Adaption of the Space Plug-and-play Interface Simulation Equipment to provide support for the Space Plug-and-play Avionics architecture.

MARTIN CARLZON

Master’s Degree Project
Stockholm, Sweden May 7, 2011

TRITA-ICT-EX-2011:83
Abstract

The Space Plug-and-play Interface Simulation Equipment (SPISE) box was developed by AAC to aid in the debugging of satellite modules. This report covers the analysis and conversion of this box to provide initial support for the Space Plug-and-play Avionics (SPA) set of protocols in a single Field Programmable Gate Array (FPGA).

A major part of this conversion is the design of SPA-U support for which both a host and slave Universal Serial Bus 1.1 (USB 1.1) stack is needed. The comparison of methods to get this functionality and the final design of the Intellectual Property (IP) cores which are implemented in the Very High Speed Integrated Circuit Hardware Description Language (VHDL) language utilizing the freely available from OpenCores, USBHostSlave controller IP by Steve Fielding.

The intended target hardware for this project is the reuse of the existing platform, an Actel A3P1000 FPGA containing one million gates.

Described in this report are the necessary choices made to implement support for the SPA protocols within a limited time and budget while maintaining compatibility with the architecture of the SPISE box.
## Contents

1 Abbreviations .................................................. 1

2 Introduction ................................................... 3
   2.1 Background .............................................. 3
   2.2 Plug-and-Play satellites .............................. 3
      2.2.1 Standard sizes ................................... 3
      2.2.2 Standard buses .................................. 3
      2.2.3 Self describing modules ......................... 4
   2.3 SPA-X protocols ........................................ 4
      2.3.1 SPA-1 .............................................. 4
      2.3.2 SPA-U .............................................. 4
      2.3.3 SPA-S .............................................. 4
      2.3.4 Extended Transducer Electronic Datasheets ... 5
   2.4 The SPISE box ........................................... 5
      2.4.1 Supported protocols ............................... 5
   2.5 SPA-X support in the SPISE box ..................... 6
      2.5.1 FPGA considerations ............................ 6
      2.5.2 Hardware considerations ....................... 6
   2.6 Expected outcome ...................................... 7

3 SPISE box internals .......................................... 9
   3.1 SPISE bus layout ....................................... 9
   3.2 SPISE bus protocol forwarding overview .......... 9
   3.3 SPISE box internal bus protocol .................... 10
      3.3.1 Messages ......................................... 10
      3.3.2 Forwarding data ................................ 10
      3.3.3 Timing ............................................ 11

4 Investigation of alternative USB 1.1 implementations 13
   4.1 Overview of the SPISE architecture ................ 13
   4.2 Selection of USB 1.1 implementation ............. 14
      4.2.1 Hardware solutions considered ................ 14
      4.2.2 USB IP blocks considered ..................... 15
      4.2.3 Hardware solution versus IP block .......... 16
      4.2.4 Summary of selection .......................... 16
   4.3 Analyzing a USB implementation ................... 17
      4.3.1 Technique used ................................ 17
5 Implementation

5.1 SPISE to Protocol block, interface overview ........................................ 19
5.2 Common building blocks ................................................................. 19
  5.2.1 Falling edge detector ............................................................... 19
  5.2.2 T-flipflop ................................................................................. 20
  5.2.3 Asynchronous RESET synchronizer ........................................... 21
5.3 USB common .................................................................................. 21
5.4 Compliance with USB 1.1 standard .................................................... 22
  5.4.1 USB FSM ................................................................................ 22
  5.4.2 Wishbone FSM ......................................................................... 22
  5.4.3 FIFO FSM ................................................................................ 23
5.5 USB Slave block .............................................................................. 24
  5.5.1 USB 1.1 Slave functionality summary ........................................ 24
5.6 USB Host block ............................................................................... 25
  5.6.1 Device state tracking ................................................................. 25
  5.6.2 USB 1.1 Host functionality summary ........................................ 26
5.7 SPISE bus interface conversion block ............................................... 28
  5.7.1 Generate output based on timing ............................................... 28
  5.7.2 Using the generated timing ....................................................... 29
  5.7.3 Problems with internal tri-state ................................................ 29
  5.7.4 Filtering data written to the protocol block .................................. 30

6 Hardware .............................................................................................. 33

6.1 SPISE box ...................................................................................... 33
6.2 PCB Design .................................................................................... 33
  6.2.1 Slave/Peripheral ....................................................................... 35
  6.2.2 Host ......................................................................................... 36

7 Verification .......................................................................................... 37

7.1 IC .................................................................................................. 37
7.2 USB Slave ...................................................................................... 38
  7.2.1 Topics covered in this test ....................................................... 38
7.3 USB Host ....................................................................................... 39
  7.3.1 Topics covered in this test ....................................................... 39
  7.3.2 Simulation of USB 1.1 PHY .................................................... 39
7.4 Host and Slave ping-pong ............................................................... 40
  7.4.1 Topics covered in this test ....................................................... 40
7.5 Single Slave to PC - SETUP and detection ...................................... 41
7.6 Single Host to Slave device - SETUP and detection ....................... 41
7.7 ASIM ............................................................................................ 41
  7.7.1 Software for testing the ASIM .................................................. 42
7.8 PC and ASIM bridge ....................................................................... 43
  7.8.1 Topics covered in this test ....................................................... 43
7.9 SPISE box conv block ................................................................... 43
  7.9.1 The verification step in more detail ......................................... 44
  7.9.2 Topics covered by this test ....................................................... 44
7.10 Hardware ....................................................................................... 45
7.11 Logic verification ........................................................................... 46
  7.11.1 Bridged USB analyzer dump ................................................ 46
  7.11.2 DATA toggle sequence error testing ........................................ 46
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>SPISE bus overview</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Protocol domain interconnect through the SPISE-bus</td>
<td>10</td>
</tr>
<tr>
<td>3.3</td>
<td>SPISE bus packet layout</td>
<td>10</td>
</tr>
<tr>
<td>5.1</td>
<td>Simplified view of data flow</td>
<td>20</td>
</tr>
<tr>
<td>5.2</td>
<td>falling edge() logic</td>
<td>20</td>
</tr>
<tr>
<td>5.3</td>
<td>Bit toggling memory (T-flipflop)</td>
<td>21</td>
</tr>
<tr>
<td>5.4</td>
<td>Synchronizer</td>
<td>21</td>
</tr>
<tr>
<td>5.5</td>
<td>USB protocol block</td>
<td>22</td>
</tr>
<tr>
<td>5.6</td>
<td>Wishbone FSM</td>
<td>23</td>
</tr>
<tr>
<td>5.7</td>
<td>ReadState FSM</td>
<td>28</td>
</tr>
<tr>
<td>5.8</td>
<td>SPISE box conversion block logic signals</td>
<td>28</td>
</tr>
<tr>
<td>5.9</td>
<td>SPISE bus WE logic</td>
<td>31</td>
</tr>
<tr>
<td>6.1</td>
<td>The SPISE box with mounted test-boards</td>
<td>34</td>
</tr>
<tr>
<td>6.2</td>
<td>The test PCBs mounted in the SPISE box</td>
<td>35</td>
</tr>
<tr>
<td>6.3</td>
<td>A view of the bottom layer of the test PCBs</td>
<td>36</td>
</tr>
<tr>
<td>7.1</td>
<td>Output from Linux program connecting to ASIM</td>
<td>42</td>
</tr>
<tr>
<td>7.2</td>
<td>System layout</td>
<td>45</td>
</tr>
<tr>
<td>7.3</td>
<td>SYNC packet on a 1m USB cable</td>
<td>46</td>
</tr>
<tr>
<td>7.4</td>
<td>Bridged error free USB connection</td>
<td>47</td>
</tr>
<tr>
<td>7.5</td>
<td>Bridged USB connection with forced DATA sequence error</td>
<td>47</td>
</tr>
<tr>
<td>7.6</td>
<td>FPGA host SETUP sequence of the ASIM</td>
<td>48</td>
</tr>
<tr>
<td>7.7</td>
<td>Linux PC host SETUP of the FPGA Slave</td>
<td>49</td>
</tr>
<tr>
<td>7.8</td>
<td>The repeated handshake of FPGA host to ASIM SETUP</td>
<td>49</td>
</tr>
<tr>
<td>B.1</td>
<td>System view of USB</td>
<td>61</td>
</tr>
<tr>
<td>B.2</td>
<td>USB Type A connectors seen from the front</td>
<td>61</td>
</tr>
<tr>
<td>B.3</td>
<td>USB Type B connectors seen from the front</td>
<td>62</td>
</tr>
<tr>
<td>B.4</td>
<td>USBee DX capture of host sending a DATA0 packet to slave</td>
<td>63</td>
</tr>
<tr>
<td>C.1</td>
<td>Wishbone read</td>
<td>70</td>
</tr>
<tr>
<td>C.2</td>
<td>Wishbone write</td>
<td>70</td>
</tr>
</tbody>
</table>
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Truth table falling edge detector</td>
<td>20</td>
</tr>
<tr>
<td>5.2</td>
<td>Truth table toggle memory (T-flipflop), signal Z</td>
<td>20</td>
</tr>
<tr>
<td>5.3</td>
<td>I_AVAIL, as seen by the SPISE bus controller</td>
<td>24</td>
</tr>
<tr>
<td>5.4</td>
<td>Host device tracking, Command states</td>
<td>26</td>
</tr>
<tr>
<td>5.5</td>
<td>Host device tracking, Minor states</td>
<td>26</td>
</tr>
<tr>
<td>5.6</td>
<td>USB device tracking states</td>
<td>27</td>
</tr>
<tr>
<td>5.7</td>
<td>SPISE bus converter interface MUX rules</td>
<td>30</td>
</tr>
<tr>
<td>7.1</td>
<td>Simulated USB transceiver (D- and D+)</td>
<td>40</td>
</tr>
<tr>
<td>7.2</td>
<td>FPGA area usage</td>
<td>50</td>
</tr>
<tr>
<td>B.1</td>
<td>USB pin numbers for figures Figure B.2 and Figure B.3</td>
<td>61</td>
</tr>
<tr>
<td>B.2</td>
<td>bus states</td>
<td>63</td>
</tr>
<tr>
<td>B.3</td>
<td>Setup procedure</td>
<td>66</td>
</tr>
<tr>
<td>E.1</td>
<td>Setup state tracking (CTRL_STATE)</td>
<td>77</td>
</tr>
</tbody>
</table>
Chapter 1

Abbreviations

API - Application Programming Interface
ASIM - Applique Sensor Interface Module
BOM - Beginning-of-Message
DFF - D flip flop
DnC - Do not Care
EOM - End-of-Message
EOP - End of Packet
EP - Endpoint
FIFO - First In First Out
FPGA - Field Programmable Gate Array
FS - Full Speed
FSM - Finite State Machine
HC - Host Controller
HCD - Host Controller Driver
HWTB - Hardware test-bench
IP - Intellectual Property
IP - Intellectual Property
LS - Low Speed
NDA - Non Disclosure Agreement
NRZI - Non Return to Zero Invert
OBC - On Board Computer
OE - Output Enable
OS - Operating System
PAR - Place And Route
PC - Personal Computer
PCB - Printed Circuit Board
PID - Packet Identifier field
PNP - Plug And Play
RE - Read Enable
SDM - Satellite Data Model
SE0 - Single Ended Zero
SE1 - Single Ended One
SIE - Serial Interface Engine
SOF - Start of Frame
SOP - Start of Packet
SPA - Space Plug-and-play Avionics
SPISE - Space Plug-and-play Interface Simulation Equipment
SPISE box - The actual SPISE hardware
SPISE bus - The special bus format used inside the SPISE box
UHS - USBHostSlave IP core
USB - Universal Serial Bus
WE - Write Enable
xTEDS - Extended Transducer Electronic Data Sheets
Chapter 2

Introduction

2.1 Background

AAC Microtec has previously developed a box for simulation and debugging of various protocols and buses. This box goes under the name SPISE-box which means Space Plug-and-play (PnP) Interface Simulation Equipment. The SPISE box already handles multiple protocols used in satellites but these protocols are not PnP capable.

With the increasing economic pressure on companies/research departments developing satellites to be launched, the time to launch becomes an important factor in the building of these satellites. This is where the PnP satellite (section 2.2), Space Plug-and-Play Avionics (SPA) protocols (section 2.3) and the AAC developed SPISE box (section 2.4) comes into the picture.

2.2 Plug-and-Play satellites

The aim of PnP satellites is to lower the time to launch for small research satellites. To achieve this several components and communication buses are made modular, standardized and PnP capable.

2.2.1 Standard sizes

By having standardized sizes for the satellite components to be added they can be measured in predefined units. Sizes of the satellite modules can vary in integer multiples of units such as a 1U, 2U or 3U. These predefined sizes will guarantee that the new modules built will fit and be easily mounted inside of the satellite.

2.2.2 Standard buses

By moving to a standardized bus, less time can be spent on integrating modules such as different types of sensors with different interfaces. By having the sensors using the standardized hardware interface SPA-X, and presenting themselves to a controller with the kind of data they are able to provide, less time will be needed to include support for these sensors in the satellite.
2.2.3 Self describing modules

Every module added to the satellite will provide a descriptor of its function and capabilities to the controller. The controller will have been previously programmed to know how to handle these functions to, for example, read the temperature of the surroundings or the current acceleration from a sensor. These descriptors are known as Extended Transducer Electronic Data Sheets (xTEDS).

Even if a custom hardware design is already available and ready to be assembled the use of xTEDS and generalized hardware connectors will give a faster time to deployment since the module is ready to be used right away, as long as the controller knows how to handle the device described in the xTEDS received.

In summary the benefits of this approach are:

- The standardized connectors provide the means to connect new hardware into the system.
- Standardized sizes in the PnPSat design introduces modular components that will fit in the satellite.
- The xTEDS communicated between the added module and the controller provide a description of the interface and functions of the added module.

2.3 SPA-X protocols

SPA-X is the collective name of all the SPA protocols. These protocols are largely based on already existing protocols and described further in [1].

2.3.1 SPA-1

SPA-1 is a low bandwidth protocol based on I^2C. It is meant to provide connectivity between components with low demands on bandwidth. An existing I^2C core had already been tested and verified by AAC and so the I^2C protocol will not be dealt with further than how that core is controlled. A summarized overview of I^2C can be found in Appendix A.

2.3.2 SPA-U

SPA-U is a medium bandwidth protocol based on USB 1.1. Since it is based on USB it is plug-and-play capable and due to USB 1.1 specifications it has an upper bandwidth of 12 Mbit. The USB 1.1 protocol is further described in Appendix B.

2.3.3 SPA-S

SPA-S is the high bandwidth protocol based on SpaceWire. The main difference between SPA-S and pure SpaceWire is that the connector also includes power wires and four optional USB 1.1 wires as well as some wires for testing purposes. As SpaceWire is one of the protocols already existing in the box it will not be dealt with in great detail in this report.
2.3.4 Extended Transducer Electronic Datasheets

The SPA-X protocols specify the use of xTEDS to describe the functionality of the module connected. The sensors will introduce themselves through the xTEDS to the micro controller in the satellite.

xTEDS list among other things possible data readouts from a device (for example a sensor) which tells the controlling unit what measurements can be extracted from this device. In the case of a sensor the xTEDS could tell the controller that this sensor is a thermometer and the measurements available are the temperature either in Fahrenheit or Celsius.

2.4 The SPISE box

The idea is that in a laboratory somewhere in the world, a sensor is being developed. At another place someone wants to either debug this sensor or to connect it to a controller. The problem is that the controlling system is being developed in another country.

The SPISE box will act as a bridge between the sensor and controller and it will appear as both are available locally to you. With this setup it does not matter if the sensors are all in your own lab, at the other side of the world or theoretically even orbiting the earth. As long as some kind of link exist between the modules and the SPISE box supports the protocol in use there will be a way to connect and use them just as if they were in the same laboratory.

This eliminates the need to transport modules between laboratories in different parts of the world during development of new satellites. By not having to transport the modules money can be saved on cost of shipment, waiting time for modules to arrive as well as eliminating the risk of damaging the modules during transport.

2.4.1 Supported protocols

As the objective of the SPISE box is to be used as an aid in debugging various protocols and buses it needs to support as many protocols as possible. Before work was started to provide SPA-X support to the SPISE-box it contained the following protocols:

- SpaceWire
- RS232
- RS485
- CAN
- USB (DLP USB245M USB\(^1\))\(^2\)
- Ethernet (TCP/IP\(^1\)) through Wiznet NM7010B+ \(^3\)

---

\(^1\)Not usable for SPA-U
\(^2\)limited TCP/IP implementation
2.5 SPA-X support in the SPISE box

To make the SPISE box support the SPA-1, SPA-U and SPA-S set of protocols the I\textsuperscript{2}C, USB 1.1 and Space Wire protocols must all exist in the FPGA controller of the SPISE box. As well as having the FPGA support the protocols, the motherboard must also have the proper connectors and transceivers added.

2.5.1 FPGA considerations

SPA-S

The Space Wire core had already been implemented and was not subject for replacement. With Space Wire already in the SPISE box SPA-S was considered needing no further development at the protocol level.

SPA-1

AAC had previously been using, and modifying, an I\textsuperscript{2}C core downloaded from the Opencores website. This I\textsuperscript{2}C core had been modified to provide multi-master support and clock stretching and was not considered for replacement either. The I\textsuperscript{2}C core had however not been used in the SPISE box before and needs to be provided with glue logic to work with the internals of the SPISE box.

SPA-U

The current USB 1.1 implementation is provided through a hardware solution based on the DLP USB245M chip. This chip provides a FIFO interface to the USB 1.1 and does not support host mode. Since the host mode is required for a proper SPA-U transparent bridge function the DLP USB245M chip can not be used.

With investigations described in further detail in chapter 4 a new solution using the USBHostSlave \cite{4} will be implemented. The USBHostSlave core is open hardware and available on the Opencores website. This solution requires the development of a USB 1.1 stack for proper operation and while the USB 1.1 stack has to support the USB 1.1 protocol standard it also need to maintain compatibility with the SPISE box internals. If the backwards compatibility can not be maintained then SPA-1 and SPA-S solutions can not be reused.

2.5.2 Hardware considerations

A complete redesign of the motherboard is considered too lengthy to finish within the time frame available. Without a complete redesign the only option available is to use the external pin-headers found on the motherboard to connect a smaller board dedicated to only a single protocol. These external pin-headers are connected straight to the pins of the FPGA.

With time and cost of designing separate boards of the new protocols some shortcuts will be made. The I\textsuperscript{2}C and Space Wire have been previously used and proved to work at AAC and can be considered to not need new testing. USB 1.1 boards will be designed to hold the transceivers and connectors for testing the complete USB 1.1 system. There is a need for one USB 1.1 Host board (type
A connector) and one USB 1.1 Slave board (type B connector). These USB 1.1 boards must also be made according to measurements in the box to fit in next to other components.

2.6 Expected outcome

By automating the tasks of the USB 1.1 layer with state machines the expected outcome is a SPA-U capable implementation which can be connected to the existing SPISE bus. The automation is needed to eliminate the need for the SPISE bus to understand the USB 1.1 protocol and to provide the simple interface that the SPISE bus requires.

By not introducing any changes to the SPISE bus the Space Wire core can be easily reused as a transport layer in the SPISE box, providing SPA-S support. Since the I\textsuperscript{2}C core have been used in another project it does not contain an interface which can be connected to the SPISE bus. Because the difference in interface the I\textsuperscript{2}C core will need some glue logic to connect properly to the SPISE bus. With the glue logic in place it is expected that SPA-1 is available since the I\textsuperscript{2}C core has been tested and verified in other work.

By providing the USB 1.1, I\textsuperscript{2}C and Space Wire cores in a way that does not change the layout of the SPISE bus it is expected that SPA-U, SPA-1 and SPA-S support will be available at the logic level in the SPISE box.
Chapter 3

SPISE box internals

In this section will be given a brief introduction to the SPISE box internals and how the protocols are connected together over the SPISE bus. The internal SPISE bus is described in section 3.3. The protocols used for implementing SPA-X support are described in the following appendices, I²C [SPA-1] (Appendix A), USB 1.1 [SPA-U] (Appendix B) and Wishbone (Appendix C).

3.1 SPISE bus layout

Figure 3.1 shows how several protocol blocks are connected to the SPISE bus which is controlled by the arbiter. The arbiter detects data available at one port and forwards the data to the destination port. The bus can contain up to N different blocks where N is a constant decided upon at compile time. The arbiter will check one block for data available every clock cycle.

3.2 SPISE bus protocol forwarding overview

Figure 3.2 pictures two different domains connected by two different protocols. These protocols are routed together over the SPISE-bus. The SPISE-box supports multiple protocols and buses, and the destination route of a packet is decided upon when it reaches the SPISE-bus. This decision is based on which interface it arrives on hence no data needs to be parsed from the packet received. Once the source port and destination port has been decided the arbiter
will connect these two sources together through a MUX. The RX FIFO of the source port will be connected straight to the TX FIFO of the destination port.

![Diagram of protocol domain interconnect through the SPISE bus](image)

It is important to realize that the SPISE-bus does not need to know the data traveling the bus and should be seen as a pipe leading from one domain to another. The same goes for the protocol specifics. The protocol specific blocks are not aware of the format of any data traveling the protocol in question and need only take care of actual forwarding of the data from one domain to the other.

3.3 SPISE box internal bus protocol

The SPISE box internal bus was developed at ÅAC and is described in a previous report. The general layout of a transaction on the bus is shown in Figure 3.3 without timing considered.

![SPISE bus packet layout](image)

3.3.1 Messages

The internal bus is nine bits wide and uses start and stop tokens to define the beginning and end of a transmission. These nine bit tokens are called Beginning-of-Message (BOM) and End-of-Message (EOM). More than just defining the beginning of a new transmission a BOM will also contain a destination for internal routing. This means that there is no predefined size for a message traveling the SPISE bus and the length can only be decided by parsing the BOM/EOM tokens.

3.3.2 Forwarding data

The bus will know a connected block has data to transmit by checking if an available-flag is asserted or not. When the available-flag is set the bus controller asserts the read enable (RE) signal. The first data expected to be seen coming from the device is a BOM. The RE flag is then toggled low while the BOM is parsed and the destination set as specified by the BOM. After the BOM follows an amount of data which is ended by the EOM token. As soon as the BOM is seen a WE signal is asserted on the destination block causing the data to be passed through from one block to the other. When the EOM is seen RE and
WE are both disabled. Both the BOM and EOM are written into the receiving block and it is up to the logic of the connected blocks to filter these out to maintain data integrity.

3.3.3 Timing

The timing of the SPISE bus proved to be a problem.

From the documentation we have the following quote “Read Enable. Asserted by the arbiter when it wants to read from the device. The device must put data on the dout pin at the next clock cycle” - [5, Table 3] which contradicts a test-bench result in a figure, showing a 2 cycle delay ([5, Figure 12]).

One reason why this contradiction might exist is because the author mentioned that pipelining would be needed to support the clock speed required It is unknown if the pipelining was actually implemented or only suggested as a fix but some paths were still seen to be too long for the 50 MHz clock suggested. The data from the device however was delayed an extra cycle compared to the earlier mentioned specification of providing the data 1 cycle after asserting “read enable”.

Because of this, it could not be decided if the pipelining had been implemented and was unsuccessful or if it had not been implemented at all. The author of the code was no longer available and so the real case of the timing could not be determined and instead a work around had to be used.

The code was seen to use Actel FIFO memories in places and therefore the same timing was assumed to apply for the SPISE bus read and write operations as for an Actel FIFO memory [6, page 171]. The delay of such a FIFO is two cycles and that together with the delay added by logic at the arbiter causes a total of three cycles delay for data to reach the bus.

A conversion block was implemented to support this timing and it needed to handle two cases. One case is when data is leaving the SPISE box via a protocol block and the other one is when data is inbound on the protocol block destined for the internal bus. The implementation of the conversion block and how the unknown timing was handled will be described in [section 5.7].
Chapter 4

Investigation of alternative USB 1.1 implementations

While the project specification suggested the use of the USBHostSlave IP core as the way to provide USB functionality to the project, limited investigation was conducted to find alternative ways of implementing USB 1.1 support.

The main factors for the final choice being cost, lifetime of the solution and introducing as few changes as possible to the existing hardware. The alternatives considered will be listed in section 4.2.

4.1 Overview of the SPISE architecture

To make a better decision for the USB 1.1 implementation the existing architecture of the SPISE box needs to be known. At the heart of the box is the SPISE bus controller which is an IP core running inside of an Actel A3P1000 FPGA.

This controller communicates with protocol cores through a FIFO-like interface. Although timings and data flow is not as simple as a stand-alone FIFO it is possible to add glue-logic between a standard FIFO and the SPISE bus controller to achieve the proper communication protocol.

The internal SPISE bus is nine bits wide but only eight of these are used for data. The ninth bit is used as a toggle for internal messaging. Although the existing protocol blocks use a nine-bit FIFO, if properly handled by the glue-logic, the protocol blocks can use eight bit FIFOs to save some of the restricted space in the FPGA (consider Figure 5.1).

While shuffling data the SPISE bus itself does not buffer data. Data read from one protocol RX FIFO is immediately written to the destination TX FIFO. The destination is read from the first nine-bit entry in the stream read from the RX FIFO. This address is currently hard coded for the destination core (TX FIFO) but is supposed to be dynamically configured with a software running on a PC. This means that, to cause as few changes as possible, a new protocol needs to follow the FIFO-like interface and timing of the SPISE bus.
4.2 Selection of USB 1.1 implementation

Although the huge amount of different choices existing to implement USB 1.1 functionality only a few of these fulfilled the requirements. In section 4.2.1 will be listed the external hardware solutions considered with a single chip providing USB 1.1 or in some cases also 2.0 functionality. In section 4.2.2 the IP blocks will be listed and then will follow a short presentation of external solutions versus IP blocks in section 4.2.3.

4.2.1 Hardware solutions considered

The alternative of having an external hardware solution seemed quite attractive at first. The main advantage being the reduced logic usage in the FPGA. Attempts were made to find the most applicable chip for the design. External solutions however does not only provide benefits. The main drawbacks being cost, lifetime, larger printed circuit board (PCB) footprints, not meeting the project requirements, requiring separate chips for host/slave functionality or the need for external firmware. The solutions considered were

- Maxim 3420 (Slave only) [7]
- Maxim 3421 [8]
- *Cypress SL811HS [9]
- Cypress EZ USB Host [10]
- FTDI Vinculum II [12]
- ST Ericsson - ISP1160 (Host only) [13]
- ST Ericsson - ISP1760 (Host only) [14]
- **PLX OXU210HP
- Atmel AT90USB-series [15]
- ELAN USBHC869 (Full-speed only) [16]

* Not recommended for new designs
** Non disclosure agreement (NDA) contract required to get data sheets

The first round of elimination was to remove all options that were clearly at the end of their lifetime or provided only host or slave functionality but not both, as these choices would require dual chips. A special case was the chip by PLX where an NDA contract was required before even the data sheet was available. Due to limited time for the selection this choice was also eliminated.

The Cypress SL811HS was a major candidate until it was confirmed by Cypress that the errata dated before the data sheet was still valid for later data sheets, meaning a lot of functionality in the chip could not be guaranteed without risk of data corruption. It was at the same time confirmed not to be recommended for new designs.
The second round of elimination considered factors such as the number of pins required for controlling the chip, if firmware was required to be developed and to some extent even the estimated price of the choice.

If too many pins are required a complete redesign of existing hardware would be needed just to be able to test the design, as compared to being able to simulate an IP core. A complete PCB redesign before being able to test the solution would require more time than was available to finish the project and had to be avoided.

This is also the reason why options such as a move to an embedded Linux system, utilizing the existing USB drivers in the Linux kernel, was not considered for the implementation. The redesign of the previous work done to the SPISE box would consume too much time.

The chips that passed the second round are Cypress EZ USB Host (which despite the name also supports slave mode), FTDI Vinculum I & II and the AT90USB series of microcontrollers.

**AT90USB** a microcontroller with a built in USB controller. Even if it supports both host and slave mode it only supports one host/slave per chip. It would require at least two chips to provide host and slave functionality at the same time. It was therefore eliminated from the selection process.

**Cypress EZ USB Host** with the ability to provide two host ports while also providing a slave port at the same time is an attractive option. The pin count of 100 however is a clear disadvantage from the PCB footprint perspective.

**FTDI Vinculum I & II** with two independent ports individually configurable as host or slave in a 48-pin package (VNC2 is available as 32 and 64 pin package as well) was also a very attractive solution for the final selection and will be compared with IP cores in section 4.2.3.

### 4.2.2 USB IP blocks considered

The hardware already consist of an FPGA so intellectual property (IP) cores is a natural way to add USB functionality. Disadvantages with IP cores being the added logic in the FPGA and the fact that USB can not be driven by the output pins directly mean that a transceiver is needed, which means added hardware. The transceiver chip will however require a minimal amount of pins compared to moving all logic to an external component. The IP cores found were the following:

- OpenCores USBHostSlave
- Corepool FHG_USBEHC Host (Host only)
- Arasan USB 1.1 Host (Host only)
- Arasan USB 1.1 Device (Slave only)
- Hitech global USB 2.0 Device (Slave only)
- Actel CUSB Device (Slave only)
As for the external chip solution a first round was done to eliminate choices. This proved to be very easy. After contact with respective IP core provider, as the tenders were received. It proved that the only core fitting within the budget of the project was the OpenCores USBHostSlave. This is also the only IP core able to provide both host and slave mode within one core and so this is the only IP core to be considered for implementation.

4.2.3 Hardware solution versus IP block

From section 4.2.1, the chosen candidates are Cypress EZ Host and FTDI VNC I and II as external chips, from section 4.2.2 there is the USBHostSlave IP core.

The comparison was carried out while keeping in mind the existing hardware which is based on an FPGA and with a fixed number of connectors on the PCB.

From a developer perspective the external hardware solutions all provided the same benefits, the ability to implement the USB stack in C code for a microcontroller. C code would provide code maintainability, readability and shorter time to target and since the code would run inside the external chip less code would occupy space inside the FPGA, at the expense of another communication bus between FPGA and external USB stack. As the SPISE bus uses hard coded timing tied to the system clock this would require the SPISE box to either run synchronous to the external chip and have the communication emulate that of the SPISE bus or add more logic for synchronization.

As the microcontroller might be busy handling interrupts and unable to process signals as needed it was deemed not feasible to try the SPISE emulation, instead an extra layer of synchronization would be required. While being a fully valid solution it would add latency to the system and more logic in the FPGA. Another disadvantage of the external chips compared to an IP block solution was the need to setup a new development environment and the need to create a test PCB for the chip selected before being able to try the code out.

Implementing an IP core into the FPGA for USB functionality suffers the disadvantage that development in VHDL takes more time and is less flexible than development in software C code. No instruction set for handling stacks and moving data around is readily available and need to be developed if required. The IP core would also occupy a great amount of space in the FPGA.

The advantages however with the IP core is that the USB stack will be running completely inside the FPGA and if care is taken the complete system can run synchronously with only one clock. This means minimal logic needed to connect the USB subsystem to the SPISE bus. It also provides the possibility to simulate the full system in a computer before moving the design to hardware. This allows the design to be tried in simulation before a test PCB is designed.

4.2.4 Summary of selection

The final decision was made based on the observations about the different methods in section 4.2.3. The decision was made in collaboration with AAC based on their needs. Although the idea of developing a USB stack as software for a microcontroller seemed very attractive the overhead of creating a new PCB for this microcontroller and connecting it to the FPGA led to the choice of using
the USBHostSlave IP block. This way only a minor PCB for the transceiver was needed and it could be easily connected through a standard 10-pin header already on the same PCB where the FPGA is mounted.

4.3 Analyzing a USB implementation

Analog signals traveling the bus

The lowest level of analysis. Due to the NRZI encoding on the bus it is not feasible to manually decode the packets at this level. Valuable information at this level is however the rise and fall time of the bus. Although the values for resistors and capacitors for a PCB design is given [17] the board design still has to be verified as working so that errors due to wrong bus capacitance can be ruled out.

Digital signals traveling the bus

With a USB-analyzer it is possible to decode the NRZI packets from the analog bus. This is the lowest level analysis that still provides valuable information in regards to the design of the USB stack.

At this level everything traversing the bus is seen and represented as a list of USB-commands. Every part of the packet can be seen and errors due to faulty handshaking or DATA sequence can easily be spotted.

Abstracted digital level in the USB stack

It is possible to gain information about the traffic on the bus by utilizing a software USB-analyzer on a PC. Unfortunately at this level of abstraction it is impossible to gather all information needed for the evaluation.

Most notably the handshake and data sequence information is lost as this has already been dealt with by the USB-hardware and driver of the OS running on the PC.

Still a lot of information can be gained about the way the USB stack of the current OS implements the SETUP procedure. Software USB-analyzers exist both for Windows and Linux.

4.3.1 Technique used

As a hardware USB-analyzer was not available the second best option to extract information about USB-stacks was to utilize a software USB-analyzer combined with reading the USB 1.1 specification [17].

The extraction of logic USB-packets was done using the free VMware Player [18] with USB debugging turned on. The generated logs could then be analyzed with the Virtual USB Analyzer [19].

The hardware bus level information missing from these logs was then filled in manually by reading the related sections in the specification [17] thoroughly.
Chapter 5
Implementation

The task of implementing USB 1.1 functionality for both host and slave had to be divided into separate projects because the host is much more complex than the slave. Only a minimal amount of code could be reused between the host and slave.

Some basic building blocks were used in more than one core and will be described in section 5.2. The USB Slave will be considered as one block and described in section 5.5. The USB Host will be dealt with in section 5.6. As these blocks provide a very simple interface meant to be easily connected to other parts another block had to be developed to convert these interfaces to and from the internal bus of the SPISE box. The implementation of the bus converter will be described in section 5.7.

Since the USBHostSlave uses the Wishbone interface a description of Wishbone can be found in Appendix C. The I2C core did not have a bus interface and needed glue logic to be attached to the SPISE bus. The glue logic is not considered big enough to list in this chapter and can instead be found in Appendix A.

5.1 SPISE to Protocol block, interface overview

From a system view this implementation will look like in Figure 5.1. The protocol specific code will remain in the protocol block to keep a clean interface. What the converting logic does is to convert the SPISE box internal bus format of 9-bits to pure data of 8 bits width while going out from the SPISE box and the reverse while traffic is going in to the box.

5.2 Common building blocks

5.2.1 Falling edge detector

Using the falling_edge() function on a signal is usually reserved for clocks and could cause the synthesizer to interpret it as such resulting in wasting clock net resources in the FPGA. To avoid this a design as shown in Figure 5.2 is used.

Only when A is low and the registered value of A is still high can the output Z be high. This signals that a falling edge has occurred on A.
5.2.2 T-flipflop

A T-flipflop is built out of a D-flipflop. Designed as seen in Figure 5.3. This is necessary since the VersaTile technology in Actel ProASIC3 only supports the D-flipflop. MemBit is the output value of the T-flipflop and will be toggled every clock cycle while T is high.

<table>
<thead>
<tr>
<th>T</th>
<th>Q(=Z[n-1])</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.2: Truth table toggle memory (T-flipflop), signal Z
5.2.3 Asynchronous RESET synchronizer

Since the RESET signal is asynchronous to the system clock the release of the RESET can happen just before a clock edge. If this is the case the signal might not propagate properly to all parts of the system resulting in metastability problems.

To solve this problem the asynchronous RESET signal must be synchronized with the system clock. This is done using two flip-flops as in Figure 5.4. The first flipflop might end up with metastability problems but this will be blocked from the the rest of the circuit by the second flipflop.

The synchronizer design is fully described in [20, Section 16.5].

5.3 USB common

An overview of the USB protocol block with the FSMs common to both host and slave can be seen in Figure 5.5.

Even if the content of the host and will be different they both use a similar approach for the interface part of the USB stack implementation. Due to the Wishbone interface of the USBHostSlave core the ACK signal is used for synchronization of the FSMs by changing state of the Wishbone FSM whenever ACK is seen.

Due to the amount of information needed to be maintained there is more than one FSM. These will all ultimately be synchronized by the ACK signal as they are all in turn synchronized by the Wishbone FSM.

The state machines are introduced here and described more in depth for the host/slave specific states in their respective section.
5.4 Compliance with USB 1.1 standard

The USB blocks developed in no way试着 to claim full USB 1.1 compatibility. Instead the specifications followed are those for embedded hosts [21]. USB allows these deviations for embedded hosts to allow USB to be implemented even on limited hardware resources.

5.4.1 USB FSM

The USB FSM has the main responsibility to keep track of the state of the bus. As the USBHostSlave only provides the Host/Device-Controller (see Figure B.1) functionality the HCD and device USB stack is needed to keep track of the USB connection status. The USB FSM in that sense is the HCD in this implementation and it will be different for the host and slave implementation.

5.4.2 Wishbone FSM

The Wishbone FSM is the closest to the USBHostSlave core. The functionality of this FSM is shown in Figure 5.6. Used by both the host and the slave this abstracts the communication with the core from the top level FSM which means that top level states can be put together to perform a certain sequence of communication with the IP core without knowing about the Wishbone bus and the register addresses involved.

The Execute state will, based on the value of the counter and which USB FSM-state it is in, execute a specific command.

If the command interfaces with the USBHostSlave core the FSM will continue to the next command when ACK is seen on the Wishbone bus. If the Wishbone bus was not used in the present cycle the counter must be forced to increment by stepping it. This will happen if the action performed by the execute state is a parsing operation from which results are needed before further communication can be made over the Wishbone bus.

Most of the communication with the USBHostSlave core is done by reading and writing registers this will let the FSM decide when a value is available by
using the counter. If a register is read the data of the register is guaranteed to be available at the next value of the counter.

As many operations require a sequence of registers to be read and written this allows grouping together these read and writes in a consecutive manner. This grouping will be referred to as a *USB State* which is a state in the USB FSM.

![Wishbone FSM](image)

**Figure 5.6: Wishbone FSM**

### 5.4.3 FIFO FSM

As the cores connected to the SPISE bus are allowed to send fragmented packages some logic is needed to ensure that a USB packet is kept intact and stop it from being split by the SPISE bus controller. If the FIFO is connected straight to the SPISE bus (through the conversion block) the AVAIL flag of the FIFO goes high as soon as a single byte is written to it. When the SPISE bus controller sees this it will immediately read this byte and transmit it. Thus breaking the packet if it contained more than one byte.

In the same way if a FIFO is holding a packet already but the FULL flag is not asserted would result in the bus controller writing more data into it. Thus again breaking the packet by adding more data to it. This is why the FIFO FSM is needed and it has two tasks

- Block subsequent writes when a packet exist in the TX FIFO.
- Hide the AVAIL flag until complete packet exist in the RX FIFO.

**Write guard**

The write guard controls the FULL flag of the TX FIFO. The SPISE bus controller will check if the FULL flag is high for the FIFO or not before it starts a write. If the FIFO is full no write operation will be attempted. By intercepting the FULL signal from the FIFO and substituting it for a signal I\FULL which
is triggered by a falling edge on WE the SPISE bus controller can be made to see a full FIFO even if just half of the FIFO is in use. The I\textsubscript{FULL} signal is then again triggered low by a falling edge on the AVAIL signal from the FIFO.

The falling edge detector of Figure 5.2 is used to detect these scenarios on WE and AVAIL. By connecting the two falling edge connectors through an OR gate to the D-flipflop of Figure 5.3 the output will be triggered by falling edges on WE and AVAIL. As the I\textsubscript{FULL} flag tells the SPISE bus controller the FIFO is full no further write attempts will be made until the FIFO has been completely emptied and the falling edge on AVAIL triggers I\textsubscript{FULL} low again.

To catch the case were the FIFO fills up before a falling edge on WE was detected the real FULL signal of the FIFO is connected through an OR gate to the I\textsubscript{FULL} output of the FIFO FSM.

Read guard

The read guard intercepts the AVAIL flag of the RX FIFO and substitutes it for I\textsubscript{AVAIL}. The read guard is a bit more complicated as WE can not be used with a falling edge detector for the RX FIFO. This is due to the need to wait for the Wishbone ACK synchronization causing a need for WE to be pulsed whenever an ACK is received.

Instead the packet state is tracked in the USB FSM through a signal rxArmed. This signal is used for controlling whether the receiving logic of the USB FSM is armed or not. While receiving is allowed (armed) the FIFO is considered empty or half-full and the AVAIL of the FIFO is hidden from the SPISE bus as in Table 5.3.

<table>
<thead>
<tr>
<th>rxArmed</th>
<th>AVAIL</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.3: I\textsubscript{AVAIL}, as seen by the SPISE bus controller

5.5 USB Slave block

The USB slave is the more simple one of the USB modules implemented. It will only respond to external events such as a SETUP command or data arriving on either the USB wire or the SPISE bus. In this sense it is entirely event driven and the clock is only used to synchronize transfers between FIFOs.

5.5.1 USB 1.1 Slave functionality summary

- EP0 for SETUP using the smallest possible memory of eight bytes.
- Full speed mode (can choose low speed mode at compile time)
- Endpoints
– EP1 for IN transfers using a 64-byte FIFO
– EP2 for OUT transfers using a 64-byte FIFO
– FIFO sizes settable through generics in the USBHostSlave wrapper
– EP type (BULK, INTERRUPT) settable through descriptors

• Vendor ID and all other data part of the reply to the host GetDescriptor requests are fully configurable to fit the application through changing values in an array of constants and recompiling.

• Silently acknowledges unimplemented CONTROL commands (like STRING descriptors) with empty return data.

The USB Slave core is capable of being run in a simulation and hence is suitable for being used as a test bench. Such a test bench is very useful during the development of a USB host as no working and free USB test benches existed at the time of this writing. The USB slave FSM is described in further detail by pseudo code in Appendix E.

5.6 USB Host block

Because the USB slave is controlled by a USB host it follows that the host needs more controlling logic than the slave. The host will need to keep track of which state the slave device is in to maintain communication according to the USB 1.1 specification. This is accomplished by separating the state of the device into a major, command and minor state.

5.6.1 Device state tracking

A table is used to keep track of which state the connected device is. This table has four major states, four command states and four minor states. An index is used with next state logic based on a combination of the major and command state. The minor state keeps track of the sequence of a command sent and next state logic is not triggered until the minor state Handshake has been completed.

This way the low level IN, OUT, SETUP and Handshake generation is abstracted from the command state and handled by the minor states. The command state in turn is abstracted from the device connection state (major state) and only knows which is the next command to send.

With such abstraction the USB FSM only needs to know whether or not the device is in the Configured state or not.

Major states

Only symbolically assigned. Binary values assigned at synthesis by synthesizer. The states are S_RESET, S_CONNECTED, S_ADDRESSED, S_CONFIGURED and are shown in the context of a device setup in Table 5.6.
Table 5.4: Host device tracking. Command states

**Command states**

The command states describe what command is to be issued or being issued to the device. Only a subset of the USB 1.1 operations are supported. The states are manually assigned according to Table 5.4.

Each command state has a set of minor states tied to it. Whenever this command state is reached these minor states are executed in the order specified and will only change state when the handshake minor state has been completed or when an error happens, in which case the bus will be reset.

**Minor states**

Minor states are manually assigned according to Table 5.5. “IN Sent” and “DATA expect” are essentially the same. The IN state is only triggered immediately after the IN request was sent. Both however will expect data from the device and the state will transfer to “DATA expect” if the packet is fragmented and sent as smaller pieces. This will let the FSM skip some unnecessary parts if the command was not sent in the most recent FSM cycle. The handshake minor state completes the current command by issuing the correct handshake depending on the direction of the transfer.

Table 5.5: Host device tracking. Minor states

<table>
<thead>
<tr>
<th>State</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETUP Sent</td>
<td>00</td>
</tr>
<tr>
<td>IN Sent</td>
<td>01</td>
</tr>
<tr>
<td>DATA expect</td>
<td>10</td>
</tr>
<tr>
<td>Handshake</td>
<td>11</td>
</tr>
</tbody>
</table>

**USB device tracking states summary**

Shown in Table 5.6 is the command state sequences traversed for each major state. The minor states are not shown in this table to maintain readability. Command states do however specify minor states based on whether they expect data back from the device or not. As the minor states are not flexible and have to behave according to the USB 1.1 standard for each command state they can be derived from the information given in Appendix B.

5.6.2 USB 1.1 Host functionality summary

- Supports both FS and LS devices connecting
Table 5.6: USB device tracking states

<table>
<thead>
<tr>
<th>Major state</th>
<th>Command state</th>
<th>Next Command State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_RESET</td>
<td>-</td>
<td>MAJOR: S_CONNECTED</td>
</tr>
<tr>
<td>S_CONNECTED</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>MAJOR: S_ADDRESSED</td>
</tr>
<tr>
<td>S_ADDRESSED</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>MAJOR: S_CONFIGURED</td>
</tr>
<tr>
<td>S_CONFIGURED</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- Supports device FIFO sizes from 8 bytes and upwards
- Automatic SETUP of device on connect.
- Retransmission of packet when NAK received or if no ACK was received.
- Continuous polling of a connected device by IN transmissions as long as the device has data to deliver. The host will request a new IN transmission immediately after RX FIFO has been emptied of the previous packet. When no data is available the host will wait for a preset time before the next IN transmission attempt is made. This preset is configurable in the VHDL code.
- Correct DATA0/DATA1 handling with dropping of duplicate packets.
- Ignores device supplied transfer mode in EP descriptor. BULK and INTERRUPT transfers are treated the same way.
- Uses a static percentage (tunable in the VHDL code) of the frame before attempting no further transmissions before the SOF is generated at the 1ms mark.

The USB host FSM is described in further detail by pseudo code in Appendix F.
5.7 SPISE bus interface conversion block

As described in section 3.3 the exact bus timing was not easy to identify. Extracting the implementation from the VHDL-code in place was not a feasible option as the size of all the parts involved were too big. The implementations did not use a single core but was interweaved with the parts belonging to other communication protocols meaning there was a new implementation for every protocol currently connected. Instead the problem of the unknown timing was attacked with a workaround.

At the first read from a connected device RE is asserted for one cycle and then negated to read the BOM. This allows the implementation of the SPISE bus to parse the BOM and decide where the data should be sent. The second time RE is asserted the data is expected followed by an EOM.

This was exploited by creating an FSM triggered by falling edge on the RE signal as in Figure 5.7. The ReadState signal can now be used to determine if the bus is expecting a BOM or data.

![Figure 5.7: ReadState FSM](image)

5.7.1 Generate output based on timing

The signal ReadState is used together with an AND gate to control whether the RE signal will be delivered to the FIFO or not as shown in Figure 5.8. Also shown in the same figure is the needed logic to generate signals for implementing the FSM for the FIFO to SPISE bus converter.

![Figure 5.8: SPISE box conversion block logic signals](image)

The signals D2, D1 and D0 is the same value as ReadState but delayed 1, 2 and 3 cycles. The same is true for the data available signal (AVAIL) which is propagated to E2, E1 and E0 with the same delay scheme. This way a MUX
connected to the SPISE bus can decide what data to make available on the bus with simple AND gates without the need for counters.

ReadStream indicates whether a BOM or DATA stage is in effect. If DATA is expected D2, D1 and D0 is used to determine when the MUX selects FIFO output instead of providing a constant BOM. In the same way, E2, E1 and E0 is used to provide an EOM as the last data from the FIFO has been read.

5.7.2 Using the generated timing

When ReadState is low, RE will be blocked from reaching the FIFO by the AND gate (Figure 5.8) and the MUX will provide a constant BOM on the bus (this BOM has a settable destination through generics in the VHDL code). When the BOM has been read by the bus controller ReadState will be toggled (catches RE falling edge) while the controller parses the BOM. The next read will be passed on straight to the FIFO.

As the output from an Actel FIFO is delayed by two cycles the delayed values of the Avail signal are also required for the FSM to insert an EOM at the end of the data. When Avail goes low the FSM waits until the signal reaches E1 to insert the EOM. By waiting until the signal reaches E1 the final data byte from the FIFO has passed by the time the EOM is put on the bus. When the bus controller sees the EOM it will toggle RE low and not continue parsing data so that any value can again be put on the bus without being parsed.

5.7.3 Problems with internal tri-state

In the existing SPISE implementation the bus was set to tri-state after EOM by the protocol core being read from. As the FPGA does not support internal tri-state for precompiled cores and the conversion core should be used as a precompiled block, this was not possible.

The reason for externally compiling the conversion block is to avoid having to copy source code files to an existing project. A strong reason to not do this is to not have to update the same source file in several projects but just to make a modification in one source file, compile it, and then import that compiled block into other projects. The precompiled block should also save time during the compilation of the project instantiating the core in that block. By using precompiled blocks it is possible to provide a modular approach where protocols can be added and removed from the SPISE box easily by instantiating the already compiled core from the top level entity.

The problem is that the synthesizer does not support converting an externally precompiled block using tri-state into MUXes and the existing code is compiled together as a huge tree with the existing protocols having their source files in the same project hierarchy. When a tri-state is set within the same source tree compilation, the compiler is able to handle and work around the internal tri-state by converting it to MUXes.

The precompiled blocks can not use tri-state due to above mentioned fact that the technology of the current FPGA does not support internal tri-state (and that the synthesizer can not convert these precompiled blocks using tri-state into MUXes). This causes any attempts to use tri-state for a precompiled block to the compiler forcing the signals to be routed through physical pins on the FPGA.
While the converter block could just be copied into the source hierarchy of the SPISE box project to solve the problem with tri-state it would violate the modularity requirement to be able to easily equip the box with different buses and protocols depending on the current needs. With a modular approach the project cores can be synthesized separately and does not need to be re-synthesized until changes are made to them, thus saving time. With such an approach the precompiled blocks are only recompiled when a change to them is needed.

Solution to internal tri-states

The solution was to add a MUX in the bus controller. The MUX allows the SPISE box interface block to leave a BOM on the bus whenever no data is being sent and this solves the problem of the documentation of the bus not matching the code in place. By providing a BOM at all times when data or EOM is not expected all the conflicting information about timing could be worked around.

This temporary MUX is also allowed to use the false tri-state as it will be compiled in the same run as the rest of the SPISE bus. The behavior of this MUX is to be in tri-state until the externally compiled protocol core is selected as a source for reading data.

<table>
<thead>
<tr>
<th>Boolean expression</th>
<th>MUXed value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadState ((D0 + D1 + D2) + \overline{ReadState}E0E1E2)</td>
<td>BOM</td>
</tr>
<tr>
<td>(RE \cdot E1)</td>
<td>EOM</td>
</tr>
<tr>
<td>(D1 \cdot D2 \cdot (E0+E1+E2))</td>
<td>FIFO DATA</td>
</tr>
<tr>
<td>Else</td>
<td>BOM</td>
</tr>
</tbody>
</table>

Table 5.7: SPISE bus converter interface MUX rules

As seen in Table 5.7 an EOM is provided two cycles after AVAIL going low. As the data is two cycles delayed this is immediately after the last byte (Avail goes low the cycle after the last byte left the FIFO).

In the same way the MUX selects the FIFO output two cycles after Read-State was set high. Leaving the BOM selected the rest of the time.

5.7.4 Filtering data written to the protocol block

For the data being written into the core from the SPISE bus. The documentation for the SPISE box [5] mentions that the core has to filter out BOM and EOM. BOM and EOM are defined such that the most significant bit on the 9-bit bus is always high while these are sent and set low during data transmission. The filtering of BOM/EOM hence can be achieved through an AND gate intercepting the WE to the FIFO as in Figure 5.9. This will force the WE to the FIFO to only be activated for data bytes.
Figure 5.9: SPISE bus WE logic
Chapter 6

Hardware

The hardware used and designed will be presented here. Starting with the SPISE box in section 6.1 and the host and slave PCB in section 6.2.

6.1 SPISE box

The SPISE box is pictured in Figure 6.1 with both the USB host and slave PCB visible, mounted on the inside.

The heart of the SPISE box is the FPGA which provides the functionality required, i.e routing and inter-protocol connectivity. The current FPGA used is the Actel ProASIC3 A3P1000 for which all data can be found in [22]. The most important specification is the number of VersaTiles available as the cores implemented grew large in size. For the FPGA in use the number of VersaTiles is 24,576 and the FPGA is a 208-pin PQFP package.

As the time available was not enough for a complete re-design of the motherboard in the SPISE box. The choice was made to create separate boards for the testing of the USB subsystem. These are further described in section 6.2.

For this purpose the box serves as a platform for providing power and FPGA connectors through the pin headers. Other protocols previously used in the box had been using both 3.3V and 5.0V so the power was available inside the box already. These levels were extracted from the FPGA V_{CC} and the RS-485 power rails respectively.

Due to connectors on the front panel being connected to the motherboard by standard 10-pin headers it was not a problem to find IO-pins to use for the USB 1.1 PCBs.

The Opencores USBHostSlave IP core author himself recommends using the USB1T11A [23] transceiver by Fairchild. This transceiver has a 14-pin TSSOP-package but several pins can be left unconnected so the 10-pin headers are enough for the data signaling.

6.2 PCB Design

Taking advantage of a 10-pin header intended for a CAN connector it was possible to have eight pins for signaling and two for power (3.3V and GND). The CAN header was used for the slave PCB.
Figure 6.1: The SPISE box with mounted test-boards

As the host is also required to provide a 5V supply on the USB wires some workarounds had to be made for the host PCB as the best physical fit was on a miscellaneous connector (data pins only). The 3.3V and GND was taken by two wires from the slave PCB while the 5.0V was taken from a nearby RS485 connector. The specifics for each board will be dealt with in section 6.2.1 and section 6.2.2.

- To design the boards a suitable connector had to be found. Once a connector possible to use was found the physical space surrounding this connector had to be measured. These measurements were then used as constraints for the board sizes in Altium Designer. To reduce costs the boards were merged into one physical PCB and was then cut apart to form two separate boards.

- The boards use four layers. Signaling routes are drawn top/bottom layer and there are two internal planes for power and ground.

- To overcome the risk of short circuiting the internal power planes when separating the two boards these planes were separated into sub-planes and a marker at the surface shows the shape of the power planes (visible in Figure 6.2). That marker is the outline of both boards with a small margin to where the power and ground planes actually start.
  
  - Cutting outside this marker ensures that the edges of the power plane will not get in contact with metal objects in the box.
Having the margin also ensures that while cutting the PCB, the planes do not get damaged in a way that they can touch each other.

Figure 6.2: The test PCBs mounted in the SPISE box

6.2.1 Slave/Peripheral

Visible in Figure 6.2 (bottom) and Figure 6.3 (right) is the slave board. The transceiver requires a 3.3V supply which was available together with ground and eight signaling pins in one of the headers found (CAN connector). Signaling pins are required for

- Transceiver
  - Differential data OUT (2 pins)
  - Differential data IN (2 pins)
  - Output Enable (1 pin)
  - Speed select (1 pin)
  - VCC/GND (2 pins)

- Pull-up resistors (2 pins)

In total, 10 pins was used by the slave board. No values for resistors or capacitors had to be calculated as the values are given in [17]."
6.2.2 Host

Visible in Figure 6.2 (top) and Figure 6.3 (left) is the host PCB.

The required amount of pins are the same for the transceiver in section 6.2.1 with the difference being the pull-up resistors. Pull-up resistors are not needed for the host as they are only used by the slave to signal a connection event. A major difference is that the host need access to a 5V VBUS source to feed bus-powered USB devices and as the header in the SPISE box providing the best fit space-wise did not have either power or ground pins (miscellaneous header with 10 signaling pins) power had to be taken from another node in the box. The slave PCB was the best candidate as it is positioned close-by and allows the SPISE box motherboard to remain unmodified. By taking the power from the slave board (by wires) the host board does not need more than 6 signaling pins + 1 wire to a 5V source.

Visible in Figure 6.2 and Figure 6.3 are the wires soldered onto the slave PCB and connected to the host PCB at connector P3. Another point to note is the fact that 5V power was not available from the slave PCB. 5V is instead accessed from a nearby RS485 connector seen in Figure 6.2.

In Figure 6.3 another set of wires can be seen coming from the type-A USB connector soldering pads. These were used to attach logic analyzer/oscilloscope probes for verification of signal quality.

Figure 6.3: A view of the bottom layer of the test PCBs
Chapter 7

Verification

The main problem with the verification process was the lack of a USB protocol analyzer. This means that the data traveling the wire can not be seen. This lack of an analyzer was further complicated by the fact that only the FlashPRO3 programmer was supported for JTAG debugging while only a FlashPRO4 device was available.

These problems together cause the only way to verify signals inside the FPGA to be with blinking LEDs or to extract the signals through an external interface. To reduce the impact of these problems the following guidelines were followed during the continuous process of testing and verification.

• Every core was separately tested by itself with its own test bench before included in a higher level test.

• Every core was tested pre-synthesis to verify logic behavior

• Every core was tested post-synthesis to guarantee as far as simulation goes that timing constraints would be met.

• Every core was, when possible, tested with a separate hardware test bench designed for that core only.

• No more than one change to the core will be made for each place-and-route (PAR)

Although the lack of a USB protocol analyzer holds true for the major part of the project an analyzer was later acquired in time for the final testing.

7.1 I²C

The problem with verifying the I²C core was that since it supports several devices connected to the bus it would need to be controlled by the software developed by AAC. The software would however need to be ported due to being developed for an older version of MS Visual Studio no longer available.

1 One was actually made available (thank you Per Selin), during the last four weeks of the project and greatly improved the rate at which errors could be detected.
The task of porting this software falls outside the scope of this SPA-X conversion and so the SPA-1 (I\textsuperscript{2}C) core was not subject to testing as a part of the complete system. The I\textsuperscript{2}C core interface was tested separately with a hardware test bench sending and receiving data between three devices with hard coded addresses.

7.2 USB Slave

As the first protocol block to be implemented there was no way to give it a solid verification in simulation. Instead this can be seen as the first step towards a synthesizable test bench to be used for hardware and system testing at a later stage.

The functionality of a host was not available at the time of first testing of the USB Slave. This was solved by the design of a behavioral test bench acting as a host. The behavior of the test bench host was, for now, accepted as represented by what had been gathered from the output of software-only USB analyzers (as described in section 4.3.1).

Even if these analyzers ignore the real bus handshaking procedure of the SETUP sequence it still provides information of the general SETUP procedure and data transfers.

7.2.1 Topics covered in this test

This test was done in simulation only.

- USB Slave protocol block
  - USB FSM
  - FIFO FSM
  - Wishbone FSM

- USBHostSlave core (SLAVE mode).

- Receiving data from a host.

- Transmitting data to a host after an IN request.

- SETUP sequence(without proper handshakes) and replies to GetDescriptor commands from the host.

- Ability to accept an address from the host.

- Ability to accept a SetConfiguration command from the host.

The outcome is a synthesizable USB slave core which is verified to be responding correctly to commands sent by a host.
7.3 USB Host

The test setup is again entirely in simulation on a computer. The behavioral host test bench used during the testing of the USB slave core is replaced by a synthesizable implementation of a USB host core. For this to work a new test entity had to be developed to simulate the function of the USB PHY (described in section 7.3.2). As the USB wire is simulated the real bit by bit NRZI encoded traffic can also be seen/analyzed.

Intended goals of this test is to have a host and a slave communicating with each other over a simulated USB wire where any data uploaded to the host on the SPISE-side interface will be received by the slave and vice-versa.

7.3.1 Topics covered in this test

This test was done in simulation only.

- USB Host protocol block
  - USB FSM
  - FIFO FSM
  - Wishbone FSM
  - Device state tracking and assignment (section 5.6.1).
  - DISCONNECTED
  - CONNECTED
  - ADDRESSED
  - CONFIGURED

- USBHostSlave core (HOST mode).
- Simulated USB PHY.
- Detection of a USB Slave connecting to the bus.
- Receiving data from a slave.
- Transmitting data to a slave.
- Sending SETUP commands to a slave.

The outcome is a synthesizable USB host core which is verified with the previously verified Slave block.

7.3.2 Simulation of USB 1.1 PHY

To test the USB host and slave against each other a USB link had to be simulated. The PHY chip used for the hardware is the Fairchild USB1T11A. This chip has one differential pair INPUT and another differential pair OUTPUT for the FPGA interface. There is also a bidirectional differential pair which connects to the USB type A/B connector (the bus).

The minus and plus line of the differential pair each has the same solution and is shown in Table 7.1. The bus value is then read by the FPGA differential INPUT and fed internally to the USBHostSlave core.
This simulates a pull-up resistor by using weak high ‘H’ and weak low ‘L’ values and allows both ends of the bus to overpower the weak signal with a ’1’ or ’0’ when output enable is asserted low.

### Table 7.1: Simulated USB transceiver (D- and D+)

<table>
<thead>
<tr>
<th>USBHostSlave Output Enable Negated (OE)</th>
<th>D-PullUp</th>
<th>BUS value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>FPGA LINE OUT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>FPGA OUT</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>‘L’</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>‘H’</td>
</tr>
</tbody>
</table>

7.4 Host and Slave ping-pong

This part of the verification includes both the Slave and Host functionality and is the first hardware test-bench (HWTB) used. The test setup utilizes the SPISE-box hardware and two extra PCBs designed especially for the project (described further in section 6.2). Both host and slave is running inside the same FPGA but the signals are passed on to the separate PCB with USB transceivers. These transceivers are however looped from the host type A connector through a USB cable back to the slave type B connector.

This loop-back cable ensures that even if the host and slave are running within the same FPGA the signals between them are still passed over the transceivers and a physical cable.

A HWTB is used to insert a packet into the host TX FIFO. The same HWTB is continuously checking the AVAIL flag of the slave and once asserted high launches a read operation of the slave FIFO. The data is then verified for integrity by comparing the value from the FIFO just read with a constant array. If an inconsistency is detected a LED on the front of the SPISE-box is set to go off and the HWTB breaks execution.

If no error is detected the very same data is uploaded into the Slave TX FIFO and the same procedure for the AVAIL flag is repeated on the host side. This process is repeated infinitely until an error is detected in a package received. The longest test run was conducted for about five hours time and it should be noted that this represents a 100% bus utilization. Even when stressed to full transfer speed, for five hours, the LED to signal a fault did not light up.

#### 7.4.1 Topics covered in this test

- FIFO interface of the protocol blocks (to be linked with SPISE-bus)
- PCB
  - Board layout
  - Physical size
  - Signal quality
  - Soldering
• USB pipe (Host FIFO Write - PIPE - Slave FIFO Read), where PIPE is the USB subsystem and made invisible to the layers on top of it.

7.5 Single Slave to PC - SETUP and detection

As the device will live in the “0xFF” interface-class domain (vendor specified interface) there is not much to be done at the PC. What however could be verified is the detection and configuration of the device. As this process is the same for vendor specified interfaces.

It was also at the time of this test that the USB analyzer became available and the final handshaking of the USB protocol SETUP sequence could be corrected and verified in the protocol blocks.

The USB slave was connected to PCs running both Windows 7 and Linux OS at different times. In both cases the slave device was detected but as no vendor specific driver for this application was developed no OS knew how to handle the type of device connected. This is not a problem due to the fact that in the end the device will be connected to an on board computer (OBC) which will detect the device properly and know how to handle it.

A test program was developed for a Linux environment and will be described in section 7.7. With the slave device detected properly by a PC OS it was considered to be stable and the test passed.

7.6 Single Host to Slave device - SETUP and detection

With the device being detected and setup properly by a PC USB stack. The device complies enough with the USB standards to be used as a test-bench for the host block in hardware.

Just as for the slave the USB analyzer was available at this time and some minor handshaking errors could be located and quickly fixed. By the end of this test the host would be able to correctly detect, setup and handshake with a connecting device.

As the test is too complicated for blinking LEDs the verification was done through the USB analyzer and by connecting a series of USB devices available. These devices were keyboards, mice, USB memory sticks and the Applique Sensor Interface Module (ASIM).

As for the case of the PC. This embedded host does not have any idea how to communicate with a keyboard or mouse but the SETUP procedure is still the same and was verified to be working. Even though the host does not know how to talk to the device a data pipe is opened to the IN and OUT EP of the device. This data pipe can send raw data without the need to know what the overlying protocol looks like.

7.7 ASIM

As the bridge test in section 7.8 requires communication with the ASIM (Appendix D). The ASIM needs to be tested separately first to verify the functionality without
other factors involved.

For this purpose a stand-alone software is developed in C under the Linux OS to implement part of the ASIM protocol. Just enough parts of the protocol is implemented to verify the functionality of the ASIM.

7.7.1 Software for testing the ASIM

In this case the ASIM had previously been demonstrated as a temperature reader and the predefined messages for reading temperature feedback in either Fahrenheit or Celsius was still programmed. The program developed hence uses the known hard-coded values to subscribe to the temperature in Celsius. By using these hard-coded values there was no need to implement xTEDS in the verification software.

The output from the program is shown in Figure 7.1. The program continuously polls the Linux kernel to find out if a device with a vendor ID matching that of the ASIM has connected. If detected, initialization and reset commands are sent followed by subscribing to the message “temperature in Celsius every 1 second”. The ASIM provides the temperature on the IN EP every 1s and is read by the Linux USB stack which in turn forwards it to the program.

Not connected
Not connected
Not connected
Not connected
Not connected
Device found
Init
Reset
Subscribing MSG1

MSG(1): 26.000000 (Packets received : 5016)

Figure 7.1: Output from Linux program connecting to ASIM

The temperature is read as a FLOAT (C type) and is printed at the bottom line in Figure 7.1 together with the amount of packets read so far. The data traveling the USB wire for an ASIM temperature message is actually shown in Figure B.3 (page 63).

This software was written in C using the USB library [23] libUSB 1.0] and for the purpose of testing the ASIM over USB it provides enough functionality. The advantage of this approach is that no kernel driver needed to be developed for testing purposes and no kernel-space programming was needed.

The only ASIM protocol features implemented were

- Subscribe message
- Reset
- Initialize
- Receiving of messages (polling)
Out of these features only the subscribing and receiving of messages is of interest from the verification point of view (i.e. to visibly see the temperature readout change on the screen when the sensor is heated). Actually the temperature readout would function just as well without the reset and initialization commands but they were still implemented for completeness of the message reading sequence.

7.8 PC and ASIM bridge

Somewhat similar to the test in section 7.4 (Ping pong) but aims at a system level test of the complete USB subsystem. At this point the USB Slave and the ASIM has been verified separately both in hardware and simulation and need to be tested together.

The FPGA USB 1.1 host however has only been tested in simulation and needs to be verified. With the previous tests made the host is now the only unknown factor in this part of verification. With this step verified the USB 1.1 subsystem hence is verified both as a system and each component by itself. This leaves the only part left to be tested, one of the main goals of the thesis, the SPISE bus connection to the USB 1.1 (SPA-U) subsystem.

The visual output of this test is the same as for the ASIM alone. That means the only way to tell the tests apart is by checking where the cables connect to. A correctly performed test will give output in the Linux test-program as in Figure 7.1. Both the program and the Linux USB host is completely unaware that the data traffic is tunneled through an FPGA and another USB host and slave and this is the whole point of the bridge, to provide transparent USB forwarding.

7.8.1 Topics covered in this test

- USB Host
  - Detection and SETUP of a connecting device
  - Providing a simple FIFO-interface to the USB 1.1 layer (transmit/receive)
- USB bridging of Host and Slave inside the FPGA
- Cross-talk of PCB layout paths and signal integrity (see section 6.2 for more details).

7.9 SPISE box conv block

The final step in verification is the action of breaking the bridge in section 7.8 to attach the SPISE-bus in between the host and the slave.

Since the SPISE-bus exists entirely in the FPGA no new hardware setups are needed. The test is simply carried out by recompiling the current project with SPISE-bus support. This provides a module ready to be attached to the SPISE bus with modifications as mentioned in section 5.7.

With this final step of verification the project has been verified as successful. All subsystems up to the final system has been verified by themselves and together as a system.
Although the output is still the same as in Figure 7.1, the data traffic is now traveling through the entire system and arrives correctly at both ends. The final verification step includes all the parts in Figure 7.2 and it can be seen in this figure how each subsystem is divided through the dotted lines.

7.9.1 The verification step in more detail

A walk through of the test performed in Figure 7.2.

Starting at the Linux PC  A direct link is the only connection the software in the PC can see. This is the logical USB pipe that is mentioned in section B.1. From the software point of view there is no SPISE-box running between the PC and the ASIM.

USB Slave  Communicates with the PC and decodes the packets sent. Again, in the PCs point of view, this is the ASIM.

SPISE bus converter  Envelopes the data in a format suitable for the SPISE-bus. Routing destination, BOM and EOM are added transparently.

SPISE bus  Uses the destination information provided and forwards the data to the USB host through the SPISE-bus converter again. In this case it unwraps the data from the envelope used for the SPISE-bus.

USB host  Apart from detecting connections, setting up devices and maintaining frame policies on the bus, the host, in this (data transfer) case, will only transparently forward the data package to the ASIM.

ASIM  Receives data packet originating from the Linux PC. If it was the first package in the sequence it should contain information about which message to subscribe to. The ASIM then provides the information requested every X second (as specified for the message number in the xTEDS).

In the case of the verification of the whole system this is the temperature in Celsius provided as a float every 1s and is transmitted back the same path that the packet from the PC arrived at the ASIM.

The ASIM, just as the PC, will only see the host connected directly to it. From the ASIM point of view the ASIM is connected straight to the PC through the USB pipe.

7.9.2 Topics covered by this test

- SPISE-bus conversion block
- entire USB 1.1 (SPA-U) subsystem
7.10 Hardware

The hardware was mostly verified as a result of a successful connection being setup over the two external PCBs. To have an idea of the margins provided the boards connections were also analyzed by oscilloscope.

Seen in Figure 7.3 is the SYNC packet mentioned in section B.3 (page 63). This measurement was acquired by connecting the Host to the Slave PCB over a 1m USB cable loop. As a full analog analysis is not feasible in the amount of time available, the target of this measurement is to validate that D+ and D- have been connected properly, that they are synchronized and that an acceptable output can be detected. The rise and fall-time were both verified to be within the acceptable range of 4ns to 20ns and matched to within ±10%.

There is some ringing which can be seen on D+ which hints at a slight mismatch which could be compensated for in a future revision of the PCBs if required. The connection was however possible to be setup even over a 6m cable (1m + 5m extension). It should then be noted that USB defines the maximum cable length to be approximately 5m due to requirements in the electromagnetic domain. It is outside the scope of this report but for more information on the electrical signals and their requirements, they are fully described in [17, Chapter 7].

In Figure 7.3 what first looks as if the bus enters a J-state (see section B.2.2) from a J-state is still seen as the same differential level by the USB controller. The reason that this happens is that a charge enters the bus as OE is toggled and the bus leaves the idle state.
The first bit of transmission hence is the K bit (SOP) and is followed by JKJKJKK which forms the SYNC packet. Immediately following the SYNC packet is a PID which however can not be fully seen in this figure.

7.11 Logic verification

7.11.1 Bridged USB analyzer dump

In Figure 7.4 an example of the bridge functionality is shown. The two packets on top (arriving from address 86) are the packets coming from the USB Slave running on the FPGA to the Linux PC software. In the same way the three packets in the lower half are the packets going from the ASIM to the USB Host running on the FPGA.

It can be seen from this figure that the host on the FPGA polls the ASIM more frequently than the Linux PC polls the slave on the FPGA by a NAKed IN (packet 1405). The Linux software however is for testing purposes only and if a higher polling rate is needed it can be set in the source code and recompiled.

7.11.2 DATA toggle sequence error testing

To verify the correct behavior of the DATA sequence states a forced error was inserted, first in the host and then the slave.

In Figure 7.5 is shown a forced DATA sequence error in the host running on the FPGA. By locking out the toggling of the DATA sequence every other
packet is dropped and not forwarded properly to the Linux PC.

Figure 7.5: Bridged USB connection with forced DATA sequence error

The OUTDATA packets (packet 28 and 889) are not included in this test but are included to provide a nice frame to show the starting point of the test. The packets involved are packet number 892, 894 and 32. Packet 894 is forwarded correctly to the Linux PC (packet 32) due to DATA1 sequence being the only sequence accepted by the FPGA host at this time. Packet 892 however is dropped at the host while still being ACKed which is in accordance with how the USB 1.1 standard specifies how a DATA sequence error should be handled.

7.11.3 SETUP sequence

The SETUP sequence need to be verified both for the host and slave implementation and both must be functional to allow a bridge to work. In Figure 7.6 is the FPGA host setting up the ASIM and in Figure 7.7 the Linux PC sets
up the FPGA slave. *Observe that handshakes have been removed from the two figures to make the relevant SETUP data easier to present.* Handshakes are however being performed as the SETUP sequence does not proceed until a valid handshake ends the current request. For clarity one such handshake is included in Figure 7.8 and comes from the same sampled data as Figure 7.6.

There are some differences between the two implementations.

**Figure 7.6:** FPGA host SETUP sequence of the ASIM

Linux PC tries to read DEVICE QUALIFIER descriptors These descriptors are for high-speed capable (USB 2.0) devices and are not required to be supported by USB 1.1. The slave implementation solves this by acknowledging the request but not returning any descriptor. After three attempts the Linux PC host gives up and reads the CONFIGURATION descriptor instead.

**FIFO sizes** As the FPGA slave supports a 64-byte FIFO the Linux PC is able to read the complete DEVICE descriptor in one read. The ASIM however has the minimum allowed FIFO of eight bytes which forces the FPGA host to read all descriptors in blocks of eight bytes.
Though it depends on too many factors to be directly comparable the FPGA host has finished the SETUP sequence (first request to Set Configuration) after 25ms while the Linux PC needs 317ms. The Linux PC however does a lot more than just provide USB functionality so the time of the SETUP could vary. For the FPGA host however it should be noted that only one command attempt is tried per frame during SETUP (note the time steps of 1ms) and the sequence could be sped up by disabling this limit. The limit is enforced by default to not hammer the slave with requests as it is assumed that the connected slave will be running a slower type of controlling unit. In this case, even with the limit of commands per frame, the first attempt to read from the ASIM returns a NAK due to the ASIM implementation being.

<table>
<thead>
<tr>
<th>Layer: USB</th>
<th>PID</th>
<th>ADDR</th>
<th>EP</th>
<th>PID</th>
<th>INDATA</th>
<th>HS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet: 23 Time: 747,3828ms</td>
<td>IN  2 0 DATA1 000010</td>
<td>ACK</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Packet: 24 Time: 747,3828ms</td>
<td>OUT 2 0 DATA1 0 NAK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Packet: 25 Time: 749,3827ms</td>
<td>OUT 2 0 DATA1 0 ACK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
unable to deliver the requested data in that time.

**Packet skip for FPGA host** The observant reader might notice that for the FPGA host communication with the ASIM there are some occurrences where two packets are missing. This is again due to the ASIM not being ready to reply with a handshake by the first time the FPGA host tries to end the request. Instead a NAK is received and the FPGA host waits another frame before retrying another handshake, which then succeeds.

**Addressing of slaves** The Linux PC implements incremental addressing so the slave gets a new address with an incremented number every time it connects until the addressing scheme loops around the address space at 127. Here the system has not been rebooted for a while and hence the addressing is at 120. To save tiles in the FPGA the FPGA host always hands out the same address. In this case the FPGA host always sets connecting devices to have address 2.

### 7.12 Resulting IP cores

The resulting IP cores occupies a huge area in the current A3P1000 FPGA. For the SPA-U (host and slave) there is a 60% core usage (including the SPISE bus converter block).

The SPA-S block is given in [25] to occupy over 7.5% of core area for each instantiation (4x SpaceWire + switch core occupies 30%). The switch core will most probably not scale linearly but remain the same size even if just one SpaceWire port is used, the size of the switch core however is not known as it was only given in combination with four ports.

The SPA-1 (I^2C) block used about 5% but was not instantiated together with the actual SPISE bus arbiter due to the problem mentioned in section 7.1 so with the converting block it is expected to grow slightly. The SPISE converter block however does not occupy more than 0.1% of area.

It is also required to have the SPISE bus logic implemented to use these protocols. The arbiter itself does not occupy a lot of space (about 0.7%) but the required blocks for the LED controller and Ethernet up-link occupy together with the SPISE bus arbiter about 25% of the area.

<table>
<thead>
<tr>
<th>IP Core</th>
<th>Area Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPA-1</td>
<td>5%</td>
</tr>
<tr>
<td>SPA-U</td>
<td>60%</td>
</tr>
<tr>
<td>SPA-S</td>
<td>7.5-15%</td>
</tr>
<tr>
<td>SPISE logic</td>
<td>25%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>97.5-105%</strong></td>
</tr>
</tbody>
</table>

Table 7.2: FPGA area usage

Seen in Table 7.2 is the approximated area usage in the currently used FPGA instantiating one of every core. Uncertainty if the area of SPA-S can really be scaled linearly will most likely push this value above 100% since the switch core is likely to not shrink in size. Even at the lower level of 97.5% it is uncertain if the cores would all fit together due to issues with routing. The cores were never implemented all at the same time but only one at a time.
Chapter 8

Conclusions

Due to the limitations in time and budget several choices had to be made to cut out parts of the USB 1.1 functionality. While several functions for a full USB host has been left out, attempts were made to still fulfill the requirements for an embedded USB host. These requirements can be found in [21].

While time and resources did not allow a full 100% verification (especially the analog requirements) of the compliance with the requirements in this specification, they were used as guidelines for the project.

8.1 Results

With the IP blocks for I²C, USB 1.1 and SpaceWire now in place the SPISE box is ready for the SPA-X protocols at a logic level. Before the box is ready for shipping it will also require:

- Redesign of the box cover to be able to fit on the front panel the SPA-1, SPA-U and SPA-S connectors.
- Redesign of the motherboard
  - Transceivers need to be placed either on motherboard or on separate boards connected like the test-PCBs manufactured for testing USB 1.1.
- Porting the software required for controlling the SPISE box

8.1.1 I²C core

While the I²C core could not be part of a system level test it can still be expected that the core will work as a part of the system once the software is ported. The reason for this assumption is because the I²C glue logic provides a generic FIFO interface for which the SPISE bus conversion block will provide correct translation and that the I²C cores were tested separately and verified to work.
8.1.2 USB Core

Host

The USB Host is able to connect to the device intended for usage in the system (ASIM). The host is also able to connect to, and properly setup, low-speed keyboards, high-speed USB memory sticks and high-speed USB hubs. Other devices for testing was not available and the sub-protocols needed for using the keyboard, memory stick and hubs were not implemented. The SETUP in this sense only means the opening of a pipe for sending and receiving the sub-protocol commands.

Slave

The USB Slave was being SETUP by both a Windows and a Linux USB stack. Since the implementation used was vendor specific none of the Windows or Linux PCs was able to natively send or receive commands to the USB Slave but instead it had to be implemented as a program written in the C-language in the Linux environment.

8.1.3 USB Software

The only sub-protocol implemented for testing was that of the ASIM devices. Successful continuous reading of the temperature and subscription to messages from the ASIM concludes that the USB 1.1 implementation is working.

8.1.4 PCB design

The PCBs were tested as part of the other tests conducted. Most notably the ping-pong test in section 7.4 which stresses the hardware by sending/receiving data as soon as the USB system is ready so that there is no unnecessary waiting. Since no problems arised the PCBs did not need further hardware testing and were concluded as working for the intended operation.

8.1.5 SPISE bus conversion block

Due to the ambiguity of the SPISE bus implementation the intended correct timing could not be verified. The work-around implemented in the SPISE bus conversion block works for the protocol blocks implemented during this project but due to the level of uncertainty can not be guaranteed to work for all combinations of external blocks (possible future additions) and SPISE bus connections.

Minor changes to use a MUX instead of tri-state had to be done to the already existing SPISE bus controller core. The MUX was only implemented for testing of the USB cores and the other blocks still use the tri-state internal bus.
8.2 Future work

8.2.1 SPISE bus

For better modularity the internal bus of the SPISE box should be considered to be re-written using MUXes instead of an internal tri-state bus. Tri-state is not supported internally by the FPGA and so externally compiled blocks can not easily be added without making changes to the SPISE bus for every new protocol block added.

The SPISE bus should probably also receive a dedicated review/re-write to decide upon the timing of the bus and correction of the documentation.

8.2.2 USB Host

Handling more devices

Continued development to support multiple connected devices would make the core far more attractive for inclusion in other projects. While not needed for testing the inter-protocol routing it will reduce size needed in the FPGA if several devices need to be connected. The “hook-states” to insert the commands needed are already in place for the HUB support so what needs to be done is:

- Implement USB HUB protocol to communicate with the HUB. This is required to read devices beyond the first device connected on the bus.
- Increase size of internal device state tracking table in USB Host to the amount of devices that the host should be able to handle.
  - Dynamic tables is a possibility but requires logic for handling the entries in a RAM.

With changes to the IP core to allow for hub commands it would also be recommended to add a root-hub to the hardware to not require an external hub. Several chips and IP cores exist for building a USB hub but some research is recommended before an alternative is chosen to evaluate how each solution meets the required outcome.

In general an IP core will provide a more portable and less vendor dependent (product lifetime) solution than a hardware chip solution. On the other hand an IP core is a lower level solution than purchasing a ready made chip and most likely will require a substantially longer time to finish.

BUS usage

The current implementation uses the amount of ticks left in a frame and compares this value to the maximum possible read/write size on either end (based on FIFO sizes) to decide if a wait for SOF is needed or if the transmission can continue within the current frame. More efficient solutions are possible but requires more work.

- With knowledge about which protocol is being used adjust the frame time-check to apply for the maximum package size for this protocol. If the maximum package size fits before the end of the frame then send it.
• A more efficient solution is to implement splitting of packages to fully utilize the current frame. This requires blocking of the data available signal at the FIFO facing the SPISE bus to prevent a package being read in pieces by another device connected to the SPISE bus. When the full amount of data has been received in the following frame the data available flag should be toggled.

8.3 Summary

The time was not enough to re-design the motherboard and provide a full SPA-X solution with the correct connectors. The SPISE box was however prepared for SPA-X by implementing the protocols used and providing them as IP cores, ready for use in the FPGA. While the current A3P1000 FPGA could hold one each of the current protocol cores of SPA (if some less priority logic like the LED controller is left out) it is recommended to switch to a larger FPGA like the A3P3000 to fit more logic and more protocols. The A3P1000 would prove to be very tight if all the SPA protocol cores are added and anything else is required in the FPGA.

If however not all SPA protocols are required at the same time there exist no problem to remain with the current hardware.

8.3.1 SPA-1

It is believed that the SPA-1 core will work as the I²C core has been verified in other work by ÅAC. The added glue logic was verified in an internal hardware test bench running only on the FPGA without being linked to real SPA-1 connectors. It can therefore be said with enough confidence that the SPA-1 block will work.

8.3.2 SPA-U

In section 2.6 it was expected that a SPA-U implementation would be possible without requiring changes to the SPISE box internals, this proves to hold true. The fact that a MUX needed to be added stems from the use of internal tri-states by the SPISE bus controller. The USB 1.1 core could be added to the SPISE box code tree instead of being compiled as an external block to solve that problem. The SPA-U implementation is able to run without external control while maintaining a compatibility with the existing SPISE box hardware.

8.3.3 SPA-S

Since the Space Wire block was not modified and was already setup to work with the SPISE box before the SPA-X implementation started it has not been verified as SPA-S. It has however been verified as Space Wire in previous work done on the SPISE box by ÅAC. It can be assumed that SPA-S will work when hardware is added to support the SPA-S connectors on the motherboard as the IP core is the same as for Space Wire.
Bibliography


Appendix A

I²C

The I²C core in use is a modified version of one of the OpenCores I²C controllers [26]. The modifications previously made by AAC provides support for multi-master arbitration, clock stretching and interrupts.

As the I²C protocol is already available through this core the protocol itself will only be briefly introduced. While designing the interface for the IP core it was however necessary to reference the I²C data sheet [27]. This section will mostly use a high level description of the functionality of the I²C core and give a quick overview of I²C.

A.1 Short introduction to I²C

The I²C bus uses two bidirectional wires, the serial data line (SDA) and the serial clock line (SCL).

These lines are used for START/STOP signaling by a master to signal to listeners on the bus that their attention is needed.

- The START condition is when SCL is HIGH and SDA does a HIGH to LOW transition
- The STOP condition is when the SCL is HIGH and SDA does a LOW to HIGH transition
- The ACK condition is when the transmitter releases the SDA HIGH and the receiver pulls SDA LOW. The transmitter sees the LOW and knows the receiver is in sync.
- Clock stretching allows the SCL LOW to be extended by the receiver continuing to pull it low, extending the time available for the receiver to handle the data arriving.

The first seven bits written to the bus after a START is the address of the slave to access. Bit eight is used to signal the direction (read/write) of further data transmission. This totals a byte of eight bits which is the standard transfer unit on the I²C bus. After this byte is transferred the master will wait for an acknowledge from the slave device.

If the addressed device is connected to the bus it should now send an ACK to the masters request and a data transfer will take place in the direction specified.
by the handshake mentioned above. An ACK is performed after every byte sent until a STOP condition is seen.

A.2 ÅAC modified I²C controller

As the IP core handles the I²C protocol the task dealt with in this project was to handle the IP interface START/STOP and shuffle the data to/from FIFOs facing the SPISE bus. By using only the two FIFOs and no signals routed around these it provides a general interface for the SPISE bus to access.

A.3 Glue logic between FIFOs and I²C controller

Using the terms write FIFO (WF) (writing from SPISE bus to I²C bus) and the read FIFO (RF) (writing to SPISE bus from the I²C bus) the functionality (FSM) can be described at a high level language as:

- WF : AVAIL, initiate START and set direction. Continue feeding data bytes into I²C core until WF is empty. Further writes to the WF (by the SPISE bus) is blocked until WF is empty. The I²C controller can be started as soon as the AVAIL flag of WF goes high due to the much higher system clock than the I²C clock.
- WF : EMPTY, initiate STOP and allow further SPISE bus writes to WF.
- RF : AVAIL, seen by the SPISE bus and RF will be read at will by the SPISE bus controller.
- RF : EMPTY, allow further START/READ commands.

Due to the way the SPISE bus works (only initiate a read from a FIFO with data available) the first byte written to the WF is passed as is to the I²C controller core. This first byte is assumed to contain the address and directions bits. The direction bit is snooped by the glue logic between the WF and the I²C controller core to find out whether to expect data from the I²C bus or if to continue feeding data from the WF to the I²C controller.

If a READ direction was seen, the WF is expected to be empty after the first byte was read, otherwise the WF is RESET to ensure a faulty write by the SPISE bus does not further block the I²C bus. If a WRITE direction was seen the WF is emptied onto the I²C bus.

Parsing the address from the FIFO is not a limitation of the architecture because of the way that the SPISE bus is already designed to handle the Space Wire network protocol, which uses the first data entry for routing information. For the Space Wire this data entry is nine-bits which gives an effective SPISE bus (Space Wire) address of eight bits [25].

One limitation lies in the fact that the addresses of the devices on the I²C bus are not known to the I²C controller or the SPISE controller. If they were data could be polled for when the controller is left idle for a preset time period. In that way the RF could be filled for the SPISE bus controller to read without having to send a read command with an address. Instead the addresses of devices is expected to be known to a PC running software developed at AAC and the I²C bus is polled at request by this software.
Appendix B

USB 1.1

The USB 1.1 protocol is an open standard and the full specification is made available online at the official site for USB developers. Since the release of the USB 2.0 standard, the document describing USB 1.1 was updated to describe the new standard and the 1.1 specification has to be extracted from the combined 1.1/2.0 document [17].

As the USB protocol is fairly complex compared to other bus standards a brief description to all the parts involved in a typical USB system is provided. This description aims to introduce the USB 1.1 protocol as fast as possible rather than being a complete reference. Though being an overview of USB 1.1 some parts will be more detailed than others with the focus being on general understanding of USB and on parts which become important as descriptions about the implementation is made. The layout of this description will give an overview of USB at a systematic level in section B.1 which is a short summary of [17] Chapter 4 and Chapter 5.

As the parts of USB implemented involve a range of other chapters as well, these will be summarized in the following sections. First by a look at the electric level signaling in section B.2. This is a summary of [17] Chapter 7. Next follows a look at how the communication is carried out at the protocol level in section B.3. This will include material from [17] Chapter 4 and Chapter 8.

The procedure the bus goes through at a device connect or disconnect event will be dealt with in section B.4 and involves material from the device framework in [17] Chapter 9. This should provide a general overview of the USB 1.1 protocol enough to understand how the implementation in chapter 5 was done.

B.1 Systematic Overview

The bus is entirely host controlled by exactly one host. There can not be more than one host and a bus can not function properly without a host controlling it. A slave device will only respond to transactions initiated by the host and is not allowed to initiate any bus transfer by itself.

The host connects directly to exactly one device and this device can have any function described in the USB specification. A very common setup for a personal computer (PC) is to have the host connected to a hub to provide more
than one connector to the bus. The hub connected directly to the host is known as the root hub and can normally not be disconnected or removed.

USB 1.1 specifies two signaling rates. Low-speed (LS) which will give a maximum transfer rate of 1.5 Mbit/s and full-speed (FS) with a transfer rate of 12 Mbit/s. Later versions of USB are able to deliver higher throughput but this work focuses only on the 1.1 specification.

As a device is connected to the bus, the host will query the device for information regarding the intended use of the device. This information is given by the device every time it is connected. Once the host has this information it will signal the overlying operating system (OS) about the device capabilities and the OS can then decide on which software drivers need to be loaded to support the connected device. This means that USB is a transport layer which supports plug and play (PnP) and in itself does not describe the protocol to control the device functionality.

USB does however define a set of classes which have predefined protocols for communication to allow a class of devices to be used in the same way independent of manufacturer as long as the manufacturer implements the right class and protocol. As an example, the common USB memory stick is part of the “USB mass storage” class. The protocol for accessing the data stored in a memory stick is well defined in a separate specification for this class. The currently existing classes are all defined in [17].

The USB system can be described in layers (Figure B.1). These layers are on the host side, the host controller (HC), host controller driver (HCD) and the client software. The HC is the hardware responsible for driving the signals on the bus and is in turn controlled by the HCD which is the system software to implement the transfer protocol. The HCD is typically a driver in the OS which then provides access to the client software through an application programming interface (API). The client software in turn could be the file browser used to browse the files on a USB stick. The design of these subsystem will let the client software talk “directly” to the function running in the device. It does so over the pipes created by the HCD for tunneling data over the physical USB wires to the USB stack on the device.

A similar layout is used on the device end of the bus but in a much simpler design due to the fact that devices are usually intended to run on limited hardware. The implementation of the device is up to the manufacturer to decide upon as long as the device is able to communicate over the bus according to the defined protocols.

What these layers suggest is that the underlying layers are transparent to the layer on top of it. The client software does not know how to control the HC and will only see the functionality provided by the device (the file system in the case of a USB mass storage device). This also means that the USB hardware does not need to know about the data travelling the bus as it belongs to a protocol layer abstracted on top of it.

B.2 Electric overview

At the electric level USB uses four wires, two wires for power and two for data signaling. The power wires are VBUS and GND. VBUS is a 5V DC source and is used to power connected devices which do not have their own power supply.
The signaling is done over D+ and D- which is a differential data pair utilizing Non Return to Zero Invert (NRZI) encoding to include clock synchronization. The polarity of the differential pair depends on the speed being utilized. This causes the need to introduce J and K states instead of defining bit levels for FS and LS modes separately.

As given in Figure B.2 (type A connectors) and Figure B.3 (type B connectors) the pin numbering of the USB signals are

<table>
<thead>
<tr>
<th>Pin #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VBUS (V&lt;sub&gt;CC&lt;/sub&gt;)</td>
</tr>
<tr>
<td>2</td>
<td>D-</td>
</tr>
<tr>
<td>3</td>
<td>D+</td>
</tr>
<tr>
<td>4</td>
<td>GND</td>
</tr>
</tbody>
</table>

Table B.1: USB pin numbers for figures Figure B.2 and Figure B.3

Figure B.2: USB Type A connectors seen from the front

B.2.1 Cable length

The USB specification does not give a definite maximum cable length in meters but it does specify requirements in propagation delay and attenuation. The one-way propagation delay is allowed to be 30ns over drivers and cable and the maximum allowed delay on the cable alone is 26ns.
Figure B.3: USB Type B connectors seen from the front

As the exact signal propagation speed for a certain cable is not exactly known a wide-spread accepted maximum length has been given as 5m (which is specified at the official USB developers forum). Other unofficial numbers seen varies from 3-5m and very likely they are all correct for a real world scenario where cheaper USB cables might not live up to the requirements. If a cable follows the specification however, 5m should be very much possible.

B.2.2 Bus states

The differential line is used with different polarity depending on whether the line speed is FS or LS. To simplify references to these levels the states J and K are introduced and from here on the J and K states will be used instead of specifying values of the differential line.

A differential “1” is defined as D+ having a voltage 200mV or greater than D- while at the same time satisfying D+ having an absolute voltage of at least 2V. In the same way a differential “0” is defined as D- having a voltage at least 200mV greater than D+ while at the same time satisfying D- having an absolute voltage above 2V. For the Idle state the USB specification states that in LS-mode, $D^- > 2.7V$ and $D^+ < 0.8V$ and for FS-mode the idle state means that $D^+ > 2.7V$ and $D^- < 0.8$. This can be simplified to define Idle state as a J state which is true since the analog levels are handled by the transceiver which is fed by digital inputs, and this report deals in the digital domain. The exact voltage requirements for every state can be found in [17, Chapter 7] but for the remainder of this introduction to USB a simplified high or low will be used to describe the level. This is illustrated in Table B.2 together with the definition for the other states possible on the bus. A properly functional bus should never allow both data lines to go high at the same time, that bus state is called SE1 and is illegal in USB.

B.2.3 NRZI encoding

NRZI encodes a “0” as a change in logical level out and a “1” as no change. Long strings of “1”’s hence gives no change in the data traveling across the bus and the communication might lose synchronization. To overcome this problem bit stuffing is used. Bit stuffing happens when six bits with no change in logic level (6 “1”’s in a row) occur. A “0” is inserted to ensure clock synchronization will be maintained. This is the task of the Serial Interface Engine (SIE) to insert and remove these extraneous bits as needed to maintain synchronization over the bus. The SIE is also responsible for encoding/decoding the J&K-states depending on the bus speed and polarity.
Table B.2: bus states

<table>
<thead>
<tr>
<th>State</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Differential “0”</td>
<td>D+ low, D- high</td>
</tr>
<tr>
<td>Differential “1”</td>
<td>D+ high, D- low</td>
</tr>
<tr>
<td>Single Ended Zero (SE0)</td>
<td>D+ low, D- low</td>
</tr>
<tr>
<td>Single Ended One (SE1)</td>
<td>D+ high, D- high (Illegal state)</td>
</tr>
<tr>
<td>FS J</td>
<td>Differential “1”</td>
</tr>
<tr>
<td>FS K</td>
<td>Differential “0”</td>
</tr>
<tr>
<td>LS J</td>
<td>Differential “0”</td>
</tr>
<tr>
<td>LS K</td>
<td>Differential “1”</td>
</tr>
<tr>
<td>Idle</td>
<td>J state</td>
</tr>
<tr>
<td>End Of Packet (EOP)</td>
<td>SE0 for 2 bit times followed by J for 1 bit time</td>
</tr>
<tr>
<td>Start of Packet (SOP)</td>
<td>Switch from Idle to K state</td>
</tr>
<tr>
<td>Connect (at downstream port)</td>
<td>Idle for ( \geq 2 \text{ms} )</td>
</tr>
<tr>
<td>Disconnect (at downstream port)</td>
<td>SE0 for ( \geq 2.5\mu s )</td>
</tr>
<tr>
<td>Reset</td>
<td>SE0 for ( \geq 10\text{ms} )</td>
</tr>
</tbody>
</table>

B.3 Protocol Overview

To help explain the protocol consider the packet in Figure B.4 throughout this section. The USB protocol can have up to 127 devices connected to one host. A device is given a unique address at connection to the bus. This is described further in section B.4. The packet contains an address field which routes a packet to the correct device and the data is then delivered to an endpoint (EP) at the device.

An EP is a destination/source for data and each device can have up to 16 EP. Every device is required to have an EP listening to number 0 as this is where all the control and configuration of the device is sent. From here on a format of “ADDRESS.EP” will be used to describe which is the address and EP of a device.

![Figure B.4: USBee DX capture of host sending a DATA0 packet to slave](image)

B.3.1 SYNC

Not shown in Figure B.4 is the SYNC field. The Start of Packet (SOP) is part of the SYNC and for USB 1.1 the SYNC is defined for FS and LS as an eight bit state transition from Idle to KJKJKJKK (see Table B.2). Where the first K bit is the SOP. The SYNC field is used only for synchronization and will not be shown in decoded packets from the bus, they will be left out from any further description of packet formats, but it should be understood that the SYNC is always there for every packet sent.
B.3.2 Packet Identifier field

Packet Identifier field (PID) is the first field to follow a SYNC. This field describes the type of package and is eight bits long. The first four bits define the PID type and the remaining four is the ones complement of the first four bits. This is to ensure the correct decoding of a packet and the check if the complemented bits match the PID is performed for every packet sent. Any mismatch between the complement and not complemented part of the PID generate a PID error and means that the packet should be discarded. Further references to the PID will only mention them by name and will leave out the bit representation of them. The PID field however is always eight bits. In Figure B.4 the first PID is called the TOKEN and will be one of

- OUT
- IN
- SETUP
- Start of frame (SOF)

Following a SOF token is expected a frame number which is 11 bits long. This is to identify the start of a new 1ms frame on the bus. Each frame has a unique ID and are used for synchronization purposes and scheduling on the bus. Following the OUT, IN and SETUP tokens is expected an address and EP. The SETUP is used when communicating with EP zero on a device and is further described in section B.4. The IN and OUT tokens are both used for data but specifies different directions on the bus. An OUT transfer is always the host sending a packet to a device while an IN transfer is always the host requesting data from the slave.

The next PID in Figure B.4 is the Data PID. The possible values here are

- DATA0
- DATA1
- DATA2
- MDATA

Only DATA0 and DATA1 are relevant to USB 1.1. One of DATA0 and DATA1 is set on every transfer involving data transferred on the bus. It is a way of synchronizing the transfers between host and device. If the receiver expects a DATA0 but receives a DATA1 transfer it will be considered a duplicate resent by the sender. This happens if the sender fails to see the handshake of the first transfer and therefore retries the transmission. When the handshake is done the value of the data PID is toggled and the next transfer a DATA0 will be received.

Following the data PID is the data to be transferred finished by a handshake ACK if the data was successfully received by the receiver. The possible values for handshake (HS) are

- ACK
- NAK
• STALL
• NYET

Again for USB 1.1 only ACK, NAK and STALL are relevant. ACK means the packet was successfully received free from error by the receiver. NAK means the receiver temporarily was unable to accept the packet. Causes to receiving a NAK could be that the receiver is temporarily out of memory to process an incoming packet or in the case of an IN request from the host it means the device did not have any new data to send. STALL means an EP has stalled (had an error) and need to be restarted or for a control EP that a control request was not supported.

Other PID values than the ones listed in this section will not be used in the scope of this report. The complete list of PID values (including the ones not used in USB 1.1) is given in \[17, Table 8.1\].

B.3.3 Endpoints

An endpoint is the interface between the bus and the device internals. A device could use one endpoint for high priority but small data amounts where low latency is desired and another EP where a high throughput is more important than when delivery is made. To distinguish between different transfer methods there are four types of transfers defined. Each EP specifies one of these modes together with an IN or OUT direction. Only one direction per EP is supported but EP zero is an exemption from this as EP0 is defined for both IN and OUT directions. The four transfer types are

• CONTROL
• BULK
• INTERRUPT
• ISOCHRONOUS

**Control** is used during the connection phase and for later status and control of a device. Setup is discussed further in [section B.4](#).

**Isochronous** transfers are a special kind of transfers with the key priority being the delivery of data on scheduled time. However if data could not be delivered on time no retry is made and the package is dropped. Isochronous is therefore only suitable for time-sensitive data where data loss is of less importance.

**Interrupt** transfers prioritize periodic transmissions on a specified interval with low amounts of data. An interrupt EP specifies an interval at which it want the host to poll it for data and the host should guarantee at least to check the device for data this often.
Bulk transfers are for low priority transfers of bulky (large) amounts of data. Bulk and interrupt transfers in that way are essentially the same while they are utilizing the bus but scheduled differently by the host.

It is specified in [17] that no more than 90% of a frame should be allocated for periodic traffic. The remaining 10% is dedicated to control transfers. If there are no control transfers or there is some time left in a frame after they are done the remaining time is scheduled for bulk transfers. In the same way if there were no periodic transfers bulk transfers get the left over time from the interrupt transfers as well.

B.4 Setup Overview

As the host detects that a device has connected to the bus it will start a process to inquire about the capabilities of the device. The typical sequence of events that take place during this phase is that the device asserts either D+ or D- line high depending on the speed with which the device wishes to connect to the bus.

When D+ is pulled high it signals to the host that a FS device is connecting. Consequently when D- is pulled high it signals that a LS device is connecting. This means that the device connects by sending the bus from a Disconnect state into a J (Idle) state (Table B.2).

The host, after detecting the J state on the bus, starts the process of querying the device for the information needed to use this device. The steps gone through for a standard PC USB host are shown in Table B.3.

<table>
<thead>
<tr>
<th>Device State</th>
<th>Host Command sent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disconnected</td>
<td>None</td>
<td>Waiting for connection</td>
</tr>
<tr>
<td>Connected</td>
<td>GetDescriptor(Device)</td>
<td>First 64 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Assign address</td>
</tr>
<tr>
<td></td>
<td>setAddress()</td>
<td></td>
</tr>
<tr>
<td>Addressed</td>
<td>GetDescriptor(Device)</td>
<td>Full descriptor</td>
</tr>
<tr>
<td></td>
<td>GetDescriptor(Config)</td>
<td>First 9 bytes</td>
</tr>
<tr>
<td></td>
<td>GetDescriptor(Config)</td>
<td>Full descriptor</td>
</tr>
<tr>
<td></td>
<td>GetDescriptor(String)</td>
<td>Get strings</td>
</tr>
<tr>
<td></td>
<td>SetConfiguration()</td>
<td>Select device configuration</td>
</tr>
<tr>
<td>Configured</td>
<td>IN/OUT Data</td>
<td>Normal operations</td>
</tr>
</tbody>
</table>

Table B.3: Setup procedure

B.4.1 Connected

At connection a device is required to reply to transmissions directed at address 0 until it has been assigned an address by the host, after which the device is no longer allowed to reply to address 0.

The device is asked to send the first 64 bytes of the Device descriptor and the device replies with a minimum of 8 or more bytes to this reply. The full listing

1 Ubuntu Linux 10.04 - 2.6.32 kernel
2 This data was empirically gathered as described in section 4.3
of descriptor types can be found in [17, Section 9.6]. Only the most relevant fields (i.e. the ones seen in Table B.3) for the setup procedure of the descriptors will be explained here.

The first 8 bytes of the device descriptor contains the maximum packet size for EP zero. With this the host can adjust any data sent during the setup phase to be in packets fitting within this limit. This ensures that the host can continue the setup procedure without sending packages which would be too large for the device to handle. Eight bytes however is the minimum amount a device will have to be able to handle as the setup commands are always eight bytes [17, Section 5.5.3].

When this information is read and parsed by the host, the host assigns a unique address to the device for subsequent transfers. The setup state is now moved to the Addressed state and address 0 is again available for another device to connect.

B.4.2 Addressed

From the previous stage the full size of the device descriptor is known. The host now reads the full size of the device descriptor for further parsing as the full size of the descriptor was also included in the first eight bytes.

Next the first nine bytes of the configuration descriptor is read. The configuration descriptor contains information about the number of interfaces and configurations, the protocol to be used and which type of EP the device support. When the configuration descriptor is read the device will return several descriptors. The first descriptor is the configuration descriptor, second is the interface descriptor and after this follows the EP descriptors. This is why only the first nine bytes were read at the first read. These nine bytes contain information about how many bytes all the descriptors to be received occupies together and the host can prepare to receive this amount of data in the next read.

While parsing these descriptors the host makes a decision on how to setup the device. The various interfaces and configurations supported by the USB protocol can let a host configure a device in different ways depending on whether the device is bus powered or supplied with power through external power supply. This lets the host choose a configuration that will use less power while the device is drawing power from the bus and another if an external power is supplied. This behavior however was not supported by any devices encountered during the testing and it will be assumed in the scope of the SPA-X implementation that the device has only one configuration and one interface.

In the last phase, string descriptors will be received, or rather indexes to them. These strings are vendor supplied and encoded in the device firmware. The strings describe in a human readable format information about the device such as vendor and product name. String descriptors are not essential for USB functionality and are optional to support by devices. Support for the GetDescriptor(String) command however is required by the device even if no strings are available.

The host finishes the Addressed state by selecting a configuration to use for the device and the setup procedure is then moved into the Configured state.
B.4.3 Configured

It is not until this stage is reached that the device is able to be used in the intended way (i.e. the functionality of the device becomes available). In the case of an USB memory stick this is the stage where the data on the stick is available to the user. Data can now be transferred using one of the transfers mentioned in section B.3.3.
Appendix C

Wishbone

The Wishbone bus \[28\] is an interconnect bus for Intellectual Property (IP) blocks and is not intended to be used over wires and Wishbone specifies no electrical requirements. The Wishbone bus is completely free to use and aims to increase the reusability of System-on-Chip components by providing a well defined interface. In the scope of this thesis only the read (section C.1) and the write (section C.2) parts of the specification will need introduction.

In the context of the Wishbone bus a master is the core controlling the STROBE signal. The slave is the core who responds to the request with an ACK. Not shown in the Figure C.1 and Figure C.2 is the address and width of the data bus. It should however be kept in mind that read and write operations operate with an addresses input. The data bus is defined to be 8, 16, 32 or 64 bits wide and the address bus between 0 and 64 bits wide.

C.1 Wishbone read

When a Wishbone read (Figure C.1) is performed the signal STROBE is asserted by the master and when the ACK is asserted by the slave the data is available to the master. It can be noted that ACK is allowed to be asserted combinatorially and not synchronous with the clock. The data however need to be synchronized with the use of a register.

The slave can use the ACK to throttle data by inserting wait states from receiving the read request until it delivers the data (ACK signal high). No assumptions on the amount of clock cycles from when STROBE is asserted until data is available can be made so the ACK signal must be monitored at all times.

C.2 Wishbone write

The Wishbone write sequence (Figure C.2) is very similar to the read sequence from a timing perspective.

Data is made available to the slave and a write enable (WE) signal is asserted. The slave can then choose to acknowledge the data immediately if it has the capacity or to wait with asserting the ACK until the data can be handled. This way the slave can throttle the speed at which data is written.
When ACK is asserted the data is written in the sense that it has been latched by the slave. The master that issued the write will assume the data is written and assert WE low again. Data can not be assumed to have been written until the ACK has been registered.

When the ACK is seen the Wishbone master signals WE low and the Wishbone slave will then pull ACK low. At this time the bus is ready for another transmission.
Appendix D

ASIM protocol

The Applique Sensor Interface Module (ASIM) is connected to a sensor of some sort to abstract the sensor from the on board computer (OBC) in a satellite. This way the satellite data model (SDM) software need to know only how to talk to an ASIM. And the ASIM can be made to have a standardized interface that does not change.

With this solution it does not matter what manufacturer of a sensor is used. If the hardware protocol between two sensors is different and a change between them is called for. It is the problem of the ASIM to read the new sensor by implementing the new protocol and buses that is required. The sensor data is then presented in the standardized format to the SDM. So the hardware connected to the ASIM never need to be changed.

The protocol was extracted from training materials provided in the course on the CubeFlow satellite systems [29] given by AAC.

Due to the ASIM protocol only being used for verification it will not be dealt with in great detail. The ASIM protocol travels on top of the USB 1.1 bus and as such is a higher level protocol not covered by the scope of this report.

The ASIM protocol is structured in the way that the first byte (8 bits) define a command. The only relevant ones used are

- **M** - subscribe to a message
- **I** - initialize
- **R** - reset

The important command here is M, which is used in section 7.7.1 to verify the design at system level. This command takes as argument (byte 2 and 3) the length of data payload and the message to subscribe to (byte 4 and 5). When subscribing to a message the bytes above index 5 where unused but reserved. See dumped USB packet in Figure B.4.
Appendix E

USB Slave state machine (pseudo description)

Default state after the RESET button was pushed and released:

- Wait until USBHostSlave core has been fully reset

Next State:
- S_HARD: Always

When no operation is being performed the FSM remains in this state. The only way to exit the idle state is by an interrupt to occur.

- Update address in UHS core if an address change is pending and the final ACK handshake has been delivered in the SETUP sequence.

Next State:
- S_IRQ_Idle: USBHostSlave interrupt.
- S_WE: SPISE bus write to the IN FIFO.
- S_RE: SPISE bus read from the OUT FIFO.

If a SPISE bus read is triggered this state waits until the read finished.

- Update signals to let the USB FSM know the SPISE bus FIFO is again empty.

Next State:
- S_IRQ_Idle: RE goes low
Waits for a write operation to the IN FIFO to finish.

- Update signals to let the USB FSM know data was written into the SPISE bus FIFO.

**Next State:**

- **S_IRQ_Idle**: WE goes low.

**S_HARD**

Events to do after a hard reset i.e when the reset button was pushed or if the USB FSM wants to trigger a complete reset of the core.

- Reset UHS and wait the 10 clock cycles specified for the USBHostSlave to finish the operation.
- Setup UHS in Slave mode
- Enable EP0 and EP2
- Force USB address register to 0
- Enable FS connection (pull-up resistors enabled for D+)
- Set UHS interrupt mask to enable all interrupts

**Next State:**

- **S_SOFT**: Always

**S_SOFT**

Triggered after a soft or a hard reset. Intended as a mean of implementing a reset of the USB without actually resetting the complete core (which also destroys any data left in the FIFOs). By moving tasks from the S_HARD state they will still be executed at a hard reset. But can also be triggered as a soft reset only.

**Next State:**

- **S_INITDONE**: Always

**S_INITDONE**

- Initialize logic to know which FIFOs contain valid data

**Next State:**

- **S_IDLE**: Always
Entry state when an interrupt is detected from the USBHostSlave or a SPISE bus read/write operation. Reads the interrupt status register and parses it.

Next State:
- **S_XFER_Idle**: Triggered by SPISE bus read/write
- **S_INT_XFERDONE**: Transfer done interrupt
- **S_INT_RESUME**: Resume interrupt
- **S_INT_RESET**: Reset interrupt
- **S_INT_SOFRECV**: SOF received interrupt
- **S_INT_NAKSENT**: NAK sent interrupt
- **S_INT_VBUS**: VBUS detected interrupt
- **S_IRQDONE**: Unsupported interrupt

Determine if interrupt was caused by a control or data transfer when an OUT transmission completed. If no data is available in the OUT FIFOs then the interrupt was caused by a completed IN transmission.

Next State:
- Reset interrupt status register

Next State:
- **S_CTRL_Idle**: SETUP packet detected
- **S_XFER_Idle**: DATA packet detected
- **S_XFER_FINISH**: End of IN transmission.

RESET signal detected on the bus.
- Force UHS address to 0
- Reset interrupt status register

Next State:
- **S_IRQDONE**: Always
Resume state detected on the bus. As suspend states are not implemented this state only acknowledges the interrupt and returns.

- Reset interrupt status register

Next State:
- \textbf{S_IRQDONE} : Always

\underline{S_INT_SOFRECV}

SOF received. Device knows 1ms has passed since last SOF was received when a SOF is received. The Slave however is entirely interrupt driven by bus actions. No timing information is needed.

- Reset interrupt status register

Next State:
- \textbf{S_IRQDONE} : Always

\underline{S_INT_NAKSENT}

A NAK was sent to the host. As the SPISE bus controller does not take input from a connected module this state does nothing.

- Reset interrupt status register

Next State:
- \textbf{S_IRQDONE} : Always

\underline{S_INT_VBUS}

VBUS detected. As the signal is not connected to hardware this interrupt is not implemented.

- Reset interrupt status register

Next State:
- \textbf{S_IRQDONE} : Always

\underline{S_IRQDONE}

Collect state for the end of interrupts.

Next State:
- \textbf{S_Idle} : Always
**S_CTRL_Idle**

Entry state for SETUP transfers.

Next State:
- **S_ReadCount**: Always

**S_CHECK_EP0**

An error detected during SETUP state. SETUP packet was not 8 bytes.

Next State:
- **S_CTRL_FINISH**: Always

**S_ReadCount**

Read the data count in the OUT FIFO for EP0. And since the byte count of a packet is always read at the beginning of a SETUP sequence. A check for the SC_SETUP_TYPE flag from UHS is also made here. If it is set, the value of a two bit state signal is reset.

- Read ENDPOINT0.TRANSTYPE.STATUS_REG
- Determine if packet is SETUP packet
- Read number of bytes pending in UHS FIFO
- Track state of the SETUP operation according to [Table E.1](#)

<table>
<thead>
<tr>
<th>Current value</th>
<th>Next value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>10</td>
<td>After initial SETUP command DATA1 is first in sequence</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>Next use DATA1</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>Next use DATA0</td>
</tr>
</tbody>
</table>

**Table E.1**: Setup state tracking (CTRL_STATE)

Next State:
- **S_CHECK_EP0**: Size of data was NOT 8 bytes.
- **S_ReadByte**: Size of data is 8 bytes.

**S_ReadByte**

Transfer SETUP packet from UHS FIFO to RAM.

Next State:
- **S_Process**: All bytes transferred.
- \textbf{S\_PROCESS}:
  - Prepare IN FIFO by resetting it.
  - Determine what kind of SETUP was received by reading address at offset 1 from RAM.

Next State:
- \textit{S\_Proc\_SETADDR} : Set address request.
- \textit{S\_Proc\_GETDSC} : Get Descriptor request.
- \textit{S\_CTRL\_FINISH} : Set configuration request.
- \textit{S\_CHECK\_EP0} : Unsupported request.

- \textbf{S\_Proc\_GETDSC}:
  - Parse offset 3 of the received data in RAM.

Next State:
- \textit{S\_Proc\_GETDSC\_DEVICE} : Request to receive Device descriptor.
- \textit{S\_Proc\_GETDSC\_CONFIGURATION} : Request to receive Configuration descriptor.
- \textit{S\_CTRL\_FINISH} : Request to receive a STRING descriptor or request is not implemented.

- \textbf{S\_Proc\_GETDSC\_DEVICE}:
  - Transfer Device descriptor to IN FIFO in UHS.

Next State:
- \textit{S\_CTRL\_FINISH} : Always

- \textbf{S\_Proc\_GETDSC\_CONFIGURATION}:
  - Transfer Configuration descriptor to UHS IN FIFO

Next State:
- \textit{S\_CTRL\_FINISH} : Always

- \textbf{S\_Proc\_SETADDR}:
  - Save received address
  - Flag a pending address change

Next State:
- \textit{S\_CTRL\_FINISH} : Always
- **S_CTRL_FINISH**
  - Reset OUT FIFO.
  - Enable EP0 IN FIFO
  - Set DATA0/DATA1 sequence

**Next State:**
- **S_IRQDONE**: Always

- **S_XFER_Idle**

  Entry state for data transfer events.

**Next State:**
- **S_CLRFIFO_EP2**: OUT FIFO read by SPISE bus.
- **S_CLRFIFO_EP1**: IN FIFO written by SPISE bus.
- **S_CheckEP2**: No SPISE bus event (triggered by UHS interrupt)

- **S_CheckEP1**

  Check if data in EP1 FIFO has been sent to host and reset the FULL flag of the IN FIFO if it has.

**Next State:**
- **S_XFER_FINISH**: Always

- **S_CheckEP2**

  Check if data has arrive on EP2.

**Next State:**
- **S_CheckEP1**: No data arrived on EP2
- **S_XFER_OUT**: Data arrived on EP2

- **S_XFER_OUT**

  - Check transfer type (Must be OUT type).
  - Check for CRC errors
  - Check for bit stuffing error
  - Check DATA0/DATA1 status with saved internal value
  - Toggle saved internal DATA0/DATA1 value

**Next State:**
- **S_CLRFIFO_EP2**: Error detected
- **S_OUTREAD**: No errors detected
Transfer data from EP2 FIFO to the OUT FIFO facing the SPISE bus.

Next State:
- S_XFER_FINISH : Transfer complete

Clear EP1 FIFO before loading it with data

Next State:
- S_CLRFIFO_EP1 : Always

Load data from IN FIFO to EP1 FIFO

Next State:
- S_INLOAD : Always
- S_XFER_FINISH : When content of IN FIFO transferred.

Clear EP2 FIFO and set it ready

Next State:
- S_CLRFIFO_EP2 : Always
- S_EP2_RDY : Always

Enable EP1 using DATA0/DATA1 sequence
- Set the new DATA0/DATA1 sequence for EP1

Next State:
- S_XFER_FINISH : Always

Enable EP2 to receive data

Next State:
- S_XFER_FINISH : Always

Collect state for transfer states

Next State:
- S_Idle : Always
Appendix F

USB Host state Machine
(pseudo description)

- **S_Reset**
  - Reset USBHostSlave core
  - Setup UHS as Host
  - Enable interrupts in UHS
  - Force USB reset (on the USB wire)
  - Enable SOF generation

Next State:
- **S_Idle** : Always

- **S_Idle**
  - Idle state. Do nothing

Next State:
  - **S_WaitPolling** : Forced WAIT state triggered to let the bus settle
  - **S_Interrupt** : UHS interrupt occurred
  - **S_PROCDEV** : A device is connected, no pending IN transfers are scheduled and host not currently waiting for a SOF to occur

- **S_Interrupt**
  - Fetch UHS interrupt status register

Next State:
- **S_Idle** : No interrupts currently triggered
- **S_Interrupt_Process** : Interrupts triggered
Process interrupt status based on priority of interrupts
Priority in descending order with most important interrupt listed first

Next State:
- **S_INT_SOFSENT**: SOF was sent.
- **S_INT_XFERDONE**: Transfer done
- **S_INT_CONNECT**: Connection occurred
- **S_INT_RESUME**: Resume event detected

Reset UHS RX FIFO

Next State:
- **S_Idle**: Always

Reset timer used for tracking frames
Reset any pending IN transfers
Reset WAIT for SOF signal

Next State:
- **S_Interrupt_Ack**: Always

Determine if interrupt caused by connection or disconnection
DISCONNECT: Remove device from internal connection table
CONNECT: Add device to internal connection table

Next State:
- **S_Interrupt_Ack**: Disconnect event
- **S_SetPolarity**: Connect event
Unimplemented. Silently discard interrupt

**Next State:**
- **S_Interrupt_Ack**: Always

**S_INT_RESUME**

- Determine direction of completed transfer

**Next State:**
- **S_Interrupt_Ack**: OUT transfer completed
- **S_RXDONE**: IN transfer completed

**S_INT_XFERDONE**

- Reset the interrupt that was last processed

**Next State:**
- **S_Interrupt**: Always

**S_PROCDEV**

- Decide based on a counter of amount of IN requests made whether or not to stop waiting for the current command to finish or to resend it. If NAK or no reply is received a preset number of times, the command is resent. If however the packet was fragmented so that it needs to be received in many pieces (passing this state for every IN request) the counter will not be reset.
- Track device connection sequence ([section 5.6.1](#))
- Next USB FSM state depends on the device tracking state as shown in [Table 5.6](#)

**Next State:**
- **S_Idle**: Device in RESET state or catch-all condition (i.e. error)
- **S_SETUP_GETDSC_DEV**: Get device descriptor
- **S_SETUP_ADDR**: Set device address
- **S_SETUP_GETDSC_CFG**: Get configuration descriptor
- **S_SETUP_SETCFG**: Set configuration
- **S_CHECKXFER**: Major mode is **S_CONFIGURED**
SetPolarity

- Force a USB wire reset
- Setup internal device connection table with correct polarity and speed

Next State:
- S_Interrupt_Ack : Always

SETUP_GETDSC_DEV

- Send GET DEVICE DESCRIPTOR command to device

Next State:
- S_SETUP_SENT : Minor state 00
- S_SETUP_REQIN : Minor state 01 or 10
- S_SETUP_HANDSHAKE : Minor state 11

SETUP_GETDSC_CFG

- Send GET CONFIGURATION DESCRIPTOR command to device

Next State:
- S_SETUP_SENT : Minor state 00
- S_SETUP_REQIN : Minor state 01 or 10
- S_SETUP_HANDSHAKE : Minor state 11

SETUP_ADDR

- Send SET ADDRESS command to device

Next State:
- S_SETUP_SENT : Minor state 00
- S_SETUP_HANDSHAKE : Minor state 11

SETUP_SETCFG

- Send SET CONFIGURATION command to device

Next State:
- S_SETUP_SENT : Minor state 00
- S_SETUP_HANDSHAKE : Minor state 11
**S_SETUP_SENT**

- Awaits an interrupt in the UHS
- Reads the status register for the interrupt and determines if it was due to a transfer.

**Next State:**
- **S_SETDEVICE_NEXTSTATE**: ACK received from device
- **S_Idle**: No ACK (remain in the same device state)

**S_SETDEVICE_NEXTSTATE**

- Reset the number of IN requests counter (used in S_PROCDEV)
- Decide the next state for the device state based on current major state, command state and minor state.
- After a minor state Handshake completed the command state is increased
- After the last command state for the current major state has been completed the major state is increased.

**Next State:**
- **S_Idle**: Always

**S_SETUP_REQIN**

- Tell UHS core to request an IN transfer from the device
- Set Pending IN transfer flag
- Set Wait for SOF flag

**Next State:**
- **S_Idle**: Always

**S_RXDONE**

- Read data count in UHS RX FIFO
- Reset Pending IN transfer flag
- Check if the device major state is S_CONFIGURED

**Next State:**
- **S_Interrupt_Ack**: Error, zero bytes in the UHS RX FIFO
- **S_CONTROLIN**: Device not yet configured
- **S_CHECKXFER**: Device has been configured
• Check DATA0/DATA1 sequence

• Store received data to a RAM by appending it at the address given by the so far received amount of data. This appends a fragmented packet to the RAM without overwriting previously received data.

• Add the received amount of data to a global register so that the next append operation will again not overwrite any data in RAM

Next State:

• **S_CONTROL_PARSE**: Always

• Based on which command state the device status has. The address of the RAM is pointed to where the total length of the packet is located.

• Total size of the packet is compared to the global register containing the amount of data received so far.

• If the complete packet is received the parser will start. If data is missing the Expect more data flag is asserted and no further processing of the packet is done.

• DEVICE descriptor. Saved to RAM but not parsed.

• CONFIGURATION descriptor.

  1. Check descriptor type is configuration descriptor
  2. Check if total size is larger than the RAM can hold
  3. Fetch the value needed for the SET_CONFIGURATION command
  4. Fetch size of configuration descriptor
  5. Skip past configuration descriptor to parse the interface descriptor
  6. Check descriptor type is interface descriptor
  7. Fetch number of endpoints
  8. Fetch size of interface descriptor
  9. Skip past interface descriptor to parse the endpoint descriptor
  10. Check descriptor type is endpoint descriptor
  11. Check that endpoint is not isochronous
  12. Fetch endpoint address
  13. Fetch endpoint direction (IN/OUT)
14. Fetch max endpoint FIFO receive size (as told by the device)

15. Add size of endpoint descriptor to RAM address and skip the rest of the descriptor

16. Check if address equals total size of packet received. If it is not equal expect another endpoint descriptor otherwise Finish parsing.

Next State:
- **S_Interrupt_Ack**: At end of parsing

-------------------------- S_SETUP_HANDSHAKE --------------------------
- Force OUT FIFO empty
- Set the type of handshake based on which SETUP command completed
- Wait until UHS interrupt triggers
- Check whether interrupt caused by ACK, NAK or timeout

Next State:
- **S_Idle**: NAK received or timeout while waiting for reply
- **S_SETDEVICE_NEXTSTATE**: Handshake completed

-------------------------- S_CHECKXFER --------------------------
- Check if SPISE bus OUT FIFO has data to transmit
- Check if SPISE bus IN FIFO is ready to receive
- Check if data received but not uploaded to SPISE bus IN FIFO yet

Next State:
- **S_CHECKXFER_TX**: Data exist in SPISE bus OUT FIFO and UHS TX FIFO not busy
- **S_CHECKXFER_RX**: Data exist in UHS RX FIFO and SPISE bus IN FIFO not busy
- **S_CHECKXFER_POLL**: No data to transmit or receive. Poll device for data
- **S_Idle**: Wait Polling flag asserted. Or no FIFO was ready

87
- **S_CHECKXFER_POLL**
  - Check frame timer that enough time exist in the frame
  - Send IN request to device on the DATA endpoint
  - Await interrupt caused by data received, NAK or timeout
  - Fetch data count in UHS RX FIFO

**Next State:**
- **S_CHECKXFER** : Data received
- **S_Idle** : No data received or not enough time was left in the frame

- **S_CHECKXFER_RX**
  - Check DATA0/DATA1 sequence on packet
  - If no errors. Transfer packet from UHS RX FIFO to SPISE bus IN FIFO.

**Next State:**
- **S_ResetRXFIFO** : DATA0/DATA1 mismatch
- **S_Idle** : Packet transferred

- **S_CHECKXFER_TX**
  - Transfer SPISE bus OUT FIFO to UHS TX FIFO
  - Check time left in frame
  - Set DATA0/DATA1 sequence
  - If time left start sending packet
  - Keep two counters. One for the end of the UHS TX FIFO and one for the max receive size of the device RX FIFO
  - If more data in SPISE bus OUT FIFO than device can handle. Send fragmented packets as large as the device can handle until the complete packet has been delivered to the device without errors
  - Await interrupt from UHS caused by data transmitted, device unable to receive or a timeout
  - Check status and trigger a resend of the data until maximum number of retries passed or until ACK received from the device.

**Next State:**
- **S_Interrupt_Ack** : Transfer done and was originally called from an interrupt state
- **S_Idle** : Not enough time left in frame to send data or transfer done and was not originally triggered by an interrupt

88
• Wait for a predefined amount of clock cycles settable at compile time

Next State:
• S_IDLE : When waiting finished