");
    if (interrupt & 0x00000040) printf("COMPARE6 MATCH\n");
    if (interrupt & 0x00000080) printf("COMPARE7 MATCH\n");
    if (interrupt & 0x00000100) printf("CAPTURE0 TRIGGERED AT: 0x%x\n",
        RFTIMER_REG__CAPTURE0);
    if (interrupt & 0x00000200) printf("CAPTURE1 TRIGGERED AT: 0x%x\n",
        RFTIMER_REG__CAPTURE1);
    if (interrupt & 0x00000400) printf("CAPTURE2 TRIGGERED AT: 0x%x\n",
        RFTIMER_REG__CAPTURE2);
    if (interrupt & 0x00000800) printf("CAPTURE3 TRIGGERED AT: 0x%x\n",
        RFTIMER_REG__CAPTURE3);
    if (interrupt & 0x00001000) printf("CAPTURE0 OVERFLOW AT: 0x%x\n",
        RFTIMER_REG__CAPTURE0);
    if (interrupt & 0x00002000) printf("CAPTURE1 OVERFLOW AT: 0x%x\n",
        RFTIMER_REG__CAPTURE1);
    if (interrupt & 0x00004000) printf("CAPTURE2 OVERFLOW AT: 0x%x\n",
        RFTIMER_REG__CAPTURE2);
    if (interrupt & 0x00008000) printf("CAPTURE3 OVERFLOW AT: 0x%x\n",
        RFTIMER_REG__CAPTURE3);

    RFTIMER_REG__INT_CLEAR = interrupt;
}
```

Connecting Two FPGA Boards for Packet Transmission

This demo software uses the radio controller to send and receive packets. However, the Single Chip Mote design on an FPGA does not have an actual radio. However, two FPGA boards loaded with the Single Chip Mote digital system can be connected directly to one another to simulate packet transmission. See section 5.8 for instructions on how to connect two FPGA boards together for this purpose.

Chapter 5

Bootloader

This chapter covers the details of the Single Chip Mote bootloader, the specialized hardware and software used to load compiled C code onto the instruction memory for the ARM Cortex-M0. The bootloader requires four distinct hardware/software components:

Instruction ROM on Single Chip Mote This is a read-only memory located in the `AHBIMEM` module. This ROM contains the basic software, referred to from now on as firmware, that runs on the ARM Cortex-M0 while the main software is loaded into the instruction RAM.

Instruction RAM on Single Chip Mote This is a synchronous SRAM located in the `AHBIMEM` module to hold the main software for the Single Chip Mote. This RAM has no valid data when the FPGA is powered on, and must be loaded with the right software through bootloading. This RAM is written either from the ARM Cortex-M0 via the AHB bus, or from external hardware using the 3 Wire Bus.

Bootload Hardware on Nexys 3 This is a custom FPGA project meant for the Digilent Nexys 3 board. The hardware in this project is designed to read software from the special programmable RAM on the Nexys 3 board and then transmit it over the 3 Wire Bus to another board, containing the Single Chip Mote digital system.

Bootload Firmware for ARM Cortex-M0 This is a small program C program for the ARM Cortex-M0, meant to perform any actions to assist the bootloading process. The end of this program resets the system and the code in the instruction RAM is executed.

5.1 Reset Signals and Bootloading

As shown in section 3.11, the `PON` module samples an external reset signal and a reset request signal (`SYSRESETREQ`) from the ARM Cortex-M0, and generates two separate system-level resets for all other modules in the digital system. These are referred to as the hard reset and the soft reset. The hard reset is triggered when the external reset button is pressed (or upon power-up or bitstream loading). The soft reset is triggered by either the external reset button or the reset request signal

from the Cortex-M0. Almost all modules in the digital system use the soft reset. However, the AHBIMEM module uses both the hard and soft reset signals for different registers. The purpose and functionality of these two resets in the AHBIMEM module are explained in the following section.

5.2 Instruction ROM on the Single Chip Mote

The instruction ROM is a 16kB single-port ROM where instruction data is fetched after the system is initially powered-on (or in the case of an FPGA, when the bitstream is initially loaded). The AHBIMEM module contains a special register, called imem_mode, to select between the instruction RAM and ROM for instruction fetches. The imem_mode register has two resets, connected to the hard reset and the soft reset, and cannot be changed at any other time. The hard reset takes precedence over the soft reset, and sets imem_mode to ROM. Thus the instructions in the ROM always execute after power-on, bitstream loading, or external reset. On a soft reset, the imem_mode register is set to a value stored in the next_imem_mode register. next_imem_mode must be set by the ARM Cortex-M0 prior to requesting a reset. This register is set by writing to the BOOTLOADER_REG__CFG register. The purpose of the firmware, executing from the ROM, is to perform any necessary preparations during bootloading, set the next_imem_mode register to RAM, and then send a reset request. After a soft reset, instructions are fetched and executed from the RAM.

In an ASIC, ROM is considered to be hard-coded and cannot be changed after tapeout. However, on an FPGA, the value of this ROM is initialized using a COE file. See section 3.20.4 for more information on initializing the ROM.

5.3 Instruction RAM on the Single Chip Mote

The instruction RAM is a 64kB dual-port RAM in where the main software for the ARM Cortex-M0 is stored and fetched during regular operation of the Single Chip Mote. This RAM is not connected to either of the two reset signals, and thus is unaffected by a system reset.

The AHBIMEM module has two AHB slave interfaces, one only for reading instructions, and one for reading bootloading status and writing bootload configuration and data. The write port of this RAM is connected to a 3 Wire Bus interface and an AHB slave interface of the AHBIMEM for bootloading. The boot_mode register in the AHBIMEM module determines whether the 3 Wire Bus, the AHB, or neither write to the RAM. This register is set to neither upon reset, and can be changed by the ARM Cortex-M0 by writing to the BOOTLOADER_REG__CFG register. The read port of this RAM is connected to the AHB slave interface of the AHBIMEM for instruction data.

5.4 3 Wire Bus Interface

The 3 Wire Bus (3WB) is an interface used to serially send instruction data from the bootload hardware (on a separate FPGA board) to the Single Chip Mote (either on an FPGA or an ASIC). The three wires for this bus are clock, data, and latch. This interface operates according to the following rules:

- The data line is connected to a 31-bit shift register.
- Data is shifted onto the shift register on each rising edge of the clock.
- The latch signal is asserted just before the 32nd rising edge of the clock.
- The latch signal causes the 31 bits in the shift register, and the signal on the data line (for 32 bits total) to be written into the instruction RAM, as long as the `boot_mode` register is set to allow the 3 Wire Bus to write into the RAM.
- The `AHBIMEM` module keeps track of the RAM write address, and increments this address on every write.
- Once the instruction RAM has been filled (in this case 64kB of data has been written), the `AHBIMEM` module considers the boot finished and does not allow further writes from the 3 Wire Bus (unless the system is reset).

The `AHBIMEM` module implements the receiving end of this interface. The bootload hardware on the Nexys 3 implements the transmitting end of this interface.

5.5 Bootload Hardware on Nexys 3

The bootload hardware on the Digilent Nexys 3 board is used to copy the main software from a computer onto the RAM on the Nexys 3 board, and then send that data over the 3 Wire Bus interface. The ISE project file for this design is `scm-digital/proj/ise/spartan6/bootloader/BootloadHW.xise` and the source code is found in `scm-digital/src/hw/spartan6/bootloader/`.

This hardware contains two state machines and one 32kB FIFO. The first state machine reads 16 bits of data out of the Nexys 3 external RAM (referred to by Digilent as Cellular RAM) at 50MHz. The RAM is operated in a special burst mode, where one line of 128 16-bit words is read back-to-back at high frequencies. The state machine reads the data out of the RAM and writes it to the FIFO, pausing at the end of each 128-word line for two reasons: the first is to allow for a data refresh cycle in the RAM, and the second is to check and wait for the FIFO to have enough room for another 128 words of data. The second state machine reads 32 bits out of the FIFO at 5 MHz, and sends one bit at a time over the 3 Wire Bus interface.

The first state machine is activated by pressing the button on the Nexys 3 board labeled BTNU. Once 64kB of instruction data has been sent over the 3 Wire Bus interface, the LED on the Nexys 3 board labeled LED0 lights up. The unlabeled button in the center of the group of buttons resets the bootload hardware.

5.6 Bootload Firmware for ARM Cortex-M0

5.6.1 Firmware Essentials

The bootload firmware is a small program designed for facilitating the bootloading process using the ARM Cortex-M0 on the Single Chip Mote. The firmware can perform any function or use any peripheral just like the main software for the Single Chip Mote; however, it is limited to a size of 16kB and is permanently kept in

Field Bits	Field Name	Possible Values
0	imem_mode	0 = ROM, 1 = RAM
1	next_imem_mode	0 = ROM, 1 = RAM
3:2	boot_mode	00 = 01 = NONE, 10 = 3WB, 11 = AHB
4	boot_3wb_done	0 = 3WB boot not done, 1 = 3WB boot done
31:5	Unused	Unspecified

Figure 5.1: Register fields for the read-only BOOTLOADER_REG__STATUS register

Field Bits	Field Name	Possible Values
1:0	boot_mode	00 = 01 = NONE, 10 = 3WB, 11 = AHB
2	next_imem_mode	0 = ROM, 1 = RAM

Figure 5.2: Register fields for the write-only BOOTLOADER_REG__CFG register

ROM. Therefore it is essential that the firmware performs exactly what is needed, nothing more, and is absolutely correct. Regardless of how the new instruction data is received and loaded into the ROM, the firmware must perform two important functions in order to ensure that the main software is executed. The first is that the instruction memory is set to boot from RAM after a soft reset. The second, is to send a reset request after the bootloading has completed.

5.6.2 Application Interrupt and Reset Control Register

The Application Interrupt and Reset Control Register (AIRCR) is a status and reset control register for the ARM Cortex-M0. See ARM documentation for more details on this register. Setting the second bit of this register, the SYSRESETREQ bit, indicates a request for a system-level reset. This asserts the SYSRESETREQ signal in the hardware, and the PON module generates the soft reset in response.

5.6.3 AHB Slave Interface for Bootloading

The AHBIMEM module has a special AHB slave interface for bootloading. This interface is accessed using addresses with the prefix 0x01 (this includes any address in the range of 0x01000000 to 0x01FFFFFF). Any AHB reads to an address with the prefix 0x01 returns the read-only BOOTLOADER_REG__STATUS register contents. This register shows the current imem_mode, next_imem_mode, and boot_mode values. It also shows whether bootloading through the 3 Wire Bus has finished (as in 64kB of data has been written through the 3 Wire Bus). Any AHB write to an address in the range of 0x01000000 - 0x0100FFFF will result in a write into the instruction RAM at the corresponding address, with the prefix of 0x01 replaced with 0x00. Any AHB write to the address 0x01F00000 will result in a write to the write-only BOOTLOADER_REG__CFG register. Writing to this register sets next_imem_mode and boot_mode. The table in Figure 5.1 describes the bit fields in the read-only BOOTLOADER_REG__STATUS register, and the table in Figure 5.2 describes the bit fields in the write-only BOOTLOADER_REG__CFG register.

5.6.4 Bootloading with the 3 Wire Bus

All firmware loading software via the 3 Wire Bus must follow the same basic procedure:

1. Set the `boot_mode` register to 3WB
2. Poll the `BOOTLOADER_REG__STATUS` register until `boot_3wb_done` is 1
3. Set the `boot_mode` register to NONE
4. Set the `next_imem_mode` register to RAM
5. Set the `SYSRESETREQ` bit of the AIRCR to trigger a soft reset

5.6.5 Bootloading with the AHB Slave Interface

The AHB slave interface for bootloading is typically used in situations where software data is sent over another interface accessible by the ARM Cortex-M0, such as the radio or UART. In future iterations of the Single Chip Mote might also include an optical interface to send software data to multiple devices at once. All firmware used to load software via the AHB slave interface must follow the same basic procedure:

1. Set the `boot_mode` register to AHB
2. Listen to the radio/UART/optical interface that is sending the instruction data
3. Copy the instruction data one word at a time into the instruction ROM by writing to addresses with the `0x01` prefix
4. Set the `boot_mode` register to NONE
5. Set the `next_imem_mode` register to RAM
6. Set the `SYSRESETREQ` bit of the AIRCR to trigger a soft reset

5.6.6 Current Firmware Implementation

The current implementation of the firmware is found in `scm-digital/proj/keil/firmware/bootloader.uvproj`. It follows the same process described in section 5.6.4. The bootloader first prints a message over UART saying "Welcome to the Bootloader. Setting boot mode to 3WB." The `boot_mode` register is changed to 3WB, and then the `BOOTLOADER_REG__STATUS` register is repeatedly polled until `boot_3wb_done` is 1. The `boot_mode` is changed to NONE, and the `next_imem_mode` register is set to RAM. At that point another message is printed over UART says "Boot complete. Restarting...". There is a long, empty for loop in order to allow for the entire message to print, and then the `SYSRESETREQ` bit of the AIRCR is set.

5.7 Loading Software Using the Bootloader

Loading software onto the Single Chip Mote requires three files:

- The bitstream file for the Single Chip Mote digital system, `ucontroller.bit`. For the Nexys 4 DDR using the Artix-7, this bitstream is generated using the ISE project found at `scm-digital/proj/ise/artix7/SingleChipMote/SingleChipMote.xise`. For the Nexys 3 using the Spartan-6, this bitstream is generated using the ISE project file found at `scm-digital/proj/ise/spartan6/uRobotDigitalController/uRobotDigitalController.xise`. For more information on how to generate a bitstream file, see section 2.4.1.
- The bitstream file for the bootloading hardware, `top.bit`. This bitstream is generated using the ISE project found at `scm-digital/proj/ise/spartan6/bootloader/BootloadHW.xise`. For more information on how to generate a bitstream file, see section 2.4.1.
- The C binary file containing the software to be loaded, code.bin. This is compiled using the Keil uVision5 project found at `scm-digital/proj/keil/uRobotDigitalController/code.uvprojx`. For more information on how to build and compile the software, see section 2.5.1.

Once these three files have been generated, the two FPGA boards must be connected to one another as well as connected to the computer via the micro-USB ports on the boards. It is also recommended that a serial terminal is used to read the UART output of the Single Chip Mote to verify that the firmware was loaded successfully.

5.7.1 Connecting UART

Only the FPGA board containing the Single Chip Mote digital system needs to be connected via UART. The bootload hardware does not use UART.

For the Nexys 4 DDR, connect the micro-USB port labeled PROG UART to the computer and move the power switch to the ON position. This USB port is used for both programming and UART communication. Once the board has been recognized by the computer and the proper drivers have been installed, use Device Manager to find the COM port associated with the Nexys 4 board. From here, open up any serial terminal program, and connect to the COM port associated with the Nexys 4 board using the following settings: baud rate of 19200, with 8 data bits, 1 stop bits, no parity bits, and no flow control. Load the bitstream file using the instructions in section 2.4.1. Once the Single Chip Mote bitstream file has been loaded onto the board, there will be a message sent over UART from the bootloader reading "Welcome to the Bootloader."

For the Nexys 3, connect the micro-USB port labeled UART to the computer and move the power switch to the ON position. This USB port is used only for UART and not for programming. Once the board has been recognized by the computer and the proper drivers have been installed, use Device Manager to find the COM port associated with the Nexys 3 board. From here, open up any serial terminal program, and connect to the COM port associated with the Nexys 3 board using the following settings: baud rate of 19200, with 8 data bits, 1 stop bits, no parity

bits, and no flow control. Load the bitstream file using the instructions in section 2.4.1. Once the Single Chip Mote bitstream file has been loaded onto the board, there will be a message sent over UART from the bootloader reading "Welcome to the Bootloader."

5.7.2 Using Nexys 3 to Load Nexys 4 DDR

The following steps show how to load the Single Chip Mote hardware and software onto a Nexys 4 DDR board:

1. The Nexys 3 and Nexys 4 DDR boards each use three ports on one of the Pmod connectors for the 3 Wire Bus. While the boards are powered off, connect port JB1 on the Nexys 4 to port JB1 on the Nexys 3, for the data wire. Connect port JB2 on the Nexys 4 to port JB7 on the Nexys 3, for the latch wire. Connect port JB10 on the Nexys 4 to port JB10 on the Nexys 3, for clock wire. Also connect the ground ports (JB5 or JB11) of the two boards together. See Figure 5.3 for an image of this setup.

2. On the Nexys 4 DDR, connect the micro-USB port labeled PROG UART to the computer.

3. On the Nexys 3, connect the micro-USB port labeled USB PROG to the computer.

4. Switch on both of the boards. Ensure that both boards are recognized and that all drivers are installed.

5. Open a serial terminal program and connect to the COM port of the Nexys 4 DDR using the instructions in section 5.7.1.

6. Open Digilent Adept, and load the C binary file, code.bin, onto the external RAM of the Nexys 3. For more instructions on this process see section 2.4.2.

7. Use Digilent Adept to load the bootload hardware bitstream file, top.bit, onto the Nexys 3. For more instructions on this process, see section 2.4.2.

8. Open iMPACT, and load the bitstream file for the Single Chip Mote digital system onto the Nexys 4. For more instructions on this process, see section 2.4.1.

9. From here a message will be sent over UART, reading "Welcome to the Bootloader. Setting boot mode to 3WB." The Single Chip Mote is now waiting for the software data to be sent over the 3 Wire Bus.

10. Press the button labeled BTNU on the Nexys 3 board. The LED labeled LED0 will light up when all of the data has been sent. Another message will be sent over UART, reading "Boot complete. Restarting..."

11. After a slight pause, the main software will load. This is indicated by a message sent over UART reading "Welcome to the uRobot Digital Controller. Initialization Complete."

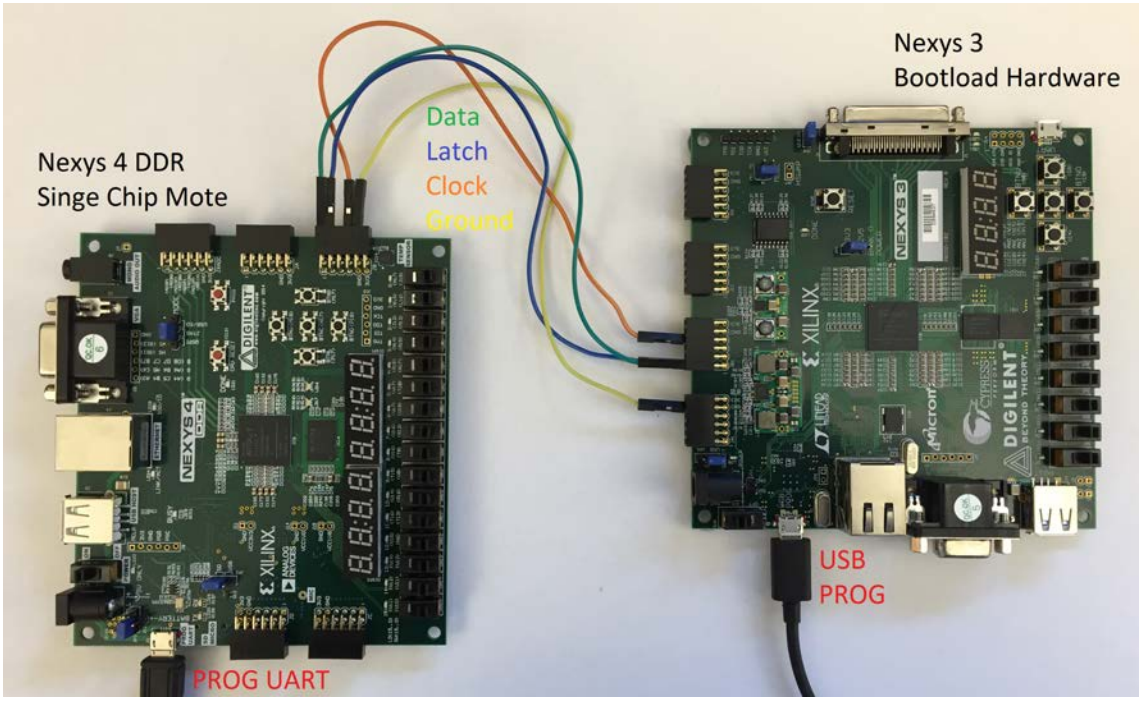


Figure 5.3: Physical connections to load the Nexys 4 DDR with the Nexys 3

5.7.3 Using Nexys 3 to Load Nexys 3

Older versions of the Single Chip Mote digital system can also be loaded onto a Digilent Nexys 3 board instead of the Nexys 4 DDR. In this case, two Nexys 3 boards are required, one now referred to as the SCMboard and one now referred to as the bootboard. The SCMboard also needs a special expansion board from Digilent called the VmodMIB [32], as the 3 Wire Bus clock input is attached to one of the Pmod connectors on this expansion board.

To load the Single Chip Mote hardware and software on the SCMboard:

1. The SCMboard and bootboard each use three ports on one of the Pmod connectors for the 3 Wire Bus. While the boards are powered off, connect port JB1 on the SCMboard to port JB1 on the bootboard, for the data wire. Connect port JB2 on the SCMboard to port JB7 on the bootboard, for the latch wire.

2. Attach the VmodMIB expansion board to the SCMboard. Connect port JB10 on the VmodMIB to port JB10 on the bootboard, for clock wire. Also connect any of the ground ports of the two Nexys 3 boards together. See Figure 5.4 for an image of this setup.

3. On the SCMboard, connect the micro-USB port labeled USB PROG to the computer. Also connect the micro-USB port labeled UART to the computer.

4. On the bootboard, connect the micro-USB port labeled USB PROG to the computer.

5. Switch on both of the boards. Ensure that both boards are recognized and that all drivers are installed.

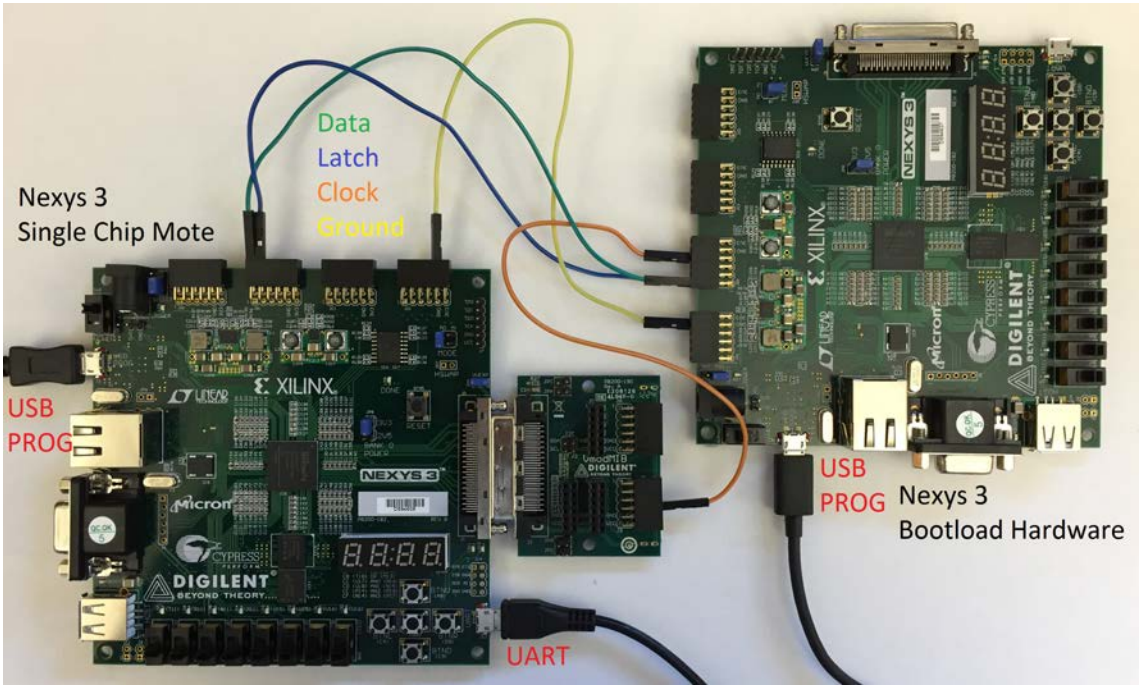


Figure 5.4: Physical connections to load the Nexys 3 with another Nexys 3

6. Open a serial terminal program and connect to the COM port of the SCMboard using the instructions in section 5.7.1.

7. Open Digilent Adept, and load the C binary file, code.bin, onto the external RAM of the bootboard. For more instructions on this process see section 2.4.2.

8. Use Digilent Adept to load the bootload hardware bitstream file, top.bit, onto the bootboard. For more instructions on this process, see section 2.4.2.

9. Use Digilent Adept load the bitstream file for the Single Chip Mote digital system onto the SCMboard. For more instructions on this process, see section 2.4.2.

10. From here a message will be sent over UART, reading "Welcome to the Bootloader. Setting boot mode to 3WB." The Single Chip Mote is now waiting for the software data to be sent over the 3 Wire Bus.

11. Press the button labeled BTNU on the bootboard board. The LED labeled LED0 will light up when all of the data has been sent. Another message will be sent over UART, reading "Boot complete. Restarting..."

12. After a slight pause, the main software will load. This is indicated by a message sent over UART reading "Welcome to the uRobot Digital Controller. Initialization Complete."

5.8 Connecting Two FPGA Boards for Simulated Packet Transmission

The Single Chip Mote on both the Nexys 3 and Nexys 4 DDR do not have actual radios connected to the FPGAs. However, there are four GPIO pins on each board dedicated to simulating packet transmission, by connecting the transmitted data output (and its clock) of one board to the received data input (and its clock) of another, and vice versa. Note that the Nexys 4 DDR design uses the same pin for the received data clock and the 3 Wire Bus clock, and therefore the Nexys 4 DDR will need to be disconnected from the bootloading board.

The first step is to program each board with the Single Chip Mote hardware and software using the bootloader. Both boards should be connected to the computer via UART if commands to send and receive packets are given from the computer using UART. If the boards are programmed using the 3 Wire Bus, then the associated wires must be disconnected after programming.

The next step is to connect the ground pins of both boards to each other. On the Nexys 4 DDR, port JB11 can be used and on the Nexys 3, port JC5 can be used.

The next steps are to connect the `tx_clk` pin of one board to the `rx_clk` pin of the other board, and vice versa. Then connect the `tx_dout` pin of one board to the `rx_din` pin of the other board, and vice versa. On the Nexys 4 DDR, the `tx_clk` pin is found on port JB7, the `tx_dout` pin is found on port JB8, the `rx_din` pin is found on port JB9, and the `rx_clk` pin is found on port JB10. On the Nexys 3, the `rx_clk` pin is found on port JC1, the `rx_din` pin is found on port JC2, the `tx_dout` pin is found on port JC3, and the `tx_clk` pin is found on port JC4. Figure 5.5 shows two Nexys 4 DDR boards connected together.

Once the two boards are connected together, packets can be "transmitted" between the two boards without using a radio. This is done to verify that the radio controller hardware and software is working correctly.

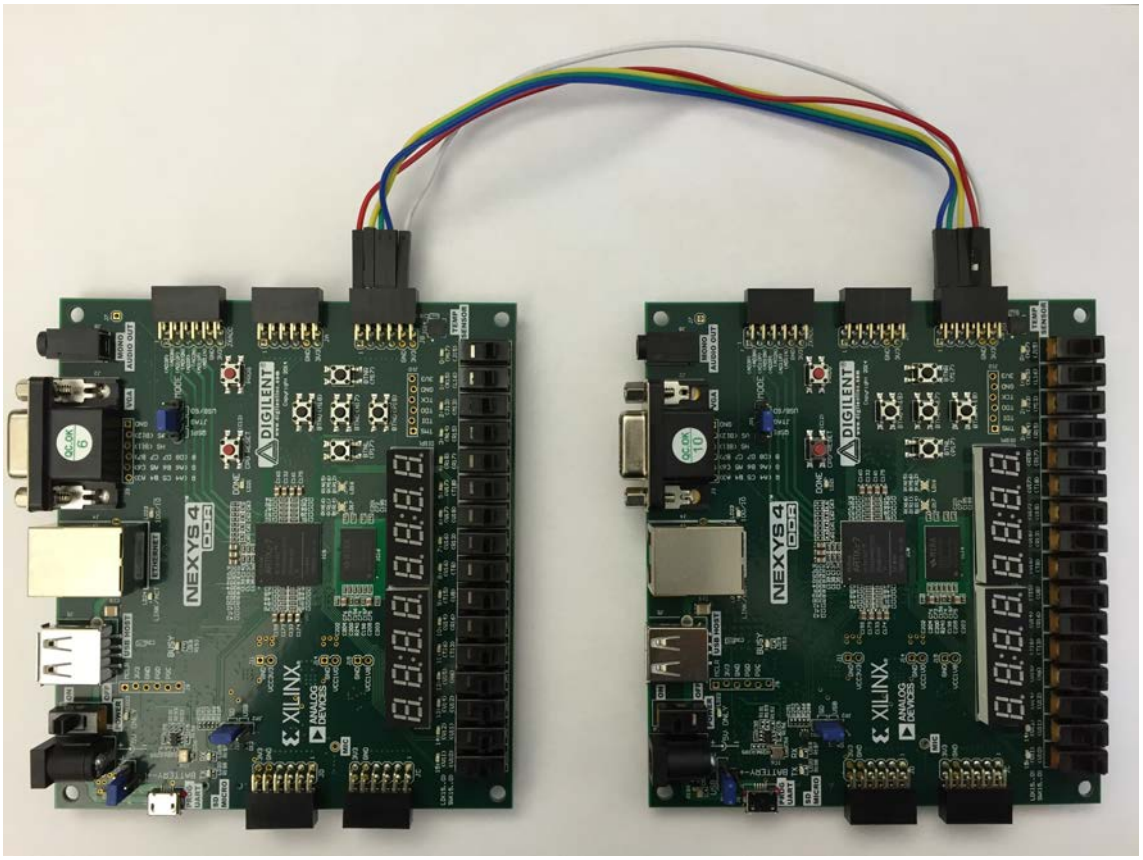


Figure 5.5: Connecting two Nexys 4 DDR boards for simulated packet transmission without a radio

Chapter 6

Testing

The chapter outlines the hardware testing procedures for the Single Chip Mote digital system. Unit-level testbenches and simulation are used to verify that modules or small groups of modules behave properly in isolation; real-time tests on the FPGA hardware indicate that the entire system works together as expected, including interfaces with the analog and radio circuits. Mistakes in the final ASIC version of this chip are extremely costly, and simulating the entire Single Chip Mote in Cadence or Synopsys is computationally intensive and slow. Therefore, it is essential that the Single Chip Mote digital system is thoroughly tested in simulation and on an FPGA before moving on to an ASIC design.

6.1 Simulation Testing Using ISim

New Verilog modules and major changes to existing modules should be verified in simulation to catch any bugs before integrating the module into the Single Chip Mote digital system. While it is difficult to simulate the entire Single Chip Mote digital system altogether, designers typically use unit-level tests to check that modules are functioning correctly and low-level integration tests to ensure that combinations modules work together and interface as expected. The module or modules being tested are referred to as the device under test (DUT) or unit under test (UUT). Xilinx ISE uses a Verilog simulator, called ISim, to simulate testbenches added to an ISE project.

6.1.1 Original Testbenches for Spartan 6

The Single Chip Mote digital system was originally designed by Francesco Bigazzi, a visiting scholar, for the Spartan 6 FPGA. Bigazzi created separate ISE projects for each testbench, with separate folders to hold the testbench and DUT Verilog code. Note that Bigazzi used a copy of the DUT Verilog code for his testbenches, rather than linking to the original files that are used for the implementation. Thus any corrections made while testing needed to be copied back to the original file. Each ISE project and its accompanying code are saved in their own folder in `scm-digital/proj/ise/spartan6/testbench/TESTs`. For example, the `scm-digital/proj/ise/spartan6/testbench/TESTs/ADC` folder contains the `ADC.xise` project file for testing Bigazzi's original ADC controller implementation. The `scm-digital/proj/ise/spartan6/testbench/TESTs/ADC/src` folder

contains a copy of Bigazzi's original ADC code and any additional testbench code.

Many of the testbenches in the `scm-digital/proj/ise/spartan6/testbench/TESTs` folder are outdated since modules have been deprecated and are no longer in use (such as the original ADC controller and the original AHB arbiter). Also, there is no coherency between the code in `scm-digital/src/hw/spartan6` and the code stored for each individual test. Modules have been updated during the course of the project while their testbenches (and the copies of the DUT modules) have not. For example, the `APBMUX` module has been modified to add and remove APB peripherials, and the testbench designed by Bigazzi is not compatible. The `Rfcontroller` module went through a large revision to add the spreader and correlator/despreader module, to add an interface to the radio timer, and to improve the memory-mapped register interface. Therefore the original testbench, which verified the TX and RX state machines before these changes, is now incompatible. This does not mean that these changes went untested, as individual testbenches were created to verify the spreader, correlator/despreader, and radio timer. The integration of these modules was verified in real-time on the FPGA rather than through testbenches. However, this is not the best practice nor is it an accurate reflection of the testing procedure used in industry, and further care must be taken in the future to ensure that all non-trivial changes are verified.

6.1.2 Artix 7 Testbenches and Improved Testing Procedure

With the transition from the Spartan 6 to Artix 7 FPGA, and the accompanying changes to the git repo and file organization, a new testing procedure is devised to ensure greater coherency between the module code and their testbenches.

There is a single ISE project, `scm-digital/proj/ise/artix7/testbench/testbench.xise`, for all new testbenches. The code for the testbenches themselves are found in the `scm-digital/src/hw/artix7/uRobotDigitalController/testbench` folder, while the code for the device under test is the same as the implementation code, found in `scm-digital/src/hw/artix7/uRobotDigitalController`. Therefore, any changes or corrections made during simulation testing are applied to the actual code for the module, rather than a copy of that module's code. Any changes or corrections made while testing with the FPGA in real-time should be verified by running the testbenches again (without the need for copying since the same code is used for simulation and implementation). Any major revisions require that the testbench code is updated alongside with the DUT code, and that the new tests pass in simulation before verifying in real-time.

The ISE project for testbenches, `scm-digital/proj/ise/artix7/testbench/testbench.xise`, currently has 5 testbenches:

RFTIMER_tb Tests the radio timer module `RFTIMER`. The compare and capture units are configured and then the timer is enabled. Stimulus is applied to the DUT and the testbench ensures that it behaves as expected.

SFD_delay_TB Tests how long (in time) it takes to send the last bit of the SFD of a packet after telling the radio controller to send a packet (using `TX_SEND`). Also test how long (in time) it takes to send the last bit of a packet after telling the radio controller to send a packet. This testbench use a clock frequency of 2MHz, to match the implementation. Example packet data is loaded into the

`tx_fifo2` module connected to the `spreader` module. The `spreader` module is then activated using the `tx_start` input and the testbench runs until the `tx_sfd_sent` and `done` outputs are asserted. The time between `tx_start` and `tx_sfd_sent` is calculated along with the time between `tx_start` and `done`.

LOAD_delay_TB Tests how long (in time) it takes in the worst-case to copy packet data into the TX FIFO for radio transmission. This is the time between when the `TX_LOAD` signal activates the radio controller, and when the radio controller indicates it is done using the `TX_LOAD_DONE` interrupt. This testbench uses a clock frequency of 5MHz, to match the implementation. The `RFcontroller`, `AHBDMEM`, `DMA_V2`, `AHBLiteArbiter_V2`, and AHBsub bus modules are required for this experiment. In order to replicate the worst-case scenario, the largest possible packet (127 bytes) is loaded into the TX FIFO while the Master 0 input of the arbiter (which is used for the Cortex-M0) is always requesting access to the AHBsub bus. This way the DMA must wait each time it tries to copy data from the data memory to the radio controller.

correlator_TB Tests the `corr_despreader` and `correlator` modules to ensure that they can detect and return a packet using an input data stream from an MSK demodulator. An example data stream of MSK chips is clocked into the `corr_despreader` module, which converts it to packet data on the `dout` output. The output is then compared with the actual packet data that corresponds to the input MSK chips.

spreader_TB Tests the combination of the `tx_fifo2`, `spreader`, `symbol2chips`, `correlator`, `corr_despreader`, and `rx_fifo` modules. Regular packet data is loaded into the `tx_fifo2` module. The `spreader` module reads it out and converts it to a serial data stream for an MSK modulator (using the `symbol2chips` module). The `corr_despreader` module reads that data stream (assuming it came from an MSK demodulator) and converts it back to regular packet data (using the `correlator` module) which is then stored into the `rx_fifo` module. This simulates sending and receiving a packet, and if the data in the two FIFOs match then the modules are operating correctly.

As the Single Chip Mote digital system is expanded and modified, more testbenches should be added to the ISE project and all changes should be verified in simulation.

6.1.3 Using ISim

To simulate a testbench using ISim, first open an ISE project containing a testbench, such as `scm-digital/proj/ise/artix7/testbench/testbench.xise`. In the Design Hierarchy panel, select the Simulation view (as shown in Figure 6.1) to see a list of all the testbenches in the project. Selecting one of these testbenches brings up the Simulate Behavioral Model process in the Process panel. Running this process compiles the testbench and Verilog code into an executable and launches ISim to run that executable. Before simulating, it is recommended that the settings for the Simulate Behavioral Model process are modified to match the current testbench. Right-clicking the Simulate Behavioral Model and selecting Process Properties... (as shown in Figure 6.1) brings up the ISim Properties window (Figure 6.2).

The first property to modify is the Simulation Run Time. The default run time (when the Run for Specified Time box is unchecked) is 1000ns. For large testbenches, the default time is too short and the simulation pauses in the middle of the test (the console can be used in ISim to manually direct the simulation to continue). Therefore the Simulation Run Time should be set to some time larger than the total run time of the testbench. This can be estimated by taking the time unit value (specified using the `` `timescale `` directive at the top of every testbench) and multiplying it by the number of time steps executed. For example, the following code sets the time unit to 1ns: `` `timescale 1ns / 1ps ``, and the following line executes 12 time steps: `#12;`. The Simulation Run Time parameter should not be too large in order to avoid simulating longer than needed. Another alternative is to make the Simulation Run Time parameter much larger than necessary, and then adding `$finish;` to the end of the testbench to stop execution exactly when the test completes.

The second property to modify is the Custom Waveform Configuration File. A waveform configuration file is used by ISim to display signals from the testbench in the waveform window. When no waveform configuration file is specified, ISim by default displays the waveforms of the top-level signals in the testbench. Waveforms for signals inside of the DUT and other instantiated modules can be added to the waveform window; however, the simulation must be run again in ISim (using the `restart` and `run` commands in the console) in order for the waveforms to display properly. If ISim is closed and re-opened, such as when the Verilog code is modified and re-compiled, all of the non-default waveforms must be added again. When a waveform configuration file is specified, ISim will automatically add all of the waveforms before executing the simulation. Therefore, the recommended approach is to run a testbench without a waveform configuration file, add all of the signals of interest to the waveform window, save the waveform configuration, and then modify the Custom Waveform Configuration File property. When using ISim, more signals can be added to the waveform view and saved to the same waveform configuration file; the new signals will also be added automatically the next time ISim is opened. Each testbench should have its own waveform configuration file, and the Custom Waveform Configuration File property must be changed when switching between testbenches.

Figure 6.3-6.6 demonstrate how to add signals to the waveform window, restart and run the simulation using the console, and save a wave configuration file to be used for later simulations. Figure 6.3 shows the ISim window after launching a simulation using the default wave configuration file. The console shows all `$display` and other varieties of print statements. The console also shows that the simulation stopped after 110100ns; this is due to the `$finish;` line added to the end of the testbench code. Otherwise, the simulation would have continued to run until it reached the value specified by the Simulation Run Time parameter. The waveform window by default shows the waveforms of all the top-level signals. Most of these signals are not necessary and can be removed. The order of these signals can also be re-arranged, and the radix used to display the values of each bus can be changed.

To add more signals to the waveform window, first click on the module or process that contains that signal in the Instance and Process panel on the left side of the ISim window. This updates the Objects panel to list all of the signals inside of that module or process. Right-clicking a particular signal and selecting Add To Wave Window (or dragging the signal into the window) adds the waveform. These steps

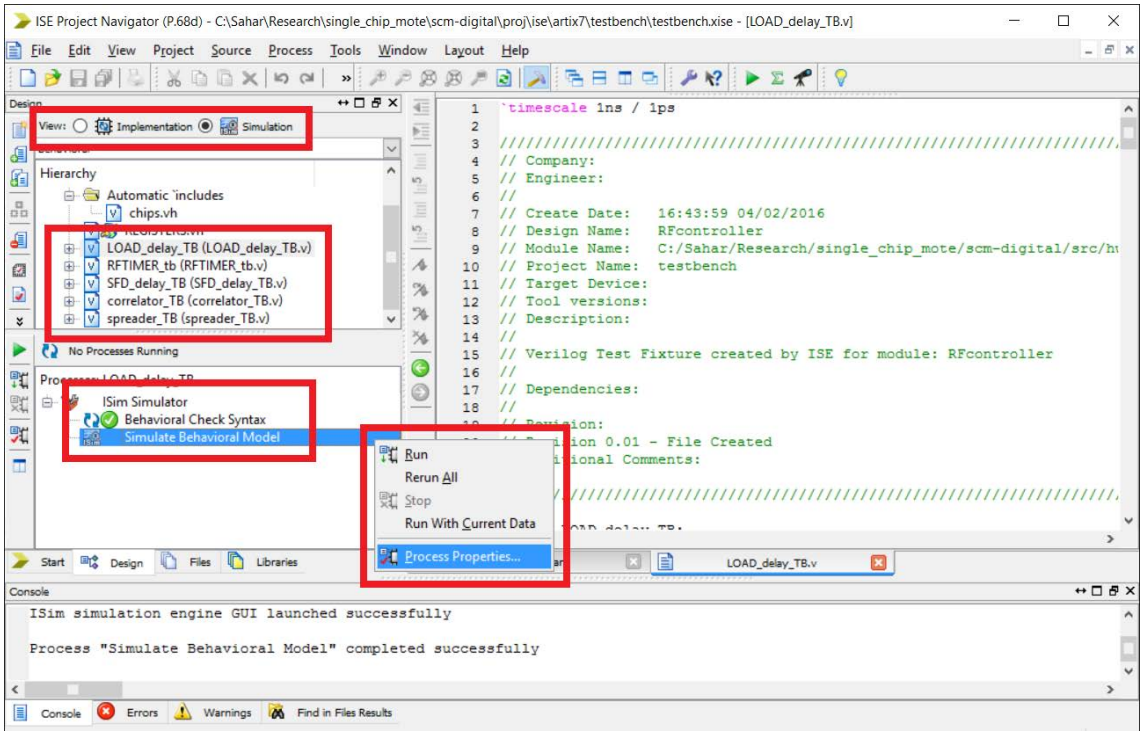


Figure 6.1: Running a simulation in ISE

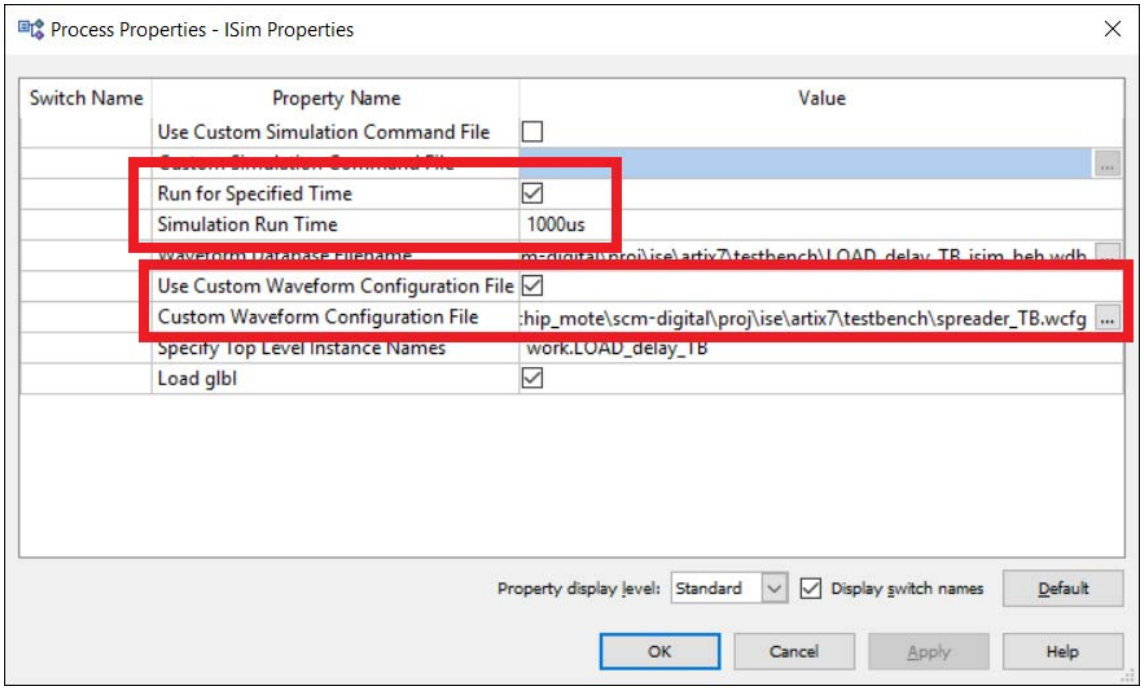


Figure 6.2: ISim Properties window

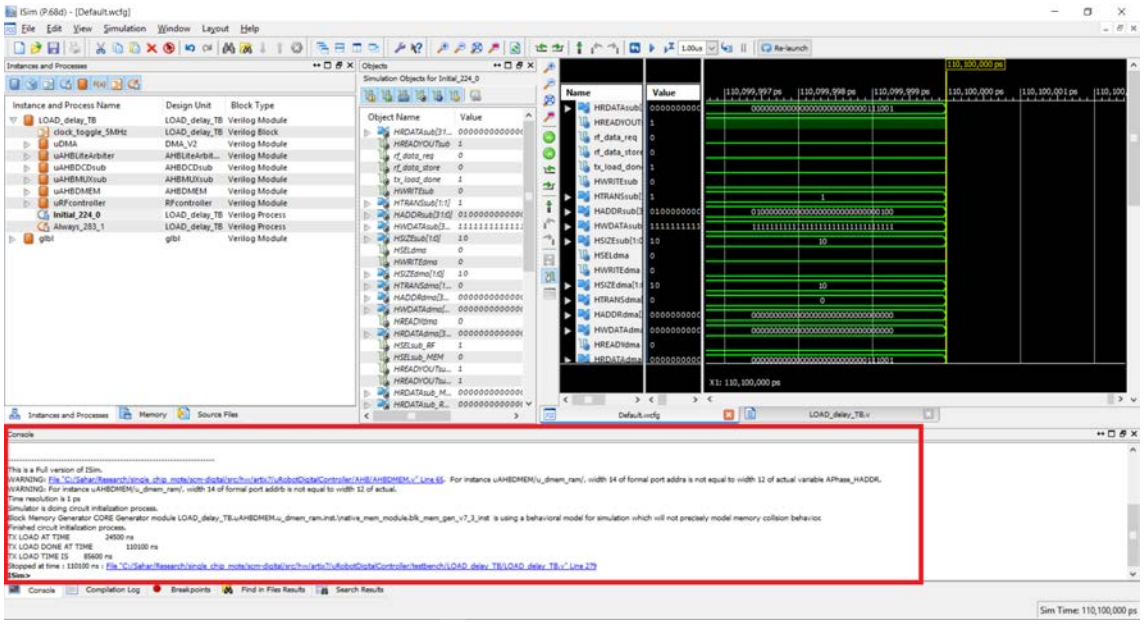


Figure 6.3: ISim window after launching a simulation using the default wave configuration file

are shown in Figure 6.4.

Once the waveform window is updated with the required signals, the console can be used to restart and run the simulation again to plot all of the waveforms. This is show in Figure 6.5. When a simulation is paused (for example at time 100ns), more signals can be added to the waveform window; however, their waveforms for all time steps before 100ns will not be displayed. If the simulation is continued after that point (by using the `run` command in the console), the waveform for all time steps after 100ns will be displayed.

Figure 6.6 shows how to save the waveform configuration file using the Save As... option in the File menu.

6.2 Real-Time Testing on FPGA

Testbenches are useful when determining that a module behaves as according to its specification in an ideal environment. The main limitation with testbenches is that they fail to uncover bugs in edge cases that are not considered or expected by the designer. Running real-time integration tests reveal problems where modules interact with one another and assumptions about interfaces break down. Test software can also indicate situations where the module fails to satisfy the requirements for the application, or the hardware does not behave according to what the software developer expects. Modules that interact with circuits and signals from outside the digital system, such as the ADC or the radio, should also be verified in real-time, in order to ensure that the digital and analog circuits interact appropriately and that the digital module can handle any non-idealities that inevitably come from a noisy system.

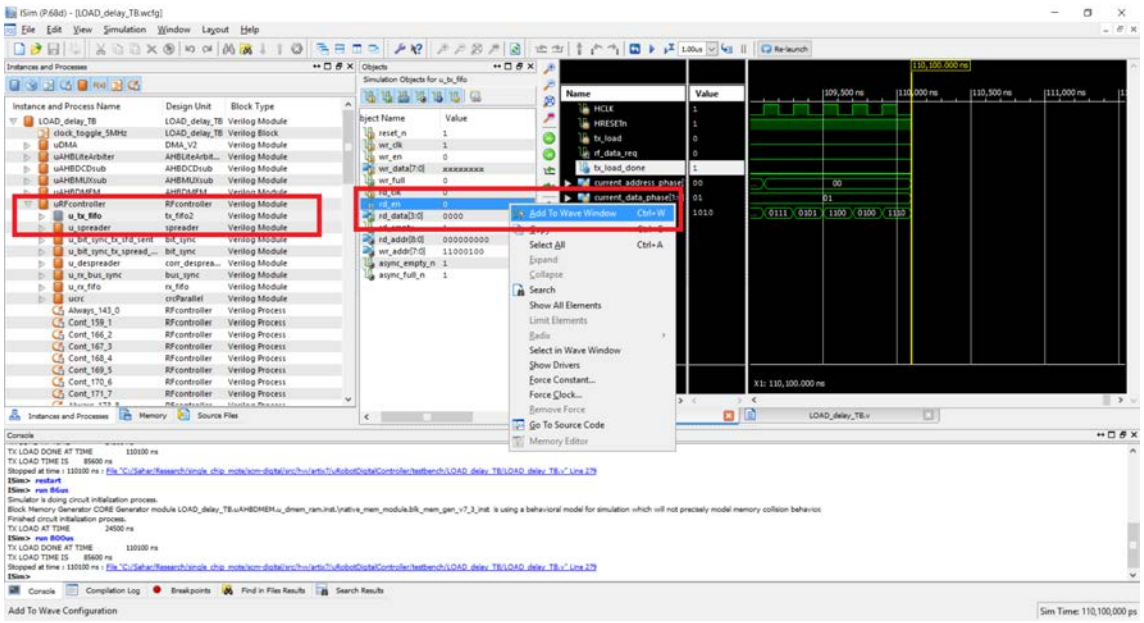


Figure 6.4: Adding a signal to the waveform window in ISim

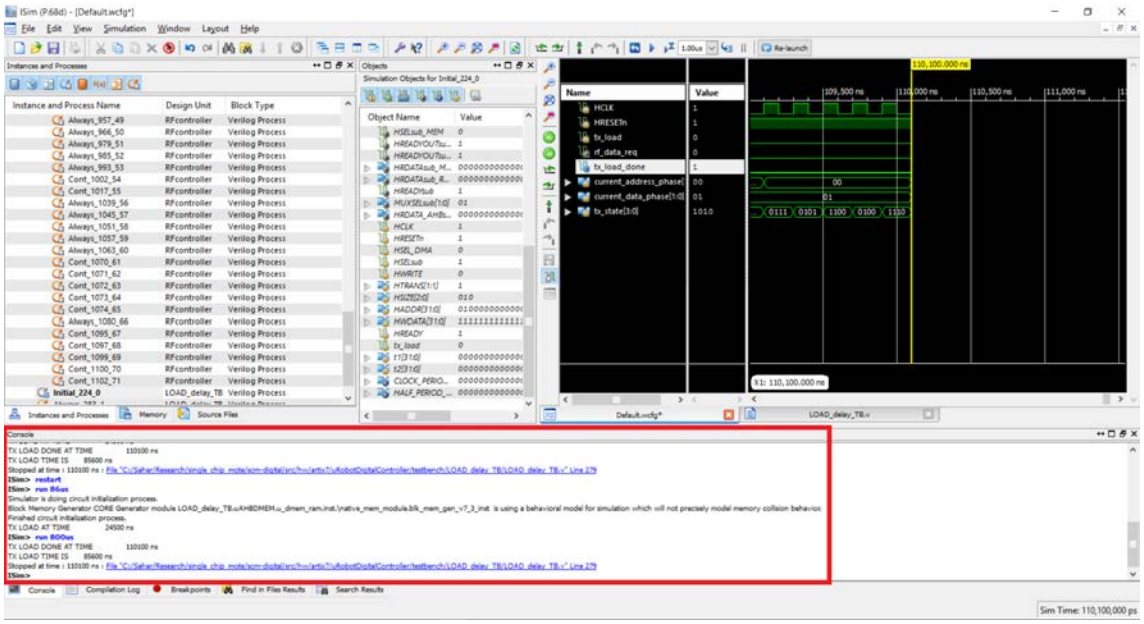


Figure 6.5: Restarting and running a simulation in ISim

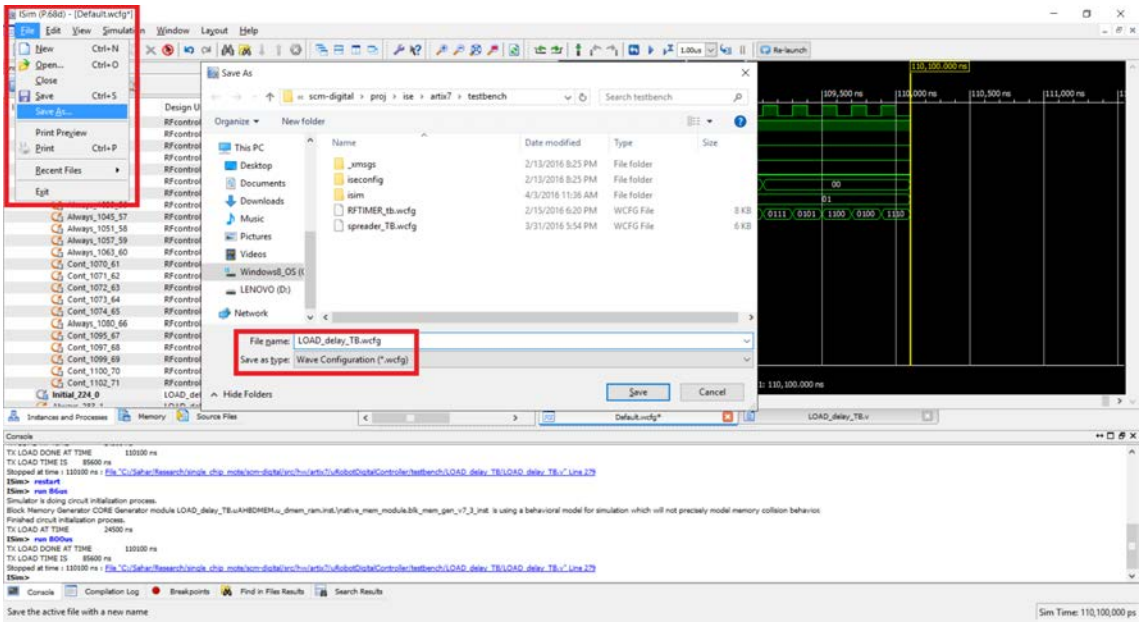


Figure 6.6: Saving a wave configuration file in ISim

6.2.1 Test Programs

Testing the Single Chip Mote digital system on an FPGA will require a C program compiled for the Cortex-M0. Ideally this program would also contain code to exercise all of the features that require testing, including code to check if the results are as expected.

The current demo code found in `scm-digital/proj/keil/uRobotDigitalController/code.uvprojx` is not an exhaustive test suite; however, it does exercise many of the features of the radio controller and radio timer, and is typically used to verify that minor changes to those modules have not caused them to stop working outright. This code is also not the best test code since it requires user input via UART.

In the future, an autonomous test suite should be developed to provide stimulus to each Single Chip Mote peripheral, and check if the outputs match expectation, and then either send the results over UART or toggle the general-purpose outputs to indicate when the test finishes and if it is a success or failure. This test suite would then be run after hardware changes and hardware testbenches.

6.2.2 ChipScope

Unexpected behavior encountered during FPGA testing is much more difficult to diagnose than in simulation. Simulations allow the designer to observe and sometimes manipulate signals and state within a design and work quickly and iteratively to resolve the issue. Within an FPGA, these signals and states are inaccessible, and it is not possible to 'pause' a circuit in order to examine its state. Xilinx does provide a tool called ChipScope, which is used to probe and measure internal FPGA signals and registers in real-time. A designer can choose which signals to probe after running the Synthesis process, and before running the Translate process. Any changes to the probed signals will require running the Translate, Map, and Place & Route

processes again. For more information on how to use ChipScope, see the following tutorial from Xilinx: Using Xilinx ChipScope Pro ILA Core with Project Navigator to Debug FPGA Applications [16]. A copy is found in `scm-digital/doc`.

The recommended debugging process using ChipScope is as follows:

1. Determine which module may be causing the erroneous behavior.
2. Make an educated guess about which signals and state in that module may reveal the source of the erroneous behavior, and connect those signals to the ChipScope ILA unit.
3. Run through the processes to generate a bitstream for the design, and load the bitstream using the ChipScope Pro software.
4. Set up the trigger settings to capture the erroneous behavior.
5. If there is no trigger or the captured data does not reveal the erroneous behavior, modify the trigger settings or add more signals to the ILA and try again.
6. Once the problem is detected, use the captured data to determine the possible bug in the Verilog code that would lead to the erroneous behavior. It also may be possible use the captured data to replicate the situation in a testbench for further debugging.
7. Update the code with a possible fix.
8. Test the new code in real-time or in simulation and determine if the problem is solved. It may be necessary to repeat the previous step multiple times before a solution is found.

Note that ChipScope ILA units use up a considerable amount of block RAM resources on the FPGA, and it is possible to create an ILA unit that requires more RAM than available due to the number of signals that are sampled. In this case the Map process will fail and the ILA unit must be modified to sample fewer signals.

Chapter 7

Transitioning to ASIC

This chapter provides an outline of the code changes required to create an ASIC version of the Single Chip Mote digital system using the Verilog code written for the Artix-7 FPGA. Most of the code is applicable to both FPGA and ASIC designs; however, the FPGA design requires primitives for clock division and ROM/RAM instantiation which are not available for ASIC designs. Most ASIC designs also implement scan chain and test logic to ensure that there are no manufacturing defects. Once the code changes are complete, the Verilog is synthesized using Synopsys Design Compiler, and the final layout is created using Synopsys IC Compiler. The layout is then combined with the required analog and RF circuits and sent for manufacturing.

Some of the changes described in this chapter are already implemented and added to a special branch in the git repo. These modifications were used to generate the results reported in Chapter 1. The `asic-src` branch contains several new folders:

- `scm-digital/src/hw/scm_v2` for the updated Verilog source code
- `scm-digital/proj/modelsim` for the ModelSim project containing system-level testbenches for the updated code
- `scm-digital/proj/keil/asic_testbench_fw` and `scm-digital/src/sw/asic_testbench_fw` for the C code written into the ROM for the system-level testbenches
- `scm-digital/proj/keil/asic_testbench_sw` and `scm-digital/src/sw/asic_testbench_sw` for the C code written into the RAM for the system level testbenches

The Synopsys toolchain and ModelSim simulator are installed and run on servers managed by the Berkeley Wireless Research Center (BWRC).

7.1 Power-On Reset and Clock Generator

The Single Chip Mote digital system implemented on the Artix-7 FPGA uses the `PON` module to generate the required clock and reset signals. This module uses FPGA primitives not available on ASIC designs and counter-based clock dividers which are relatively inaccurate. The ASIC version of the Single Chip Mote requires

a separate analog circuit, with inputs and outputs to/from the digital system to handle clocks and resets. This analog circuit is responsible for:

- Generating the 5MHz system clock
- Generating the 2MHz radio transmission clock (with the option of a clock enable signal)
- Generating the 500kHz radio timer clock (with the option of a clock enable signal), and ensuring that this clock is phase-aligned with the system clock
- Generating a 2MHz radio receive clock (with the option of a clock enable signal) using the input data from the radio transceiver
- Buffering the 3 Wire Bus clock from a pad on the chip
- Debouncing the input reset signal from a pad on the chip
- Sampling the reset request signal from the Cortex-M0
- Generating the two reset signals using the input reset and the reset request signal

All of the inputs and outputs to the `PON` module must be moved to the top module for the digital system.

7.2 Memories

The Single Chip Mote digital system implemented on the Artix-7 FPGA uses instantiated memories created in COREGenerator for the instruction ROM, instruction RAM, and data RAM. These memories must be replaced with memories generated by the appropriate memory compiler, and this process requires changes to the `AHBIMEM` and `AHBDMEM` modules. The radio controller also has two FIFOs to store TX and RX packet data, and the memories for those FIFOs must be implemented using two-port register files generated by the appropriate memory compiler. This requires changes to the `tx_fifo_mem` and `rx_fifo_mem` modules.

The memory compiler takes in the parameters of the memory (such as width, depth, and number of write enable signals), and generates the files describing the layout and behavior of that memory on an ASIC chip. The layout is used when creating the chip. The behavior is modeled in Verilog code and used for simulation. This behavioral model is also instantiated in the Verilog code for the Single Chip Mote digital system (in place of the memories used on the FPGA version), and the scripts that create the final chip know to use the layout generated by the memory compiler.

7.3 Scan Chain Insertion and Debug Interface

Most, if not all, ASIC designs include scan-chain registers to test the digital logic for manufacturing errors. The scan chain is used to apply input test vectors to a module and then read out the output of that module. A scan chain can also be used to control

more advanced debug and benchmarking hardware included on the chip. JTAG is a commonly-used standard for scan chain insertion and control. The advantage of the JTAG standard is that hardware used to communicate with a chip via JTAG from a computer is commercially available. However, most researchers use their own custom scan chain and debug interface. This would require developing additional hardware and accompanying software to communicate with the chip through a test program executed from a computer. Similar to the bootloader, Verilog code can be written for an FPGA to communicate with both the debug interface on the chip and a computer using a serial port or USB. The advantage of a custom debug interface is the ability to implement more advanced features not found in the JTAG standard. In either case, scan chain insertion, where scan chain registers are placed throughout the design, is done in Synopsys Design Compiler, after the Verilog design is synthesized.

7.4 Integrated Logic Analyzer

Outside of the scope of the Single Chip Mote project, an integrated logic analyzer unit is being designed in collaboration with graduate student Nathaniel Mailoa and undergraduate student Jimin Yoon. This project, nicknamed BearClaw, intends to mimic the functionality of the ChipScope integrated logic analyzer, for the purposes of debugging ASIC designs. Multiple signals within a design are connected to the inputs of BearClaw (under the restriction that they are all in the same clock domain), and any subset of those signals can be sampled into a dedicated memory in real-time. BearClaw is designed such that any debug interface can be used to configure which signals to sample (of the ones that are permanently connected the inputs), the conditions for triggering a sample, and read the data out of the memory. Once the data is transferred to a computer, it can be arranged and plotted as digital waveforms for debugging purposes. The addition of BearClaw to the Single Chip Mote may aid in diagnosing transient errors that are difficult or impossible to replicate in simulation (for example, issues when sending or receiving radio packets).

7.5 Optical Serial Interface

The current method of bootloading allows the firmware to use the 3 Wire Bus, the radio, or UART to receive the main software code for the instruction RAM. Both the 3 Wire Bus and UART require a physical connection to the chip, which is inconvenient and unreliable if the chip is not soldered to a printed circuit board (PCB). Using the radio removes the need for a PCB; however, in order to ensure reliable delivery, the firmware should implement a network protocol stack, such as OpenWSN [26], which requires larger and more complex firmware code.

An alternative solution is to add a low-power optical receiver, such as the one described in [17], to the Single Chip Mote. To keep the interface simple, data can be transmitted using a protocol similar to (or exactly the same as) UART, using light in place of a physical connection. This method is simple and lightweight in terms of both hardware and software, and has the added advantage that multiple Single Chip Motes can be programmed at once.

Unfortunately, the optical receiver circuit has not been implemented. Once it

is complete, the Single Chip Mote Digital system can use a modified version of the APBUART module to connect to this circuit. The final steps are to write the proper firmware for the Single Chip Mote and design an optical transmitter.

7.6 Changes to Top-Level IOs

The external power-on reset and clock generator will require the following changes to the top level IOs:

- Remove the CLK and RESETn inputs
- Remove the tx_clk output and the rx_clk input
- If present, remove the clk_3wb input
- Add an input for HCLK, CLK_TX, CLK_RX, CLK_3WB, and CLK_RFTIMER. These clocks used to come from the PON module and connect to the rest of the system.
- Add an output for the clock enable signals, CLK_RX_EN and CLK_3WB_EN. There is also the option of adding outputs for CLK_TX_EN and CLK_RFTIMER_EN. The current RFcontroller module does not have a clock enable for CLK_TX but this can be added if necessary. The current RFTIMER module does not have a clock enable for CLK_RFTIMER but this can be added if necessary.
- Add an output for the SYSRESETREQ signal from the Cortex-M0.
- Add an input for the hard reset, HARD_RESETn, and the soft reset, HRESETn. These resets used to come form the PON module and connect to the rest of the system.

The chosen scan chain and debug interface will also have its own set of IOs. If an optical interface is added for bootloading, it will also require additional IOs.

Chapter 8

Conclusion

This report serves to document the two years of work behind the development of the Single Chip Mote digital system, and pass on the knowledge obtained during this process to those who continue to iterate and improve on this initial design. A tested and functioning FPGA prototype is presented, with a built-in ARM Cortex-M0 microprocessor, radio controller, radio timer, and ADC interface. Instructions on how to install the FPGA toolchain and software development tools are included, as well as overviews of their purpose and use in the Single Chip Mote project. This document also covers the testing procedures used to verify this design, and the changes required to take the FPGA-based design and create an ASIC.

The established architecture and interfaces to the radio and analog circuits are merely the bare minimum required for a fully-functioning Single Chip Mote; this project still has a long way to go before it is ready to interface with embedded sensors and microrobots. In the short-term, this project still requires an interface for hardware debugging, and system-level simulations for its mixed-signal interfaces, before it is ready for tapeout in August 2016. In the long-term, this project requires a dedicated group of hardware and software designers to converge on the preferred system-level specifications for the ideal wireless sensor node and microrobot controller. The Single Chip Mote digital system also lacks power management hardware for powering down modules that are not in use. Once it is completely solar powered, the Single Chip Mote will also need on-chip nonvolatile memory and brownout detection circuitry to operate in environments with inconsistent levels of illumination, and energy storage solutions to continue operating in environments with little or no light. Finally, achieving the optimal design in terms of energy consumption requires design space exploration to find the best combination of voltage and frequency while still meeting the requirements of software developers.

The Single Chip Mote is an incredibly ambitious project. Integrating a fully-functioning microprocessor, radio, and sensors onto a single die with zero external components is unprecedented in both academia and industry. That being said, the Single Chip Mote team is composed of hardworking and resourceful engineers, who will undoubtedly prove that this is both achievable and useful for real-world applications.

Appendix A

Appendix

A.1 AHBLiteArbiter_V2 State Transition Table

This table lists all possible combinations of inputs and state for AHBLiteArbiter_V2, and lists the next state for each combination as well as any actions that must be taken. The names of the columns are abbreviated versions of the signals in AHBLiteArbiter_V2 and are described below:

current_aphase This column corresponds to the current_address_phase signal. The possible values in the column are PASS_M0, LATCH_M0, and LATCH_M1. These three values correspond with the three address phase states.

current_dphase This column corresponds to the current_data_phase signal. The possible values in the column are NONE, M0, and M1. These three values correspond with the three data phase states.

latched_M1 This column corresponds to the inputs_latched_M1 signal. The possible values are 0 and 1.

req_M1 This column represents the req_M1 signal. The possible values are 0 and 1.

latched_M0 This column corresponds to the inputs_latched_M0 signal. The possible values are 0 and 1.

req_M0 This column represents the req_M0 signal. The possible values are 0 and 1.

HREADY This column represents the HREADYOUT_S signal. The possible values are 0 and 1.

next_aphase This column corresponds to the next_address_phase signal. The possible values in the column are PASS_M0, LATCH_M0, and LATCH_M1. These three values correspond with the three address phase states. If this column is blank, then the combination of state and inputs is invalid.

next_dphase This column corresponds to the `next_data_phase` signal. The possible values in the column are NONE, M0, and M1. These three values correspond with the three data phase states. If this column is blank, then the combination of state and inputs is invalid.

notes/actions This column is used to indicate whether a combination of state and inputs is invalid or if there are any actions that must be taken based on this combination of state and inputs. The possible actions are to latch or clear the address phase signals from M0 or M1. These correspond to the `latch_M0`, `latch_M1`, `clr_M0`, and `clr_M1` signals.

current_aphase	current_dphase	latched_M1	req_M1	latched_M0	req_M0	HREADY	next_aphase	next_dphase	notes/actions
PASS_M0	NONE	0	0	0	0	0	PASS_M0	NONE	
PASS_M0	NONE	0	0	0	0	1	PASS_M0	NONE	
PASS_M0	NONE	0	0	0	1	0	PASS_M0	M0	
PASS_M0	NONE	0	0	0	1	1	PASS_M0	M0	
PASS_M0	NONE	0	0	1	0	0			invalid state
PASS_M0	NONE	0	0	1	0	1			invalid state
PASS_M0	NONE	0	0	1	1	0			invalid state
PASS_M0	NONE	0	0	1	1	1			invalid state
PASS_M0	NONE	0	1	0	0	0	LATCH_M1	NONE	latch M1 signals
PASS_M0	NONE	0	1	0	0	1	LATCH_M1	NONE	latch M1 signals
PASS_M0	NONE	0	1	0	1	0	LATCH_M1	M0	latch M1 signals
PASS_M0	NONE	0	1	0	1	1	LATCH_M1	M0	latch M1 signals
PASS_M0	NONE	0	1	1	0	0			invalid state
PASS_M0	NONE	0	1	1	0	1			invalid state
PASS_M0	NONE	0	1	1	1	0			invalid state
PASS_M0	NONE	0	1	1	1	1			invalid state
PASS_M0	NONE	1	0	0	0	0			invalid state
PASS_M0	NONE	1	0	0	0	1			invalid state
PASS_M0	NONE	1	0	0	1	0			invalid state
PASS_M0	NONE	1	0	0	1	1			invalid state
PASS_M0	NONE	1	0	1	0	0			invalid state
PASS_M0	NONE	1	0	1	0	1			invalid state
PASS_M0	NONE	1	0	1	1	0			invalid state
PASS_M0	NONE	1	0	1	1	1			invalid state
PASS_M0	NONE	1	1	0	0	0			invalid state
PASS_M0	NONE	1	1	0	0	1			invalid state
PASS_M0	NONE	1	1	0	1	0			invalid state
PASS_M0	NONE	1	1	0	1	1			invalid state
PASS_M0	NONE	1	1	1	0	0			invalid state
PASS_M0	NONE	1	1	1	0	1			invalid state
PASS_M0	NONE	1	1	1	1	0			invalid state
PASS_M0	NONE	1	1	1	1	1			invalid state
PASS_M0	M0	0	0	0	0	0	PASS_M0	M0	
PASS_M0	M0	0	0	0	0	1	PASS_M0	NONE	
PASS_M0	M0	0	0	0	1	0	LATCH_M0	M0	latch M0 signals
PASS_M0	M0	0	0	0	1	1	PASS_M0	M0	
PASS_M0	M0	0	0	1	0	0			invalid state
PASS_M0	M0	0	0	1	0	1			invalid state
PASS_M0	M0	0	0	1	1	0			invalid state
PASS_M0	M0	0	0	1	1	1			invalid state
PASS_M0	M0	0	1	0	0	0	LATCH_M1	M0	latch M1 signals
PASS_M0	M0	0	1	0	0	1	LATCH_M1	NONE	latch M1 signals
PASS_M0	M0	0	1	0	1	0	LATCH_M0	M0	latch M0 and M1 signals
PASS_M0	M0	0	1	0	1	1	LATCH_M1	M0	latch M1 signals
PASS_M0	M0	0	1	1	0	0			invalid state
PASS_M0	M0	0	1	1	0	1			invalid state
PASS_M0	M0	0	1	1	1	0			invalid state
PASS_M0	M0	0	1	1	1	1			invalid state
PASS_M0	M0	1	0	0	0	0			invalid state
PASS_M0	M0	1	0	0	0	1			invalid state
PASS_M0	M0	1	0	0	1	0			invalid state
PASS_M0	M0	1	0	0	1	1			invalid state
PASS_M0	M0	1	0	1	0	0			invalid state
PASS_M0	M0	1	0	1	0	1			invalid state
PASS_M0	M0	1	0	1	1	0			invalid state
PASS_M0	M0	1	0	1	1	1			invalid state
PASS_M0	M0	1	1	0	0	0			invalid state
PASS_M0	M0	1	1	0	0	1			invalid state
PASS_M0	M0	1	1	0	1	0			invalid state
PASS_M0	M0	1	1	0	1	1			invalid state
PASS_M0	M0	1	1	1	0	0			invalid state
PASS_M0	M0	1	1	1	0	1			invalid state
PASS_M0	M0	1	1	1	1	0			invalid state
PASS_M0	M0	1	1	1	1	1			invalid state
PASS_M0	M1	0	0	0	0	0	PASS_M0	M1	
PASS_M0	M1	0	0	0	0	1	PASS_M0	NONE	
PASS_M0	M1	0	0	0	1	0	LATCH_M0	M1	latch M0 signals
PASS_M0	M1	0	0	0	1	1	PASS_M0	M0	
PASS_M0	M1	0	0	1	0	0			invalid state
PASS_M0	M1	0	0	1	0	1			invalid state
PASS_M0	M1	0	0	1	1	0			invalid state
PASS_M0	M1	0	0	1	1	1			invalid state
PASS_M0	M1	0	1	0	0	0	LATCH_M1	M1	latch M1 signals
PASS_M0	M1	0	1	0	0	1	LATCH_M1	NONE	latch M1 signals
PASS_M0	M1	0	1	0	1	0	LATCH_M0	M1	latch M0 and M1 signals
PASS_M0	M1	0	1	0	1	1	LATCH_M1	M0	latch M1 signals

current_aphase	current_dphase	latched_M1	req_M1	latched_M0	req_M0	HREADY	next_aphase	next_dphase	notes/actions
PASS_M0	M1	0	1	1	0	0			invalid state
PASS_M0	M1	0	1	1	0	1			invalid state
PASS_M0	M1	0	1	1	1	0			invalid state
PASS_M0	M1	0	1	1	1	1			invalid state
PASS_M0	M1	1	0	0	0	0			invalid state
PASS_M0	M1	1	0	0	0	1			invalid state
PASS_M0	M1	1	0	0	1	0			invalid state
PASS_M0	M1	1	0	0	1	1			invalid state
PASS_M0	M1	1	0	1	0	0			invalid state
PASS_M0	M1	1	0	1	0	1			invalid state
PASS_M0	M1	1	0	1	1	0			invalid state
PASS_M0	M1	1	0	1	1	1			invalid state
PASS_M0	M1	1	1	0	0	0			invalid state
PASS_M0	M1	1	1	0	0	1			invalid state
PASS_M0	M1	1	1	0	1	0			invalid state
PASS_M0	M1	1	1	0	1	1			invalid state
PASS_M0	M1	1	1	1	0	0			invalid state
PASS_M0	M1	1	1	1	0	1			invalid state
PASS_M0	M1	1	1	1	1	0			invalid state
PASS_M0	M1	1	1	1	1	1			invalid state
LATCH_M0	NONE	0	0	0	0	0			invalid state
LATCH_M0	NONE	0	0	0	0	1			invalid state
LATCH_M0	NONE	0	0	0	1	0			invalid state
LATCH_M0	NONE	0	0	0	1	1			invalid state
LATCH_M0	NONE	0	0	1	0	0	PASS_M0	M0	clear M0 signals
LATCH_M0	NONE	0	0	1	0	1	PASS_M0	M0	clear M0 signals
LATCH_M0	NONE	0	0	1	1	0	LATCH_M0	M0	latch M0 signals
LATCH_M0	NONE	0	0	1	1	1	LATCH_M0	M0	latch M0 signals
LATCH_M0	NONE	0	1	0	0	0			invalid state
LATCH_M0	NONE	0	1	0	0	1			invalid state
LATCH_M0	NONE	0	1	0	1	0			invalid state
LATCH_M0	NONE	0	1	0	1	1			invalid state
LATCH_M0	NONE	0	1	1	0	0	LATCH_M1	M0	clear M0 signals latch M1 signals
LATCH_M0	NONE	0	1	1	0	1	LATCH_M1	M0	clear M0 signals latch M1 signals
LATCH_M0	NONE	0	1	1	1	0	LATCH_M1	M0	latch M0 and M1 signals
LATCH_M0	NONE	0	1	1	1	1	LATCH_M1	M0	latch M0 and M1 signals
LATCH_M0	NONE	1	0	0	0	0			invalid state
LATCH_M0	NONE	1	0	0	0	1			invalid state
LATCH_M0	NONE	1	0	0	1	0			invalid state
LATCH_M0	NONE	1	0	0	1	1			invalid state
LATCH_M0	NONE	1	0	1	0	0	LATCH_M1	M0	clear M0 signals
LATCH_M0	NONE	1	0	1	0	1	LATCH_M1	M0	clear M0 signals
LATCH_M0	NONE	1	0	1	1	0	LATCH_M1	M0	latch M0 signals
LATCH_M0	NONE	1	0	1	1	1	LATCH_M1	M0	latch M0 signals
LATCH_M0	NONE	1	1	0	0	0			invalid state
LATCH_M0	NONE	1	1	0	0	1			invalid state
LATCH_M0	NONE	1	1	0	1	0			invalid state
LATCH_M0	NONE	1	1	0	1	1			invalid state
LATCH_M0	NONE	1	1	1	0	0	LATCH_M1	M0	clear M0 signals
LATCH_M0	NONE	1	1	1	0	1	LATCH_M1	M0	clear M0 signals
LATCH_M0	NONE	1	1	1	1	0	LATCH_M1	M0	latch M0 signals
LATCH_M0	NONE	1	1	1	1	1	LATCH_M1	M0	latch M0 signals
LATCH_M0	M0	0	0	0	0	0			invalid state
LATCH_M0	M0	0	0	0	0	1			invalid state
LATCH_M0	M0	0	0	0	1	0			invalid state
LATCH_M0	M0	0	0	0	1	1			invalid state
LATCH_M0	M0	0	0	1	0	0	LATCH_M0	M0	
LATCH_M0	M0	0	0	1	0	1	PASS_M0	M0	clear M0 signals
LATCH_M0	M0	0	0	1	1	0	LATCH_M0	M0	
LATCH_M0	M0	0	0	1	1	1	PASS_M0	M0	clear M0 signals
LATCH_M0	M0	0	1	0	0	0			invalid state
LATCH_M0	M0	0	1	0	0	1			invalid state
LATCH_M0	M0	0	1	0	1	0			invalid state
LATCH_M0	M0	0	1	0	1	1			invalid state
LATCH_M0	M0	0	1	1	0	0	LATCH_M0	M0	latch M1 signals
LATCH_M0	M0	0	1	1	0	1	LATCH_M1	M0	clear M0 signals latch M1 signals
LATCH_M0	M0	0	1	1	1	0	LATCH_M0	M0	latch M1 signals
LATCH_M0	M0	0	1	1	1	1	LATCH_M1	M0	clear M0 signals latch M1 signals
LATCH_M0	M0	1	0	0	0	0			invalid state
LATCH_M0	M0	1	0	0	0	1			invalid state
LATCH_M0	M0	1	0	0	1	0			invalid state
LATCH_M0	M0	1	0	0	1	1			invalid state
LATCH_M0	M0	1	0	1	0	0	LATCH_M0	M0	
LATCH_M0	M0	1	0	1	0	1	LATCH_M1	M0	clear M0 signals
LATCH_M0	M0	1	0	1	1	0	LATCH_M0	M0	
LATCH_M0	M0	1	0	1	1	1	LATCH_M1	M0	clear M0 signals
LATCH_M0	M0	1	1	0	0	0			invalid state
LATCH_M0	M0	1	1	0	0	1			invalid state
LATCH_M0	M0	1	1	0	1	0			invalid state
LATCH_M0	M0	1	1	0	1	1			invalid state
LATCH_M0	M0	1	1	1	0	0	LATCH_M0	M0	
LATCH_M0	M0	1	1	1	0	1	LATCH_M1	M0	clear M0 signals
LATCH_M0	M0	1	1	1	1	0	LATCH_M0	M0	
LATCH_M0	M0	1	1	1	1	1	LATCH_M1	M0	clear M0 signals
LATCH_M0	M1	0	0	0	0	0			invalid state
LATCH_M0	M1	0	0	0	0	1			invalid state
LATCH_M0	M1	0	0	0	1	0			invalid state
LATCH_M0	M1	0	0	0	1	1			invalid state
LATCH_M0	M1	0	0	1	0	0	LATCH_M0	M1	
LATCH_M0	M1	0	0	1	0	1	PASS_M0	M0	clear M0 signals
LATCH_M0	M1	0	0	1	1	0	LATCH_M0	M1	
LATCH_M0	M1	0	0	1	1	1	LATCH_M0	M0	latch M0 signals
LATCH_M0	M1	0	1	0	0	0			invalid state
LATCH_M0	M1	0	1	0	0	1			invalid state
LATCH_M0	M1	0	1	0	1	0			invalid state
LATCH_M0	M1	0	1	0	1	1			invalid state
LATCH_M0	M1	0	1	1	0	0	LATCH_M0	M1	latch M1 signals
LATCH_M0	M1	0	1	1	0	1	LATCH_M1	M0	clear M0 signals latch M1 signals

current_aphase	current_dphase	latched_M1	req_M1	latched_M0	req_M0	HREADY	next_aphase	next_dphase	notes/actions
LATCH_M0	M1	0	1	1	1	0	LATCH_M0	M1	latch M1 signals
LATCH_M0	M1	0	1	1	1	1	LATCH_M1	M0	latch M0 and M1 signals
LATCH_M0	M1	1	0	0	0	0			invalid state
LATCH_M0	M1	1	0	0	0	1			invalid state
LATCH_M0	M1	1	0	0	1	0			invalid state
LATCH_M0	M1	1	0	0	1	1			invalid state
LATCH_M0	M1	1	0	1	0	0	LATCH_M0	M1	
LATCH_M0	M1	1	0	1	0	1	LATCH_M1	M0	clear M0 signals
LATCH_M0	M1	1	0	1	1	0	LATCH_M0	M1	
LATCH_M0	M1	1	0	1	1	1	LATCH_M1	M0	latch M0 signals
LATCH_M0	M1	1	1	0	0	0			invalid state
LATCH_M0	M1	1	1	0	0	1			invalid state
LATCH_M0	M1	1	1	0	1	0			invalid state
LATCH_M0	M1	1	1	0	1	1			invalid state
LATCH_M0	M1	1	1	1	0	0	LATCH_M0	M1	
LATCH_M0	M1	1	1	1	0	1	LATCH_M1	M0	clear M0 signals
LATCH_M0	M1	1	1	1	1	0	LATCH_M0	M1	
LATCH_M0	M1	1	1	1	1	1	LATCH_M1	M0	latch M0 signals
LATCH_M1	NONE	0	0	0	0	0			invalid state
LATCH_M1	NONE	0	0	0	0	1			invalid state
LATCH_M1	NONE	0	0	0	1	0			invalid state
LATCH_M1	NONE	0	0	0	1	1			invalid state
LATCH_M1	NONE	0	0	1	0	0			invalid state
LATCH_M1	NONE	0	0	1	0	1			invalid state
LATCH_M1	NONE	0	0	1	1	0			invalid state
LATCH_M1	NONE	0	0	1	1	1			invalid state
LATCH_M1	NONE	0	1	0	0	0			invalid state
LATCH_M1	NONE	0	1	0	0	1			invalid state
LATCH_M1	NONE	0	1	0	1	0			invalid state
LATCH_M1	NONE	0	1	0	1	1			invalid state
LATCH_M1	NONE	0	1	1	0	0			invalid state
LATCH_M1	NONE	0	1	1	0	1			invalid state
LATCH_M1	NONE	0	1	1	1	0			invalid state
LATCH_M1	NONE	0	1	1	1	1			invalid state
LATCH_M1	NONE	1	0	0	0	0	PASS_M0	M1	clear M1 signals
LATCH_M1	NONE	1	0	0	0	1	PASS_M0	M1	clear M1 signals
LATCH_M1	NONE	1	0	0	1	0	LATCH_M0	M1	latch M0 signals clear M1 signals
LATCH_M1	NONE	1	0	0	1	1	LATCH_M0	M1	latch M0 signals clear M1 signals
LATCH_M1	NONE	1	0	1	0	0	LATCH_M0	M1	clear M1 signals
LATCH_M1	NONE	1	0	1	0	1	LATCH_M0	M1	clear M1 signals
LATCH_M1	NONE	1	0	1	1	0	LATCH_M0	M1	clear M1 signals
LATCH_M1	NONE	1	0	1	1	1	LATCH_M0	M1	clear M1 signals
LATCH_M1	NONE	1	1	0	0	0	LATCH_M1	M1	latch M1 signals
LATCH_M1	NONE	1	1	0	0	1	LATCH_M1	M1	latch M1 signals
LATCH_M1	NONE	1	1	0	1	0	LATCH_M0	M1	latch M0 and M1 signals
LATCH_M1	NONE	1	1	0	1	1	LATCH_M0	M1	latch M0 and M1 signals
LATCH_M1	NONE	1	1	1	0	0	LATCH_M0	M1	latch M1 signals
LATCH_M1	NONE	1	1	1	0	1	LATCH_M0	M1	latch M1 signals
LATCH_M1	NONE	1	1	1	1	0	LATCH_M0	M1	latch M1 signals
LATCH_M1	NONE	1	1	1	1	1	LATCH_M0	M1	latch M1 signals
LATCH_M1	M0	0	0	0	0	0			invalid state
LATCH_M1	M0	0	0	0	0	1			invalid state
LATCH_M1	M0	0	0	0	1	0			invalid state
LATCH_M1	M0	0	0	0	1	1			invalid state
LATCH_M1	M0	0	0	1	0	0			invalid state
LATCH_M1	M0	0	0	1	0	1			invalid state
LATCH_M1	M0	0	0	1	1	0			invalid state
LATCH_M1	M0	0	0	1	1	1			invalid state
LATCH_M1	M0	0	1	0	0	0			invalid state
LATCH_M1	M0	0	1	0	0	1			invalid state
LATCH_M1	M0	0	1	0	1	0			invalid state
LATCH_M1	M0	0	1	0	1	1			invalid state
LATCH_M1	M0	0	1	1	0	0			invalid state
LATCH_M1	M0	0	1	1	0	1			invalid state
LATCH_M1	M0	0	1	1	1	0			invalid state
LATCH_M1	M0	0	1	1	1	1			invalid state
LATCH_M1	M0	1	0	0	0	0	LATCH_M1	M0	
LATCH_M1	M0	1	0	0	0	1	PASS_M0	M1	clear M1 signals
LATCH_M1	M0	1	0	0	1	0	LATCH_M1	M0	latch M0 signals
LATCH_M1	M0	1	0	0	1	1	LATCH_M0	M1	latch M0 signals clear M1 signals
LATCH_M1	M0	1	0	1	0	0	LATCH_M1	M0	
LATCH_M1	M0	1	0	1	0	1	LATCH_M0	M1	clear M1 signals
LATCH_M1	M0	1	0	1	1	0	LATCH_M1	M0	
LATCH_M1	M0	1	0	1	1	1	LATCH_M0	M1	clear M1 signals
LATCH_M1	M0	1	1	0	0	0	LATCH_M1	M0	
LATCH_M1	M0	1	1	0	0	1	LATCH_M1	M1	latch M1 signals
LATCH_M1	M0	1	1	0	1	0	LATCH_M1	M0	latch M0 signals
LATCH_M1	M0	1	1	0	1	1	LATCH_M0	M1	latch M0 and M1 signals
LATCH_M1	M0	1	1	1	0	0	LATCH_M1	M0	
LATCH_M1	M0	1	1	1	0	1	LATCH_M0	M1	latch M1 signals
LATCH_M1	M0	1	1	1	1	0	LATCH_M1	M0	
LATCH_M1	M0	1	1	1	1	1	LATCH_M0	M1	latch M1 signals
LATCH_M1	M1	0	0	0	0	0			invalid state
LATCH_M1	M1	0	0	0	0	1			invalid state
LATCH_M1	M1	0	0	0	1	0			invalid state
LATCH_M1	M1	0	0	0	1	1			invalid state
LATCH_M1	M1	0	0	1	0	0			invalid state
LATCH_M1	M1	0	0	1	0	1			invalid state
LATCH_M1	M1	0	0	1	1	0			invalid state
LATCH_M1	M1	0	0	1	1	1			invalid state
LATCH_M1	M1	0	1	0	0	0			invalid state
LATCH_M1	M1	0	1	0	0	1			invalid state
LATCH_M1	M1	0	1	0	1	0			invalid state
LATCH_M1	M1	0	1	0	1	1			invalid state
LATCH_M1	M1	0	1	1	0	0			invalid state
LATCH_M1	M1	0	1	1	0	1			invalid state
LATCH_M1	M1	0	1	1	1	0			invalid state
LATCH_M1	M1	0	1	1	1	1			invalid state

current_aphase	current_dphase	latched_M1	req_M1	latched_M0	req_M0	HREADY	next_aphase	next_dphase	notes/actions
LATCH_M1	M1	1	0	0	0	0	LATCH_M1	M1	
LATCH_M1	M1	1	0	0	0	1	PASS_M0	M1	clear M1 signals
LATCH_M1	M1	1	0	0	1	0	LATCH_M1	M1	latch M0 signals
LATCH_M1	M1	1	0	0	1	1	LATCH_M0	M1	latch M0 signals clear M1 signals
LATCH_M1	M1	1	0	1	0	0	LATCH_M1	M1	
LATCH_M1	M1	1	0	1	0	1	LATCH_M0	M1	clear M1 signals
LATCH_M1	M1	1	0	1	1	0	LATCH_M1	M1	
LATCH_M1	M1	1	0	1	1	1	LATCH_M0	M1	clear M1 signals
LATCH_M1	M1	1	1	0	0	0	LATCH_M1	M1	
LATCH_M1	M1	1	1	0	0	1	PASS_M0	M1	clear M1 signals
LATCH_M1	M1	1	1	0	1	0	LATCH_M1	M1	latch M0 signals
LATCH_M1	M1	1	1	0	1	1	LATCH_M0	M1	latch M0 signals clear M1 signals
LATCH_M1	M1	1	1	1	0	0	LATCH_M1	M1	
LATCH_M1	M1	1	1	1	0	1	LATCH_M0	M1	clear M1 signals
LATCH_M1	M1	1	1	1	1	0	LATCH_M1	M1	
LATCH_M1	M1	1	1	1	1	1	LATCH_M0	M1	clear M1 signals

Bibliography

[1] *AMBA 3 AHB-Lite Protocol Specification*. Version 1.0. ARM. 2008. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0033a/index.html.

[2] *AMBA 3 APB Protocol Specification*. Version 1.0. ARM. 2008. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0024b/index.html.

[3] *ARM compiler toolchain. Linker Reference*. Version 5.0. ARM. 2011. URL: http://infocenter.arm.com/help/topic/com.arm.doc.dui0493e/DUI0493E_arm_linker_reference.pdf.

[4] *Bin2Coe*. URL: https://sourceforge.net/projects/bin2coe/.

[5] Joe Bungo. *ARM Cortex-M0 DesignStart Processor and V6-M Architecture*. ARM. URL: http://www.sase.com.ar/2012/files/2012/09/M0_v6M_Q312.pdf.

[6] *CC2538 Powerfull Wireless Microcontroller System-On-Chip for 2.4-GHz IEEE 802.15.4, 6LoWPAN, and ZigBee Applications*. SWRS096D. Texas Instruments. 2015. URL: http://www.ti.com/lit/ds/symlink/cc2538.pdf.

[7] A. C. K. Chan et al. "Low power wireless sensor node for human centered transportation system". In: *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*. 2012, pp. 1542–1545. DOI: 10.1109/ICSMC.2012.6377955.

[8] Peter Alfke Clifford E Cummings. "Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons". In: *SNUG 2002 (Synopsys Userse Group Conference, San Jose, CA, 2002) User Papers*. Sunburst Design. 2002. URL: http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf.

[9] *Cortex-M System Design Kit*. ARM. URL: https://www.arm.com/products/processors/cortex-m/cortex-m-system-design-kit.php.

[10] *Cortex-M0 Devices Generic User Guide*. ARM. 2009. URL: http://infocenter.arm.com/help/topic/com.arm.doc.dui0497a/DUI0497A_cortex_m0_r0p0_generic_ug.pdf.

[11] *DesignStart for Processor IP*. ARM. URL: http://www.arm.com/products/processors/designstart-processor-ip/.

[12] *Digilent Adept 2*. Digilent. URL: https://reference.digilentinc.com/digilent_adept_2.

[13] *Guide: Getting Xilinx ISE to work with Windows 8 / Windows 10 (64-bit).* URL: http://www.eevblog.com/forum/microcontrollers/guide-getting-xilinx-ise-to-work-with-windows-8-64-bit/.

[14] *Hardware.* URL: https://openwsn.atlassian.net/wiki/display/OW/Hardware.

[15] "IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)". In: *IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)* (2011), pp. 1–314. DOI: 10.1109/IEEESTD.2011.6012487. URL: http://ieeexplore.ieee.org/servlet/opac?punumber=6012485.

[16] *ISE Tutorial: Using Xilinx ChipScope Pro ILA Core with Project Navigator to Debug FPGA Applications.* Version 14.5. Xilinx. 2013. URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_6/ug750.pdf.

[17] G. Kim et al. "A 695 pW standby power optical wake-up receiver for wireless sensor nodes". In: *Proceedings of the IEEE 2012 Custom Integrated Circuits Conference.* 2012, pp. 1–4. DOI: 10.1109/CICC.2012.6330603.

[18] J. Lu et al. "Toward the World Smallest Wireless Sensor Nodes With Ultralow Power Consumption". In: *IEEE Sensors Journal* 14.6 (2014), pp. 2035–2041. ISSN: 1530-437X. DOI: 10.1109/JSEN.2014.2309176.

[19] *MDK Microcontroller Development Kit.* ARM. URL: http://www2.keil.com/mdk5.

[20] University of Michigan Electrical Engineering and Computer Science. *Michigan Mirco Mote (M3) Makes History.* 2015. URL: http://www.eecs.umich.edu/eecs/about/articles/2015/Worlds-Smallest-Computer-Michigan-Micro-Mote.html.

[21] *MSP430F15x, MSP430F16x, MSP430F161x Mixed Signal Microcontroller.* SLAS368G. Texas Instruments. 2011. URL: http://www.ti.com/lit/ds/slas368g/slas368g.pdf.

[22] *Multi-File Download: ISE Design - 14.6 Full Product Installation.* Xilinx. URL: http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools/v2012_4---14_6.html.

[23] *Nexys 3 Spartan-6 FPGA Trainer Board (LIMITED TIME) » see Nexys4 DDR.* Digilent. URL: http://store.digilentinc.com/nexys-3-spartan-6-fpga-trainer-board-limited-time-see-nexys4-ddr/.

[24] *Nexys 4 DDR Artix-7 FPGA: Trainer Board Recommended for ECE Curriculum.* Digilent. URL: http://store.digilentinc.com/nexys-4-ddr-artix-7-fpga-trainer-board-recommended-for-ece-curriculum.

[25] John Notor, Anthony Caviglia, and Gary Levy. *CMOS RFIC Architectures for IEEE 802.15.4 Networks.* Tech. rep. Cadence Design Systems, Inc, 2003. URL: https://www.cadence.com/rl/Resources/white_papers/CMOSRFICArchforIEEE80215.pdf.

[26] *OpenWSN Home.* URL: https://openwsn.atlassian.net/wiki/display/OW?src=breadcrumbs-homepage.

[27] C. Park, J. Liu, and P. H. Chou. "Eco: an ultra-compact low-power wireless sensor node for real-time motion monitoring". In: *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*. 2005, pp. 398–403. DOI: 10.1109/IPSN.2005.1440956.

[28] R. Send et al. "Granular Radio EnErgy-sensing Node (GREEN): A 0.56 cm3 wireless stick-on node for non-intrusive energy monitoring". In: *SENSORS, 2013 IEEE*. 2013, pp. 1–4. DOI: 10.1109/ICSENS.2013.6688133.

[29] Karthik Shivashankar. *ARM AMBA 3 AHB-Lite*. ARM. URL: http://web.mit.edu/clarkds/www/Files/ahblite.pdf.

[30] Evgeni Stavinov. "A Practical Parallel CRC Generation Method". In: *Circuit Cellar* 234 (2010), pp. 38–45. URL: http://outputlogic.com/my-stuff/circuit-cellar-january-2010-crc.pdf.

[31] M. Tabesh et al. "A power-harvesting pad-less mm-sized 24/60GHz passive radio with on-chip antennas". In: *2014 Symposium on VLSI Circuits Digest of Technical Papers*. 2014, pp. 1–2.

[32] *VmodMIB: VHDC Module Interface Board*. Digilent. URL: http://store.digilentinc.com/vmodmib-vhdc-module-interface-board/.