



Jaakko Korhonen

**TESTER FOR BATTERY INTERFACE MASTER**

# **TESTER FOR BATTERY INTERFACE MASTER**

Jaakko Korhonen  
Bachelor's Thesis  
Spring 2013  
Degree Programme in Information Technology  
Oulu University of Applied Sciences

# TIIVISTELMÄ

Oulun seudun ammattikorkeakoulu  
Tietotekniikan koulutusohjelma, langattomien laitteiden suuntautumisvaihtoehto

---

Tekijä: Jaakko Korhonen

Opinnäytetyön nimi: Tester for Battery Interface Master

Työn ohjaajat: Markus Littow (ST-Ericsson), Ensio Sieppi (OAMK)

Työn valmistumislukukausi ja -vuosi: Kevät 2013

Sivumäärä: 86 + 1 liite

---

Opinnäytetyön tavoitteena oli luoda testausympäristö MIPI-allianssin akkuliitännän isäntälaitteelle. Testausympäristön vaatimuksina oli se, että testausympäristön avulla voidaan testata ja mitata sähköisiä ja ajoituksellisia parametreja akkuliitännän isäntälaitteesta. Testausympäristön vaatimuksena oli myös, että testausympäristöllä voi toistaa korkean tason käyttötilanteita kaupallisten orjalaitteiden kanssa. Työn tilaajana toimi ST-Ericsson Oy.

Opinnäytetyön toteutus jakaantui kolmeen osioon ja seurasi pääpiirteittäin ohjelmistokehityksessä yleisesti käytettyä vesiputousmallia. Ensimmäinen osa käsitteli akkuliitännäjärjestelmän korkean tason ohjausta. Toisessa osiossa luotiin säädettävä orjalaite akkuliitännään. Viimeisessä osiossa näitä kahta osaa käytettiin mittauksien tekemiseen. Jokaisen osion läpivienti seurasi polkua määrittely, suunnittelu, toteutus ja testaus.

Opinnäytetyön kaikkien kolmen osion toteutus onnistui hyvin, ja integrointivaiheessa ei tullut ongelmia. Alkuperäinen suunnitelma osoittautui oikeaksi, koska siihen ei tarvinnut tehdä kompromisseja tai muutoksia. Testausympäristön avulla ST-Ericsson voi varmistaa MIPI-allianssin akkuliitännän isäntälaitteen sisältävien piirien toiminnan useiden parametrien vaikuttaessa järjestelmään.

---

Avainsanat:

testausympäristö, MIPI, BIF, LabVIEW, C/C++, DLL

## ABSTRACT

Oulu University of Applied Sciences  
Information Technology and Telecommunications, Wireless Devices

---

Author: Jaakko Korhonen

Title of thesis: Tester for Battery Interface Master

Supervisors: Markus Littow (ST-Ericsson), Ensio Sieppi (OAMK)

Term and year when the thesis was submitted: Spring 2013    Pages: 86 + 1 Appendice

---

The purpose of this thesis was to create a test environment for a Battery Interface Master device. The Battery Interface Master is a part of Battery Interface specification by MIPI Alliance. The test environment was required to be able to test and measure electrical and timing parameters from a Battery Interface Master in Signalling Layer. The test environment was also required to be able to reproduce high level use case situations with real commercially available BIF Slave devices. The thesis was commissioned by ST-Ericsson.

The thesis was divided into three parts which generally followed the waterfall model used in software development. In the first part high level control for the Battery Interface system was created. The second part was to create a configurable BIF Slave. Finally, these two parts were used in the measurements part to create example measurements. Every part started with a definition phase followed by a design, an implementation and finally a testing phase.

The implementation of all three parts was smooth and integration phase did not cause any additional issues. The initial design proved to be a good choice because no compromises or design changes needed to be done in the process. As the result of this thesis work, ST-Ericsson is now equipped with a test environment to verify multiple performance parameters of a BIF Master device on ICs implementing it.

---

Keywords:

test environment, MIPI, BIF, LabVIEW, C/C++, DLL

## **ACKNOWLEDGEMENTS**

This thesis was started and completed in the first half of 2013.

I wish to thank Esko Kurttila from ST-Ericsson for providing me this great thesis opportunity, Markus Littow from ST-Ericsson for guiding and supervising the thesis, Ensio Sieppi from OAMK who supervised and guided the thesis and the whole characterization laboratory team.

In addition, I wish to thank my girlfriend and family for supporting me through my studies.

Oulu 27 April 2013

Jaakko Korhonen

## TABLE OF CONTENTS

TIIVISTELMÄ	3
ABSTRACT	4
ACKNOWLEDGEMENTS	5
LIST OF ACRONYMS	8
1 INTRODUCTION	10
2 MIPI BATTERY INTERFACE SPECIFICATION	12
2.1 Typical Battery Interface System	14
2.2 Architecture of Battery Interface Master	16
2.3 Data Encoding	18
2.4 Parity Checksum	19
2.5 Physical Layer	20
3 TEST ENVIRONMENT OVERVIEW	23
3.1 Overview	25
3.2 Communication Standards of Measurement Devices	28
3.2.1 National Instruments Virtual Instrument Software Architecture	28
3.2.2 General Purpose Interface Bus	29
3.2.3 Universal Serial Bus Test & Measurement Class	29
3.2.4 Inter-Integrated Circuit	30
3.2.5 Digital Interface Board	31
4 WRAPPER FOR BATTERY INTERFACE SOFTWARE ENGINE	32
4.1 Battery Interface Software Engine and its Parts	33
4.2 Using the Battery Interface Software Engine	34
4.3 Storing the Required Variables for Battery Interface Manager	36
4.4 Callback Functions and LabVIEW	37
4.4.1 LabVIEW User Events	39
4.4.2 Windows Event Objects	40
4.4.3 Read Byte Callback Wrapper Function	41
4.5 Architecture for Wrapped Battery Interface Software Engine	43
4.6 LabVIEW Interface to Battery Interface Software Engine	45
4.6.1 Initializing Battery Interface Software Engine	45
4.6.2 Scheduling Commands for Battery Interface Software Engine	46
4.6.3 Allocating Memory for dataIn and dataOut Buffers	48
4.6.4 Reading Memory With Call Library Node	49

4.6.5 Freeing Allocated Memory	50
4.6.6 Running the Battery Interface Manager State Machine	50
4.6.7 Retrieving Recall Delay from Battery Interface Software Engine	50
4.6.8 Closing the Battery Interface Software Engine	51
4.6.9 Resetting the State Machine	52
4.6.10 Translating Battery Interface Result into LabVIEW Error	52
5 SIMULATED BATTERY INTERFACE SLAVE	53
5.1 Battery Interface Slave Controller	55
5.2 Timebase Multiplier	56
5.3 Avalon Memory Mapped Interface	57
5.4 Dual-Port RAM	59
5.5 Nios II Interface	60
5.6 Simulating Battery Interface Slave	61
5.7 Control Registers	63
5.7.1 Errors Register	63
5.7.2 Timebase Multiplier Register	64
5.7.3 Control Register	65
5.7.4 Measured clk_div	65
5.7.5 State Machine Registers	66
5.8 LabVIEW Interface	68
5.8.1 Reading and Writing Control Register	69
5.8.2 Reading and Writing Random Access Memory	69
5.8.3 Reading and Clearing Errors	70
5.8.4 Reset	71
5.8.5 Configuring Interrupts	71
5.8.6 Controlling Timebase	72
5.8.7 Reading Timebase Multiplier	73
5.8.8 Writing Timebase Multiplier	73
5.8.9 Reading State Machine States	74
6 MEASUREMENTS	76
6.1 Timebase Sweep for Battery Interface Master Receiver	76
6.2 Temperature Measurement with Real BIF Slaves	78
7 THOUGHTS AND CONCLUSIONS	81
8 LIST OF REFERENCES	83
APPENDIXES	
Appendix 1 Compilation Guide for Battery Interface Software Engine	

## LIST OF ACRONYMS

API	Application Programming Interface
ATE	Automatic Test Environment
BCL	Battery Communication Line
BIF	MIPI Battery Interface Standard
CTS	Conformance Test Suite
DLL	Dynamic-Link Library
EMC	Electromagnetic Compatibility
ESD	Electrostatic Discharge
FPGA	Field-Programmable Gate Array
GPIB	General Purpose Interface Bus
GPIO	General Purpose Input/Output
HAL	Hardware Abstraction Layer
HP-IB	Hewlett-Packard Interface Bus
I/O	Input/Output
I2C	Inter-Integrated Circuit
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IVI	Interchangeable Virtual Instrument Foundation
LSB	Least Significant Bit
MIPI	Mobile Industry Processor Interface

MM	Memory Mapped
MSB	Most Significant Bit
MSDN	Microsoft Developer Network
NMOS	N-Channel Metal-Oxide-Semiconductor Field-Effect Transistor
NVM	Non-Volatile Memory
PLL	Phase-Locked Loop
PMOS	P-Channel Metal-Oxide-Semiconductor Field-Effect Transistor
RAM	Random Access Memory
RC	Resistor-Capacitor
RS-232	Radio Sector of The Electronic Industries Association Standard Number 232
SCL	Serial Clock Line
SCPI	Standard Commands for Programmable Instruments
SDA	Serial Data Line
SL	Signalling Layer
SoPC	System on a Programmable Chip
SW	Software
UID	UniqueID
USB	Universal Serial Bus
USBTMC	USB Test & Measurement Class
VISA	Virtual Instrument Software Architecture
VME	Versa Module Eurocard
VXI	VME eXtensions for Instrumentation

## 1 INTRODUCTION

BIF (Battery Interface) is a mobile device battery interface specification by MIPI (Mobile Industry Processor Interface) Alliance. It is a robust, cost-effective and flexible interface between a mobile device and a battery pack or packs. There can be Low cost battery packs or Smart battery packs in BIF. The Low cost battery packs use an identification resistor  $R_{ID}$  value to define the battery pack model whereas the Smart battery packs use digital communication to pass the same information. The digital communication also provides other services than identification through it. A Smart battery pack also contains an  $R_{ID}$ , but the resistor value is only used to tell that it is a Smart battery pack. For identification and/or communication BIF uses a BCL (Battery Communication Line) which is a single-wire interface. BCL is used for both Low cost battery packs and digital Smart battery packs. (4, p. 1, 4)

BIF specification is relatively young because the work for it started in 2010 and the first version 1.0 release came in 2012. A new version 1.1 with added features is planned to be released sometime in 2013. (13)

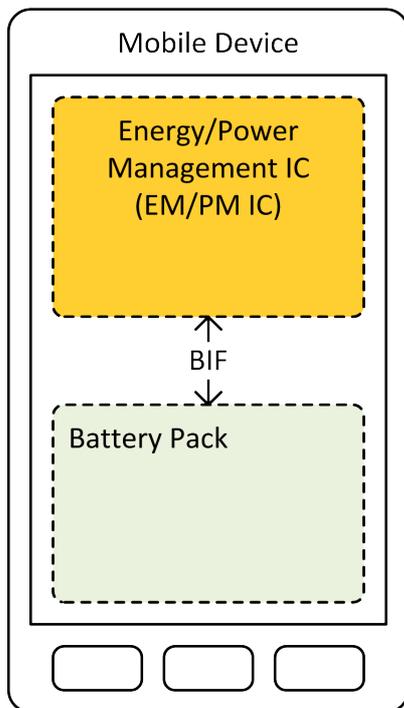
To ensure reliable operation BIF introduced a CTS (Conformance Test Suite) for both interoperability and protocol. Every BIF device has to pass these CTSs. The interoperability CTS guarantees that devices developed by different parties work flawlessly together. The protocol CTS ensures that the communication protocol specification is strictly followed. Conforming the protocol CTS provides a good basis for running interoperability CTS tests successfully. In other words the interoperability CTS verifies that the devices from different parties work together and the protocol CTS guarantees that it was not an accident.

For the time being there is no CTS or test environment for the physical layer even though it is very important to know whether the physical layer conforms to the BIF physical layer specification. The test environment is called Tester for BIF Master and it

can be used to measure electrical and timing parameters from a BIF Master. Additionally, erroneous use cases reported from the field can be also reproduced with the Tester for BIF Master.

## 2 MIPI BATTERY INTERFACE SPECIFICATION

BIF is the first dedicated specification to offer a robust, cost-efficient and flexible communication interface between a mobile device and its battery pack (see figure 1). A communication interface is needed because the mobile device must identify the model of the battery pack in order to ensure safe operation. To manage a battery safely, it is required that the charging parameters, the temperature and the authenticity are known. A standardized battery interface reduces development costs throughout the lifespan of the battery pack. (4, p. 1)



*FIGURE 1. A mobile device with BIF*

The cost-efficiency is provided by the reduced development costs in several links of the product chain. Development costs are reduced because of the IP reusability. The IP reusability is increased more because of the fact that a BIF Master can support both Low cost and Smart battery packs. In Low cost battery packs a cheap resistor is used for the identification. Also the digital logic required by a Smart battery pack can be implemented with a relatively low number of digital gates. (4, p. 1, 6)

The connector used between a battery pack and a host device is usually relatively expensive because it has to provide a good connection in demanding conditions. A typical battery pack connector is required to have a low resistance, be able to withstand vibration and resist corrosion. For example, corrosion resistivity is usually achieved by gold plating which in turn adds an additional step to the manufacturing process. Increased complexity in production increases the price. BIF uses a single-wire interface for communication which minimizes the pin count of the connector and thus reduces the price.

The flexibility of BIF comes from the support for both Low cost battery packs and Smart battery packs (see figure 2). The physical connector used in an application can be freely selected by the manufacturer because it is not specified in BIF. BIF only specifies the electrical interface requirements. The support for both types of battery packs allows the manufacturer to use a simple resistor identification for the simplest and cheapest batteries or a digital interface for more sophisticated battery packs. The possibility for digital communication brings a lot of new use cases for the battery pack. For example with the NVM (Non-Volatile Memory) function, the battery pack usage data can be stored inside the battery pack itself. By default BIF specifies a list of functions that every BIF device must support. These default functions alone are enough to implement a robust battery identification, and the option for user defined functions increases the flexibility even more. (4, p. 2, 8, 12)

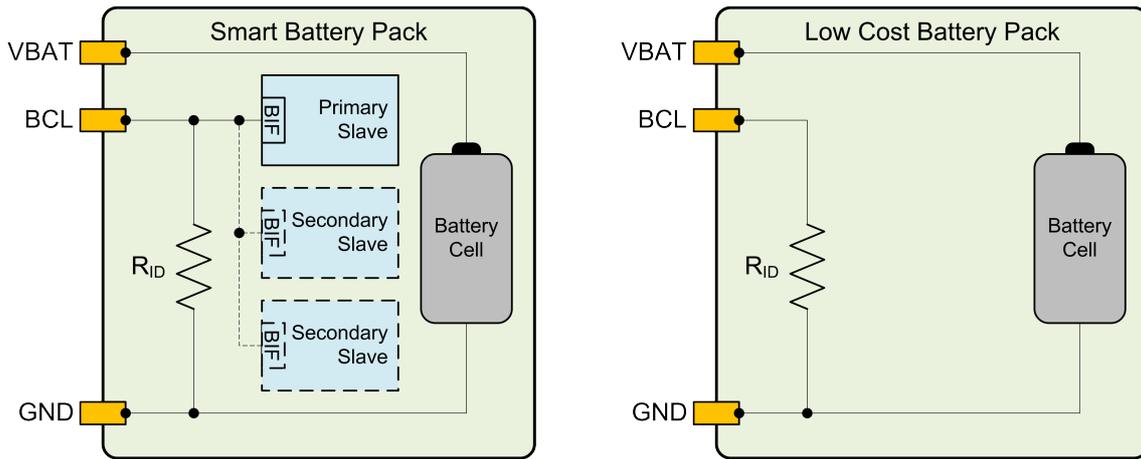


FIGURE 2. Example of a Smart battery pack and a Low cost battery pack (14, p. 18)

## 2.1 Typical Battery Interface System

A typical BIF system contains a BIF Master that lies somewhere in the host device and a BIF Slave that is in the battery pack (see figure 3). The BIF Master is usually located inside an EM/PM IC (Energy/Power Management Integrated Circuit). This is because the EM/PM IC is used for charging and draining the battery pack. A BIF Slave is usually located in the battery pack (see figure 3 left).

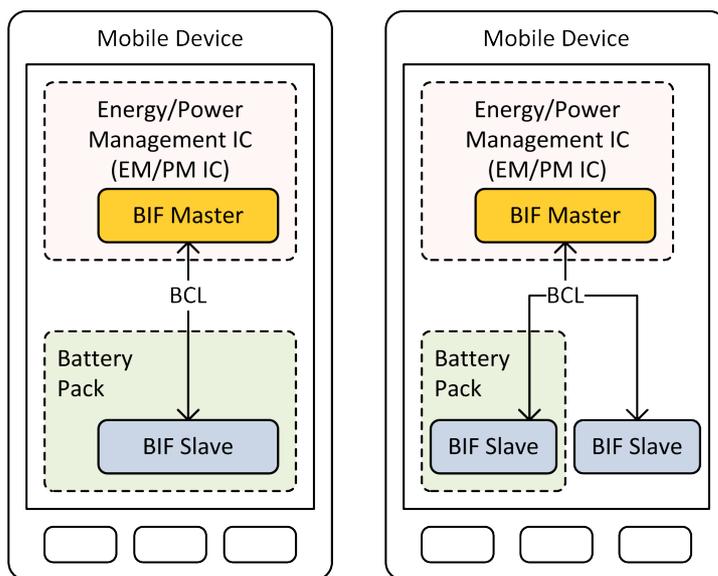


FIGURE 3. BIF systems with one slave or multiple slaves

BIF specifies that there can be BIF Slaves outside the battery pack but still inside the mobile device (see figure 3 right). This configuration is possible if a Smart battery pack with digital communication is used. For example, This external BIF Slave could be used as a temperature sensor. (4, p. 2, 8)

The structure of a BIF device can be presented as different layers (see figure 4) (4, p. 5). For a BIF Master these layers are the Application layer, the Protocol layer and the Physical layer. There is a BIF Manager in the Application layer that controls high level services and operations of the BIF system. The Protocol layer receives BIF words from the Application layer and after encoding, transmits them to the Physical layer. The Protocol layer is also responsible for receiving BIF words. When BIF words travel through the Physical layer, non-idealities are added to the waveforms due to the loading effect of external components connected to the BCL and the finite driving capability of the driver circuit. Also the parasitic components present on the BCL can further degrade the waveform.

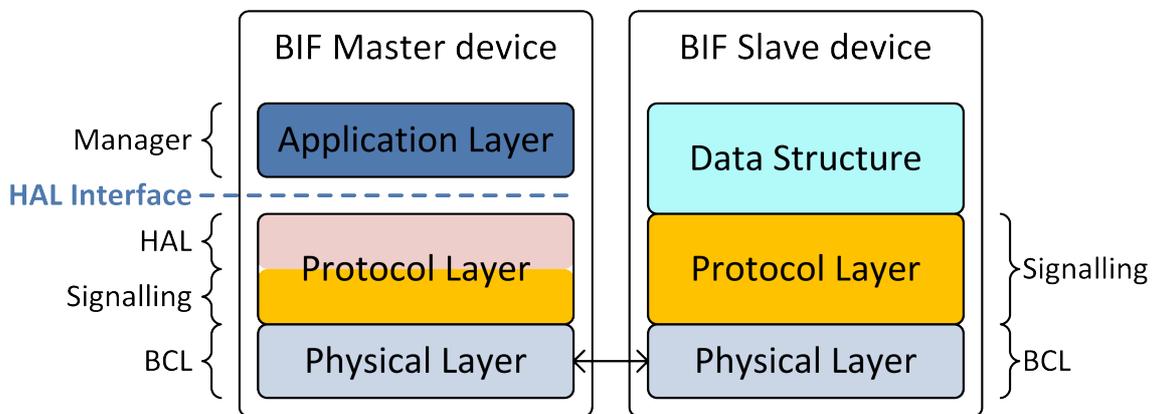


FIGURE 4. BIF Device types

Instead of Application layer, a BIF Slave device has Data Structure (figure 4) that contains the actual Application Data and Services. This means that there is no high level intelligence in a BIF Slave device. The Protocol layer executes predefined routines to the Data Structure depending on the commands sent by the BIF Master.

## 2.2 Architecture of Battery Interface Master

The three layers presented in figure 4 can be expanded to the full architecture of a BIF Master (see figure 5). Here the Protocol layer is divided to two distinct layers called the Hardware Abstraction Layer (HAL) and the Signalling Layer (SL). The Physical layer is also simplified to a path between the SL and the BIF Slave.

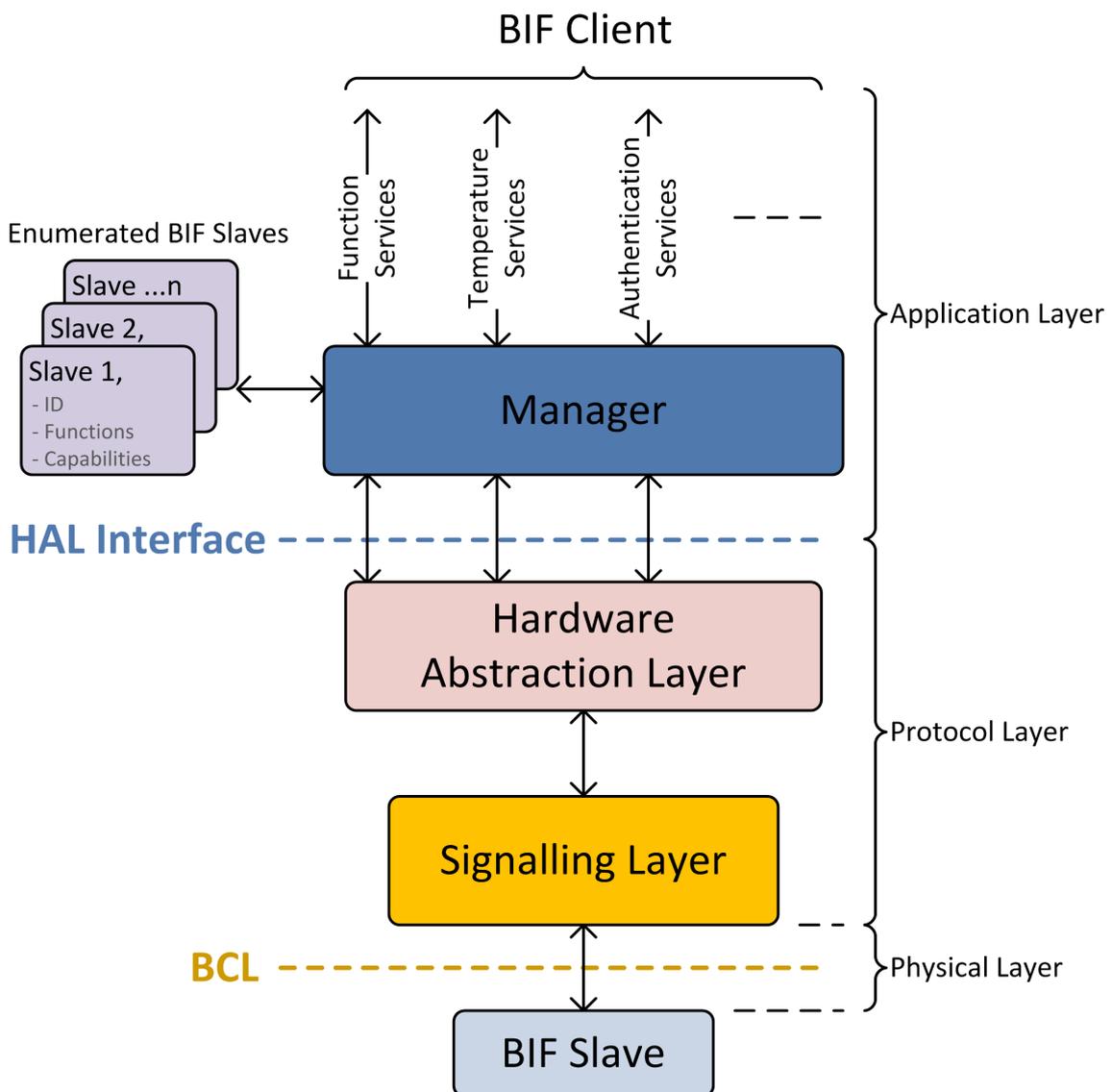


FIGURE 5. The architecture of a BIF Master

The Application Layer which contains the BIF Manager, controls the BIF system at a high level. For example the BIF Manager enumerates all the BIF Slaves present on the BCL (see figure 5). These enumerated BIF Slaves are then controlled through the Client interface with the help of the BIF Manager. The BIF Client interface in Tester for BIF Master is controlled with LabVIEW.

Under the BIF Manager there is a HAL Interface. The HAL Interface is a list of functions specified by BIF HAL Specification that the HAL must implement. In this sense the HAL acts as a driver for the SL. With the appropriate HAL the BIF Manager can control any SL platform independently.

The SL is the layer where the BIF words are transmitted and received from the BCL. The communication in the BCL is very time critical and should be implemented by digital logic. For example, in Tester for BIF Master the digital logic for the SL was located in the EM/PM IC.

### 2.3 Data Encoding

Data encoding is done in the SL and it includes a parity bit calculation, inversion bit setting and transmission to the BCL. An example of data words in the BCL is shown in figure 6. BIF words are separated by a Stop symbol. Data transmission in BIF is always initiated by the BIF Master and if applicable followed by a reply from the BIF Slave (4, p. 2, 6-7).

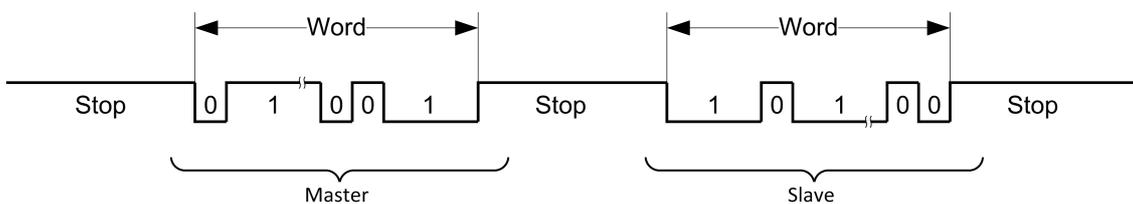


FIGURE 6. Data flow in the BCL (14, p. 29)

The bits inside a BIF word are encoded by TDC (Time Distance Coding) which means that the time between the signal edges varies by the sent bit. An example of TDC can be seen in figures 6 and 7. In BIF the TDC is based on a predefined timebase called  $\tau_{BIF}$ . The possible symbols for the BCL are  $0_B$ ,  $1_B$  and STOP and their corresponding lengths are one  $\tau_{BIF}$ , three  $\tau_{BIF}$ s and five  $\tau_{BIF}$ s. (4, p. 7)

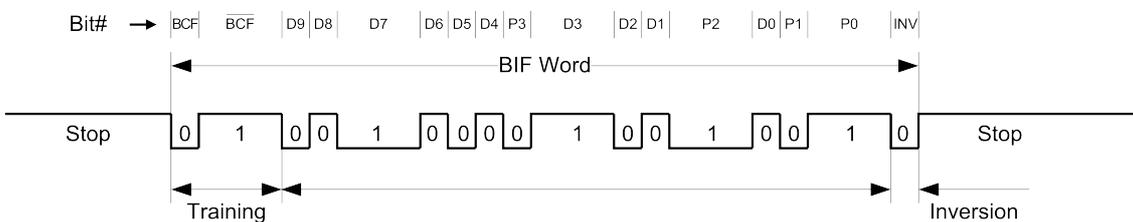


FIGURE 7. An example data word sent by a BIF Master (14, p. 31)

A full BIF word contains 17 bits which include 2 BCF (Bus Command Flag) bits, 10 data bits, 4 parity bits and 1 inversion bit. The BCF bits in the beginning serve also as a training sequence for the BIF Slave. By measuring the length of the training sequence, the BIF Slave can calculate the  $\tau_{\text{BIF}}$  used in the BIF word in order to be able to reply with the same timebase. (4, p. 7)

## 2.4 Parity Checksum

The parity algorithm in BIF conforms Hamming-15 coding to check the validity of the data in a BIF word. Hamming-15 has 11 data bits, 4 parity bits and a minimum distance of 3. The minimum distance means that it can either detect up to 2 bit errors or correct a 1 bit error but not both because it cannot distinguish between 1 and 2 bit errors. In BIF the parity checksum is only used for error detection and not for correction. Because no correction is applied, the checksum is able to detect bit errors up to 2 bits. The parity checksum calculation is explained in figure 8 below. (4, p. 7; 14, p. 33; 21)

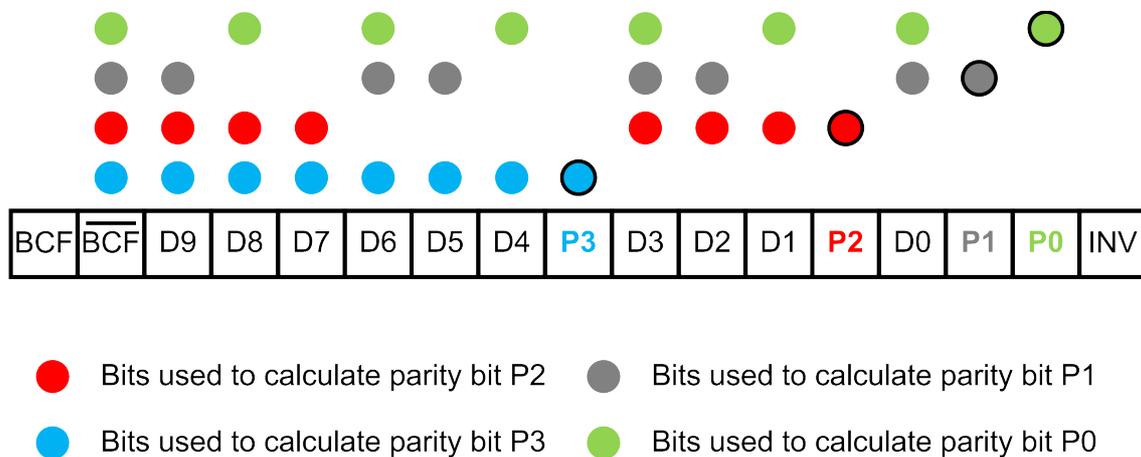


FIGURE 8. Hamming-15 Parity (14, p. 32)

The parity bits form a logical pattern (see figure 8) where the parity bit P0 (green) contains every other bit. The P1 parity bit (grey) follows the same pattern but it is always in pairs. This is done for every parity bit with the group size being  $2^n$ . The parity result bits are placed in logical positions to complete the pattern.

## 2.5 Physical Layer

Once the BIF words are encoded and the TDC is applied they are converted to voltage levels in the BCL. Voltage levels in the BCL are decided by the pull up resistor  $R_{PU}$  and the voltage  $V$  in a BIF Master driver illustrated in figure 9. The timebase used in TDC is defined by a BIF Master because the BIF specification states that the BIF Slave shall reply with the same  $\tau_{BIF}$  as it received (4, p. 5-6).

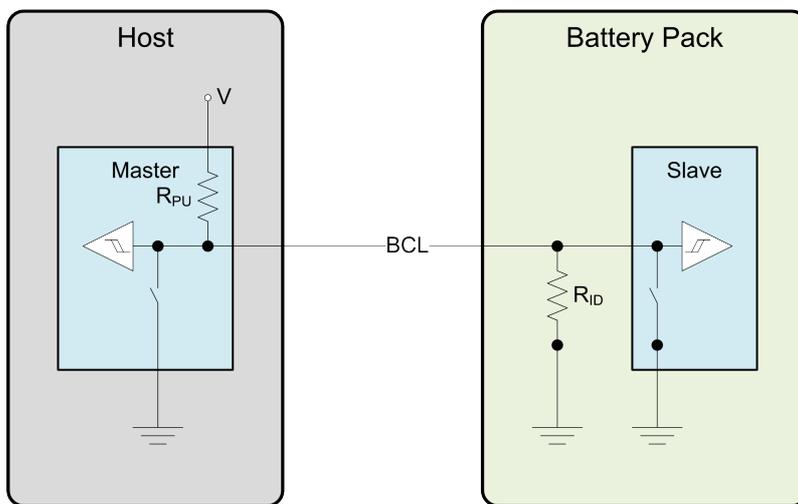
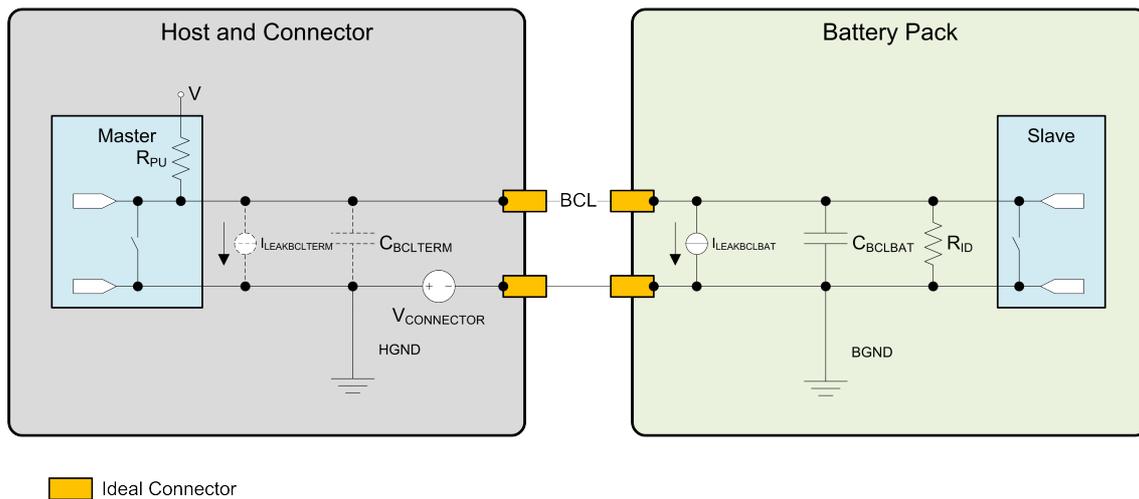


FIGURE 9. A physical Layer in a BIF System (14, p. 23)

Some parasitic capacitances and resistances may exist in the physical layer and they may introduce more non-idealities to the signal. The most important parasitic components are presented in figure 10. (14, p. 23-24)



*FIGURE 10. Stray parasitic components in a BIF System (14, p. 24)*

Both ends of the BCL, the host device and the battery pack, have parasitic capacitances (see figure 10). On the host device this capacitance is called  $C_{BCLTERM}$  and on the battery pack  $C_{BCLBAT}$  (14, p. 23). Most of the capacitance is created by the ESD/EMC (Electrostatic Discharge/Electromagnetic compatibility) protection circuits, PCB wires and the connector (14, p. 23). These parasitic capacitances affect the rise and fall times of the BCL. The biggest effect is on the rise time because the signal is pulled up with a resistor. The fall time is not affected so much because it is pulled down with a low resistance switch. There is also a leakage current  $I_{LEAKBCLBAT}$  and a little resistance from connectors.

The parasitic resistors describing the connectors are not drawn because they are presented as a voltage drop  $V_{CONNECTOR}$ . The voltage drop was chosen over the connector resistance because the maximum current between applications can vary. This means that by only defining the voltage drop, it is left to the manufacturer to select appropriate connectors for the application. By appropriate connector it is meant that the  $V_{CONNECTOR}$  voltage resulted from the maximum current shall not exceed the value specified in the BIF specification.

The  $V_{\text{CONNECTOR}}$  is specified over a range which also contains negative values. For example, if a high current is drained from VBAT, a small voltage drop appears on the connector (see figure 2). This is also true for the GND connector because the current flows in a loop. Because the current direction on the GND is towards the battery, it creates a negative voltage on  $V_{\text{CONNECTOR}}$ . This effectively alters the GND levels for the host device and the battery pack. When the battery pack transmits a ground referenced signal through the BCL, which does not suffer from the voltage drop, it will be seen as a negative voltage at the host device. This also applies in reverse when the battery is charged with a high current.

### **3 TEST ENVIRONMENT OVERVIEW**

Tester for BIF Master is needed to investigate the effect of various parasitic components present in the Physical layer of a BIF system as described in chapter 2.5. These parasitic components add non-idealities to the waveform generated in the SL. The biggest non-idealities that are caused by the parasitic components are slower rise times and ground level differences. Impacts to the connector can also cause short contact breaks to the BCL. Temperature and manufacturing process add their own errors to the waveform.

#### **Signal Rise Time**

The rise time is determined by the pull up circuit strength of a BIF Master and the total capacitance in the BCL (see figure 10). The capacitance does not have as big an effect to the fall time because the signal is pulled low with a switch. When the switch is conducting, the resistance is near to a short circuit and as such it provides greater drive strength than a pull up resistor. The rise time directly alters the way a BIF Slave or a BIF Master receive messages. Being a digital interface these RC (Resistor-Capacitor) slopes on edges will result in timing errors and as such the BCL capacitance is a limiting factor to the amount of BIF Slaves that can be simultaneously connected.

#### **Connector Contact Breaks**

Noise is also generated in the BCL by connector contact breaks (4, p. 6-7). These can happen when a phone is dropped or gets hit. Connector contact breaks are not studied further in this thesis. Nevertheless, it can be added to the test environment because the test environment is built loosely on a laboratory environment.

### **Slow Clock**

In some situations the BIF Master and/or the BIF Slave might be running on a slow 32,768 kHz clock frequency. Even with the slow clock frequency, the BIF Slave must be able to measure the  $\tau_{\text{BIF}}$  accurately enough in order to be able to respond to the BIF Master with the correct timebase. The BIF Master should then be able to receive these timebase skewed messages up to certain limits. This timebase testing is one of the main objectives for Tester for BIF Master. (4, p. 6-7)

### **Temperature**

Physical layer signalling performance depends on the operating temperature due to temperature sensitivity of the components. Temperature testing was not conducted in the thesis. However, it is possible to heat/cool the test environment to the desired temperature without any modifications to the test environment itself.

### **Manufacturing**

When manufacturing integrated circuits into wafer, the process contains some tolerances. These tolerances can be simulated with a corner process where worst case situations are created intentionally. Different manufacturing corners can be achieved by altering the operating speeds of the NMOS (N-Channel Metal-Oxide-Semiconductor Field-Effect Transistor) or PMOS (P-Channel Metal-Oxide-Semiconductor Field-Effect Transistor) transistors on the wafer. Different speed grades are then divided into Typical (T), Fast (F) or Slow (S) categories. (24)

For each corner sample the speed grade of each transistor family is altered and then named accordingly. This creates five possible corner samples which are TT, FF, SS, FS and SF. The first TT corner is not really a corner because both of the NMOS and PMOS transistors are manufactured to meet the typical speed grade. However, it is still provided as a reference for a typical device. The next two FF and SS corner samples do not usually cause major issues in the chip because both transistor families are altered evenly. Usually FS and SF corners cause most of the issues, because one part of the IC is working faster than the other part which can create timing errors that can cause the logic to malfunction. (24)

### 3.1 Overview

The Tester for BIF Master (see figure 11) consists of three parts which are the Wrapper for BIF SW Engine, Simulated BIF Slave and Measurements. The first two parts create the essential control and response for the BIF Master in order to test and measure it. The control is created with an existing BIF SW (software) Engine integrated into LabVIEW. The BIF SW Engine provides a high level interface to the BIF system (see red path in figure 11).

The response can be created with a Simulated BIF Slave (see the black path in figure 11) or with a real BIF Slave (see the red path in figure 11). The Simulated BIF Slave is a modified version of an existing BIF Slave Verilog design. The existing BIF Slave design originates from an older project. The Simulated BIF Slave is synthesized to an FPGA (Field-Programmable Gate Array) chip on the DIB (Digital Interface Board). The red path route in figure 11 represents logical operation flow when a real BIF Slave is under test. The black path represents the configuration when the Simulated BIF Slave is used.

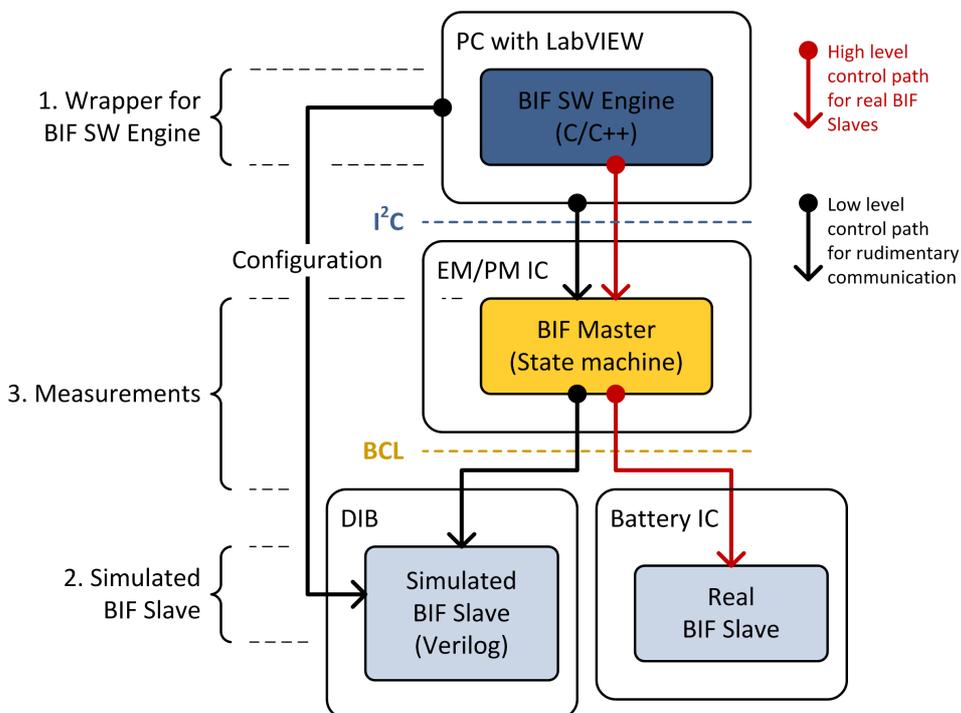


FIGURE 11. The components in the Tester for BIF Master

Figure 11 shows the physical components (PC, EM/PM IC, DIB...) in the test environment. However, it is more descriptive to inspect the system from the BIF Master Architecture view (see figure 12).

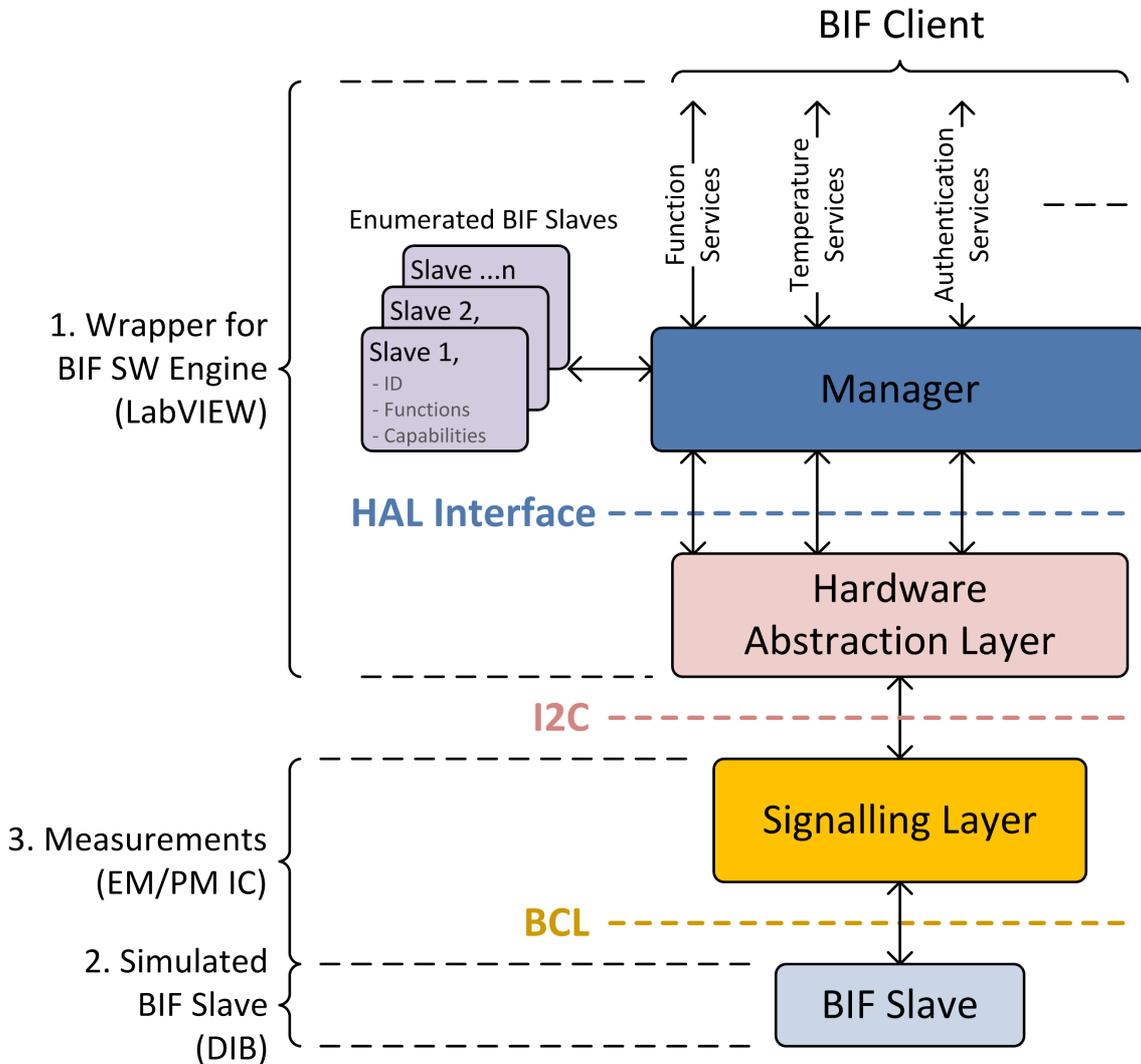


FIGURE 12. The BIF Master Architecture and the Tester for BIF Master

The BIF Master Architecture shown in figure 12 is very similar to the component overview in figure 11. The main difference is the Wrapper for the BIF SW Engine which is expanded over several blocks. This is important because the HAL has an important part in the Tester for BIF Master. The EM/PM IC shown in figure 11 is now located in the SL in figure 12.

## **Wrapper for Battery Interface Software Engine**

The BIF SW Engine did not require any modifications during Labview integration. The BIF SW Engine includes the BIF Manager and the HAL as illustrated in figure 12. The BIF Manager is used to perform all platform independent high level control of the BIF system through the HAL Interface. The purpose of the HAL is to unify the control interface to SL because the SL can be implemented in various ways.

The existing BIF SW Engine implementation used for Tester for BIF Master was written in C/C++ code and as such it could not be directly executed in LabVIEW. However, the BIF SW Engine could be compiled into a DLL (Dynamic-Link Library) and run from there.

## **Simulated BIF Slave**

An existing BIF Slave Verilog hardware design was modified for the Simulated BIF Slave step in order to generate  $\tau_{BIF}$  skewed responses. The single-port RAM (Random Access Memory) inside the BIF Slave was also changed to a dual-port RAM so it could be modified from outside with LabVIEW. With the new RAM custom patterns can be read from the BIF Slave easily. For example eye diagram tests can be done with the custom patterns.

When using a Simulated BIF Slave, the control follows the black path in figure 11. This is because the existing Simulated BIF Slave design was limited to very basic communication and as such it does not support all the features of BIF. The simulated BIF Slave cannot be used with the BIF SW Engine because it does not support some of the required mandatory features.

However, if the BIF SW Engine is used to control the Simulated BIF Slave, the BIF SW Engine will end up in an endless loop of UID (UniqueID) scans and resets. This happens because the Simulated BIF Slave does not support UID scans and the BIF SW Engine will think that there are no BIF Slaves present. The BIF SW Engine will then keep on scanning the BCL for new connected devices.

The configuration path in figure 11 is used to read and write the RAM contents in the Simulated BIF Slave. The configuration path is dedicated for configuration, monitoring and testing purposes, and it is not required by the BIF System.

### **Measurements**

In the Measurements step (figure 12), example measurements were done by using the Simulated BIF Slave and the BIF SW Engine. The operation of the Simulated BIF Slave was tested with a timebase sweep test to test the receiver performance of the BIF Master device.

The BIF SW Engine was also tested with real BIF Slaves that had a temperature function implemented. The test begun with the BIF SW Engine enumerating all the BIF Slaves. After enumeration, the temperatures were read from each BIF Slave regularly. LabVIEW was used to record and plot the resulted temperature values in real time.

## **3.2 Communication Standards of Measurement Devices**

This chapter describes different communication standards used while working with the Tester for Battery Interface Master. At the end of the chapter, the DIB is also briefly presented.

### **3.2.1 National Instruments Virtual Instrument Software Architecture**

NI-VISA (National Instruments Virtual Instrument Software Architecture) was developed by National Instruments to implement the VISA I/O (Input / Output) standard maintained by IVI (Interchangeable Virtual Instrument) Foundation. VISA specifies an interchangeable communication interface between similar test and measurement instruments. VISA compliant drivers exist for common I/O interfaces such as a GPIB (General Purpose Interface Bus), RS-232 (Radio Sector of the Electronic Industries Association standard number 232), USBTMC (Universal Serial Bus Test & Measurement Class) and VXI-11 (Versa Module Eurocard eXtensions for Instrumentation). (25)

From the developer's point of view the VISA greatly simplifies the access to the test and measurement instruments. For example, when writing code in LabVIEW, the same NI-VISA SubVIs can be used to communicate with GPIB, USBTMC and RS-232 devices.

### **3.2.2 General Purpose Interface Bus**

GPIB is a communication bus for test and measurement instruments. GPIB was originally named HP-IB (Hewlett-Packard Interface Bus) by Hewlett Packard in 1960s. Data transmission in a GPIB is byte-serial and bit-parallel. This means that the data is sent one byte at a time in a serial fashion. In Tester for BIF Master GPIB devices were widely used to support the EM/PM IC and its startup procedure. (23; 3, p. 8)

The GPIB devices found today are most likely built on these three standards IEEE-488.1 (Institute of Electrical and Electronics Engineers), IEEE-488.2 and SCPI (Standard Commands for Programmable Instruments). Each of the three standards specifies a section of the communication. Generally the IEEE-488.1 defines the hardware, the IEEE-488.2 defines the protocol and the SCPI defines standard commands for instruments. (23)

### **3.2.3 Universal Serial Bus Test & Measurement Class**

USBTMC is a USB (Universal Serial Bus) device class for test and measurement devices. The specification addresses communication with simple sensor devices, devices that communicate with IEEE-488 messages and devices with sub-addressable components. The device API (Application Programming Interface) for host is not specified in USBTMC but on the earlier mentioned VISA. (20, p.1)

LabVIEW communication with USBTMC devices is easy through NI-VISA SubVIs. In this test environment USBTMC is essential because it is the main communication channel between the Simulated BIF Slave on DIB and LabVIEW. The EM/PM IC where the BIF Master is located is controlled with I<sup>2</sup>C (Inter-Integrated Circuit) mastered by DIB.

### 3.2.4 Inter-Integrated Circuit

I<sup>2</sup>C is an industrial de facto communication standard for inter IC communications. It was developed by Philips Semiconductors (currently NXP Semiconductors) for simple bidirectional communication between ICs. (19. p. 1, 3)

An I<sup>2</sup>C device has two data wires, SDA (Serial Data Line) and SCL (Serial Clock Line), which operate in an open drain configuration. An I<sup>2</sup>C bus supports multi-master with collision detection. The data transfer is 8-bit oriented with bidirectional communication up to 3.4 Mbit/s in the High-speed mode. (19, p. 3-4)

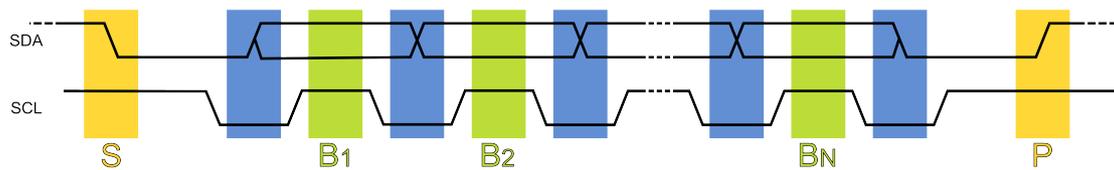


FIGURE 13. A timing diagram for I<sup>2</sup>C (22)

The basic operating principle of I<sup>2</sup>C is simple as illustrated in the timing diagram of figure 13. I<sup>2</sup>C words begin with a start symbol (S) and end with a stop (P) symbol. The start and stop are the only symbols where the SDA changes while the SCL is high. After the start symbol, the bits (B<sub>N</sub>) are transmitted one by one and sampled at the rising edge of the SCL.

### 3.2.5 Digital Interface Board

The DIB is an FPGA board used internally in ST-Ericsson and it offers various digital interfaces. The FPGA on the DIB is a Cyclone II device by Altera and it has 50 thousand logic elements and 294 I/O pins (1. p.4,6,). The DIB board has an external PLL (Phase-Locked Loop) for various clock signals, a 16 kB RAM for Nios II system, a 128 Mbit flash memory for general use and a 16 Mbit flash memory for storing the configuration of the FPGA (see figure 14).

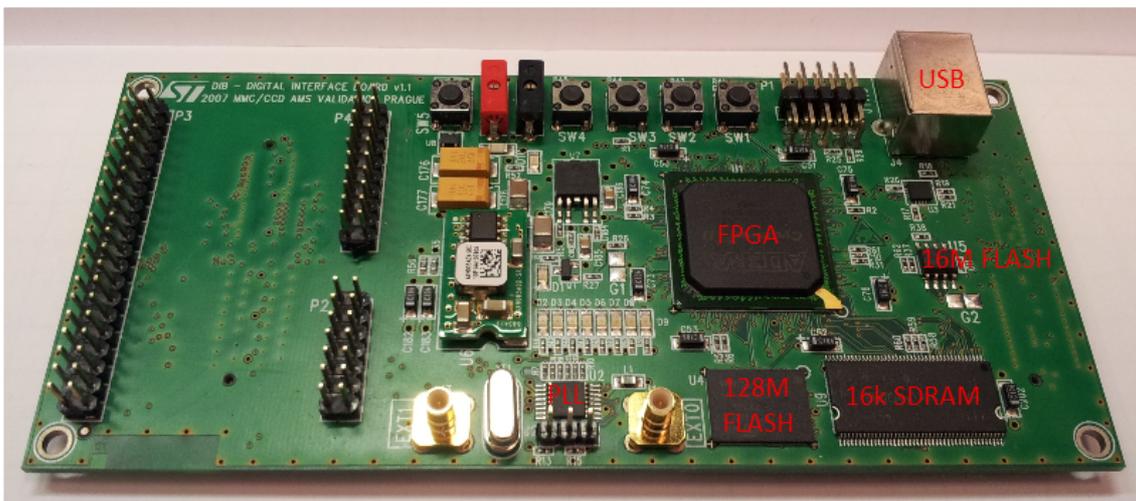


FIGURE 14. A Digital Interface Board

The DIB is a USBTMC USB device and it communicates with LabVIEW through NI-VISA. The Cyclone II FPGA on the DIB has a Nios II SoPC (System on a Programmable Chip) synthesized on it. The SoPC runs a C code that interprets USBTMC commands sent from LabVIEW.

The Simulated BIF Slave was synthesized as a part of the SoPC system as an Avalon MM Slave. This means that the Simulated BIF Slave can be controlled through USBTMC messages like the rest of the SoPC.

## 4 WRAPPER FOR BATTERY INTERFACE SOFTWARE ENGINE

The Wrapper for the BIF SW Engine is the first part of the Tester for BIF Master and it is located at the top of the architecture (see figure 15). The BIF SW Engine accepts high level commands from LabVIEW through the BIF Client interface. These commands are then translated into logical routines that accomplish the desired action. For example, there are high level commands for temperature measurement, authentication or  $\tau_{BIF}$  control.

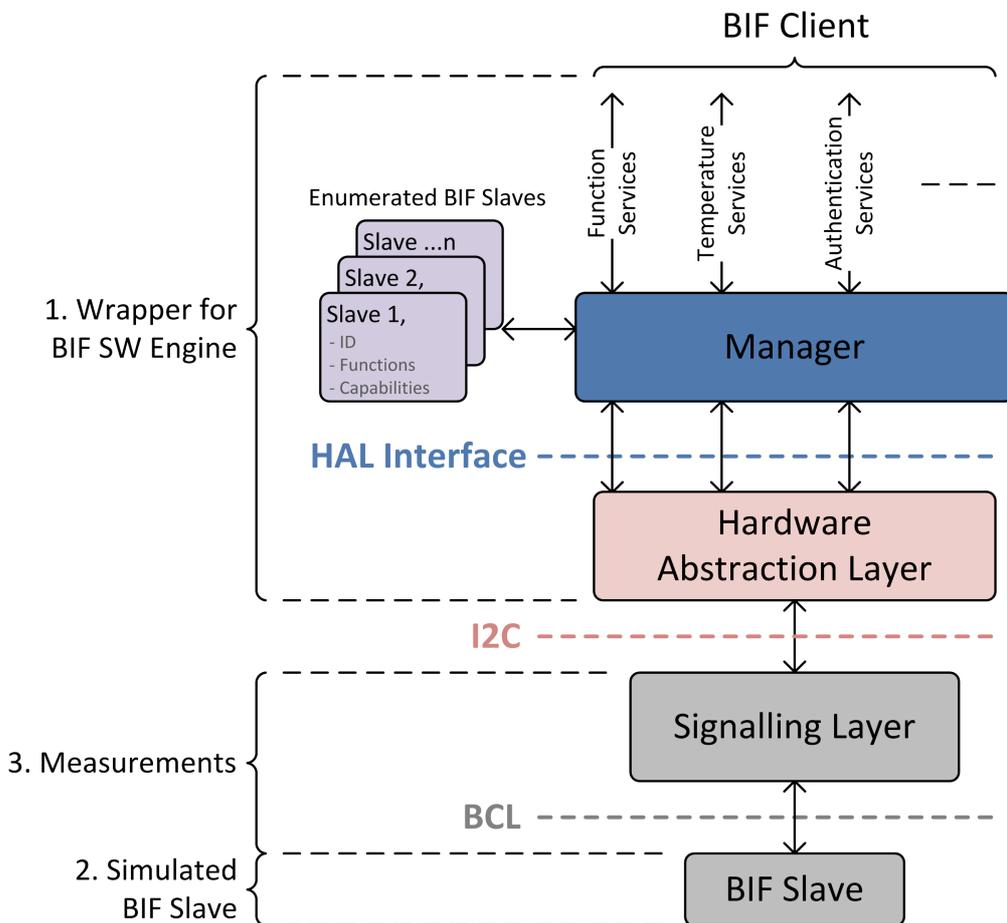


FIGURE 15. A wrapper for a BIF SW Engine in a BIF Master Architecture

The full potential of the BIF system cannot be utilized without the help of the BIF SW Engine because it would be too complex. For example UID (Unique Identifier) scanning,

enumeration and creating local structures of all the present BIF Slaves requires a few thousand operations from the SL.

#### 4.1 Battery Interface Software Engine and its Parts

The BIF SW Engine can be divided into two parts which are the BIF Manager and the HAL. The HAL can be further divided into the EM/PM IC driving logic and the callback function interface. A detailed block diagram of the BIF SW Engine is shown in figure 16.

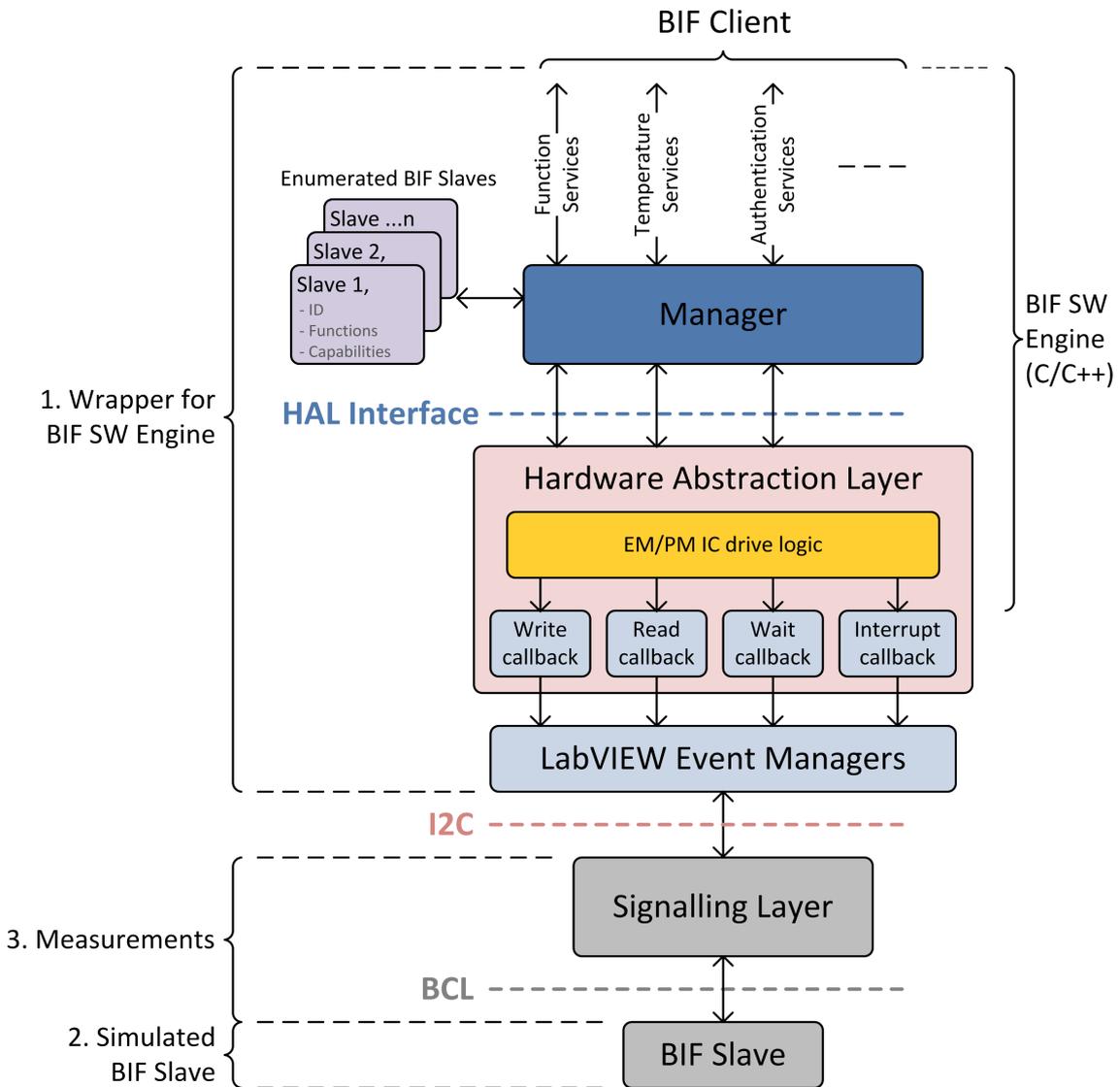


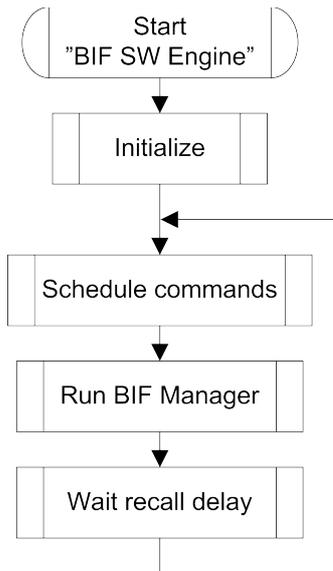
FIGURE 16. A more detailed block diagram of the Wrapper for the BIF SW Engine

The BIF SW Engine is written in C/C++ and it is implemented as a state machine. The BIF Manager uses the HAL Interface to control the BIF Master in the SL. The HAL contains the software routines for the EM/PM IC. The EM/PM IC can be controlled through the callback functions. The callback functions are used because the communication is not specified between the HAL and the SL. This means that the callback functions implement host specific communication. In the Tester for BIF Master the host system is LabVIEW and the communication with the EM/PM IC is done through I<sup>2</sup>C.

Unfortunately, since LabVIEW does not support C/C++ code, the BIF SW Engine cannot be directly run inside LabVIEW. However, LabVIEW can execute functions from external libraries such as DLLs. This means that the BIF SW Engine can be compiled to a DLL and then executed from there. Another problem is that LabVIEW does not directly support callback functions, but these can be circumvented with special wrapper functions. A compilation guide for the BIF SW Engine is provided in appendix 1.

## **4.2 Using the Battery Interface Software Engine**

Using the BIF Manager inside the BIF SW Engine is simple and it can be divided into four parts (see figure 17). Firstly, the variables for the BIF Manager are initialized and the internal callback pointers are stored. These callback functions include the write byte, the read byte, the wait time and the wait interrupt functions.



**FIGURE 17.** *Using the BIF SW Engine*

Secondly, in the initialization phase some configuration for the BIF Manager is done. The configuration includes settings for authentication, task completion and  $\tau_{BIF}$  setting. The configuration is done by calling the `bifMgrCommand` function from the BIF SW Engine library. Next, when the initial configuration is done, actual commands can be scheduled for the BIF Manager. This is done with the same `bifMgrCommand` function.

Thirdly, after scheduling commands for the BIF Manager state machine, it is ran with `bifMgrMachineRun` function. Depending on the scheduled commands, running the state machine results in various transactions with the SL.

Finally, after each time the state machine is ran, a recall delay is retrieved. This recall delay can be read with `bifMgrCommand` function with correct parameters. The recall delay can be immediate, fast, slow or no recall. If applicable, the recall delay is waited and the state machine is ran again.

When all the scheduled commands are executed, the BIF Manager raises a done flag. The done flag can be used to check if the BIF Manager is ready for new commands. The BIF Manager can also be configured to have background tasks which it runs without interaction from the user.

### **4.3 Storing the Required Variables for Battery Interface Manager**

The BIF SW Engine requires two variables in order to operate. The first variable is `bifMgrMachineState` which holds the current state of the state machine. The second variable is called `bifHwHal` and it holds the pointers to the callback functions.

The original idea was to store the `bifMgrMachineState` inside LabVIEW as a cluster variable. The cluster variable is the closest match for a C/C++ structure. This cluster variable would then have been passed to the `bifManagerMachineRun` function when needed.

Unfortunately, even though the LabVIEW cluster closely resembles the structures used in C/C++, it was not possible to pass a cluster inside a cluster to a DLL. This was also mentioned in the Help of LabVIEW: “Call Library Function Node does not support structures or arrays containing a pointer to other data or structures containing flat arrays that can be variably sized.” (16).

Since the `bifMgrMachineState` variable could not be stored as a LabVIEW cluster, it was stored as a global variable for the DLL. The variable gets initialized when the DLL gets loaded and stays in the memory until the end of the host process execution. Having the state machine data inside the DLL is a little drawback because it is harder to inspect and alter the data with LabVIEW.

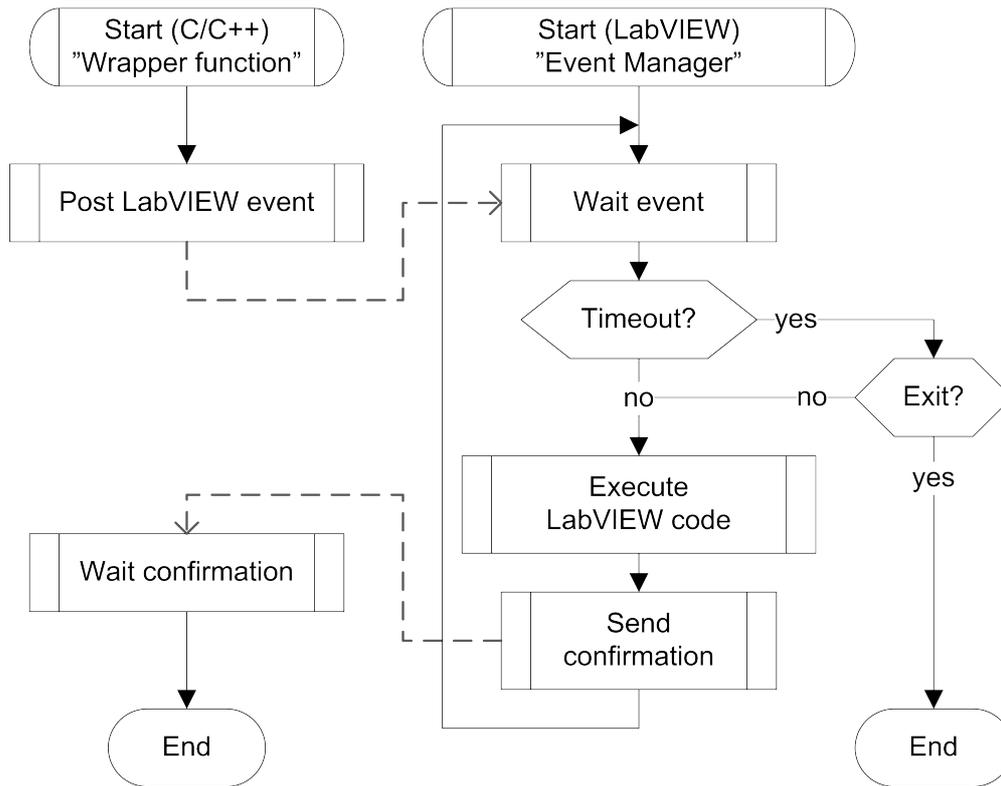
#### 4.4 Callback Functions and LabVIEW

BIF SW Engine uses callback functions to communicate with the SL. The host system must implement these callback functions in order to use the BIF SW Engine. These functions include reading and writing to the control registers of the BIF Master, accepting interrupts from the BIF Master and having access to a sleep function.

Unfortunately the CLN (Call Library Node) from LabVIEW does not support callback functions. Although there is a tab in the CLN for callback functions, it is used to signal status messages from LabVIEW instead of configuring LabVIEW code to be used as callback functions. This is also clarified in the Help of LabVIEW: “You cannot use the Callback tab to pass callback functions as parameters to library functions” (16).

There are some ideas presented in the Help of LabVIEW about how to implement callback functionality. It is suggested to write two new C/C++ functions for the function that requires the callback function as a parameter. The problem with this approach is that the callback function runs in the C/C++ environment and not in LabVIEW. For the purpose of writing and reading I<sup>2</sup>C messages it is required that the callback functions are executed in LabVIEW environment.(16)

To solve this problem, special C/C++ wrapper functions were used. The wrapper functions are used in the place of the callback functions. When executed, the wrapper function elevates the execution to LabVIEW. The general idea for the solution is shown in figure 18.



**FIGURE 18.** The general idea for a LabVIEW callback function

When the BIF Manager calls for a callback function, a special wrapper function is executed. The wrapper function uses a LabVIEW Manager Function PostLVUserEvent to trigger a user event for LabVIEW event manager. After posting the event, the execution of the wrapper function is blocked with a waitForSingleObject function. The waitForSingleObject function is a part of Windows synchronization functions. (18; 11)

When the LabVIEW program receives the user event, it executes the desired operation and stores the results in a temporary transportation variable from the wrapper function. Next, the LabVIEW event manager releases the wrapper function by signalling to the event object that the wrapper function was waiting. After receiving the confirmation signal from LabVIEW, the wrapper function reads the transportation variable and returns the result to the BIF SW Engine.

#### 4.4.1 LabVIEW User Events

LabVIEW User Events can be used to handle asynchronous events. For example, the events sent by the C/C++ wrapper functions are asynchronous because they could be ran in different thread than LabVIEW. An example of User Events in LabVIEW is illustrated in figure 19.

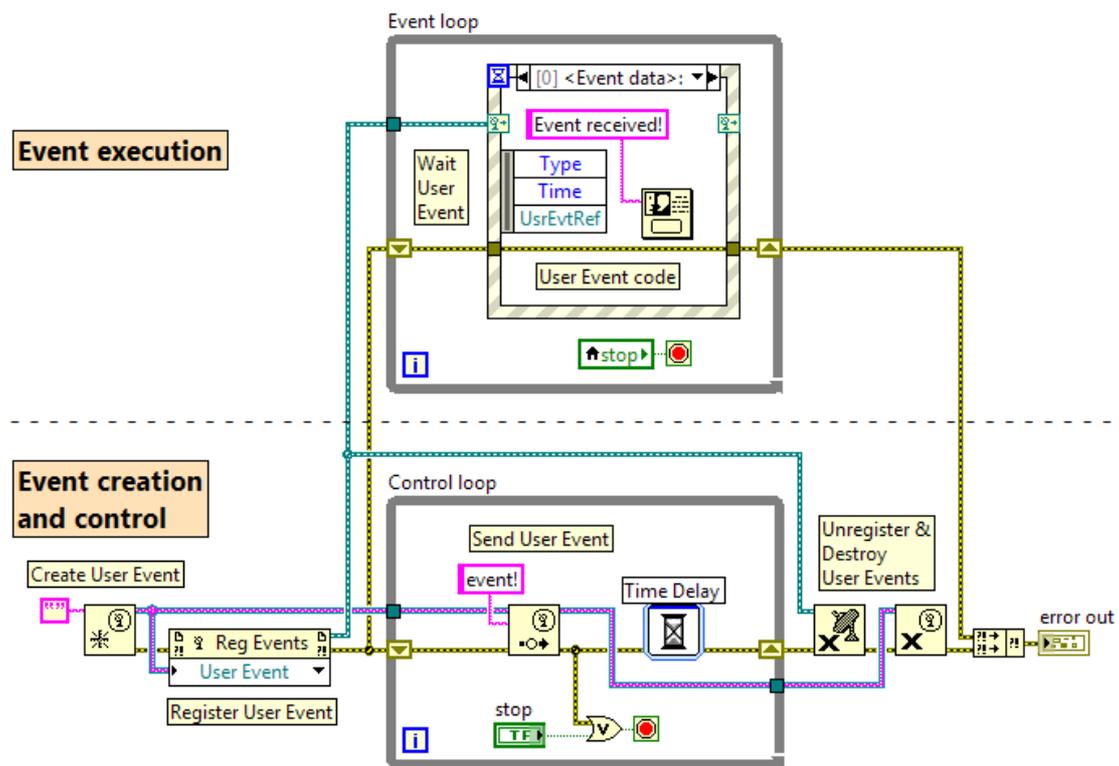


FIGURE 19. An example of how to use User Events in LabVIEW.

In the beginning (see figure 19) the User Event is created and registered. After this, the program execution is divided into two loops. The first loop is an Event loop which waits for events, executes the code and starts to wait for a new event. The second loop is the control loop and it only sends events at certain time intervals. When an event is sent from the control loop, the event loop is triggered to execute. When the program exits, the User Event is first unregistered and then destroyed. After this, the errors are merged and the program execution is stopped.

The example shown in figure 19 does not do anything rational and the way Boolean control is used to stop the execution is non-ideal. However, the example program clearly shows how the user events can be used in LabVIEW.

#### **4.4.2 Windows Event Objects**

Windows operating system has Event Objects that can be used to synchronize execution between different processes. The Event Objects are a part of the synchronization functions in Windows. There can be multiple Event Objects in a system and they are identified by their name. (8)

When programming in C/C++, the Event Objects can be created and opened with CreateEvent function. CreateEvent function can be found from standard Windows.h header by Windows. (5)

When an Event Object already exists, it can be opened locally to be used in the code. This can be done with the OpenEvent function which also is found from Windows.h header. (9)

These Event Objects can also be used from LabVIEW in conjunction with .NET Framework. The class that must be used is called EventWaitHandle class (6). In LabVIEW the .NET Framework can be used with the help of a .NET Constructor block. The EventWaitHandle Class is a part of the System.Threading namespace, under which it can be found in LabVIEW. The use of .NET Constructor block is shown in figure 20.

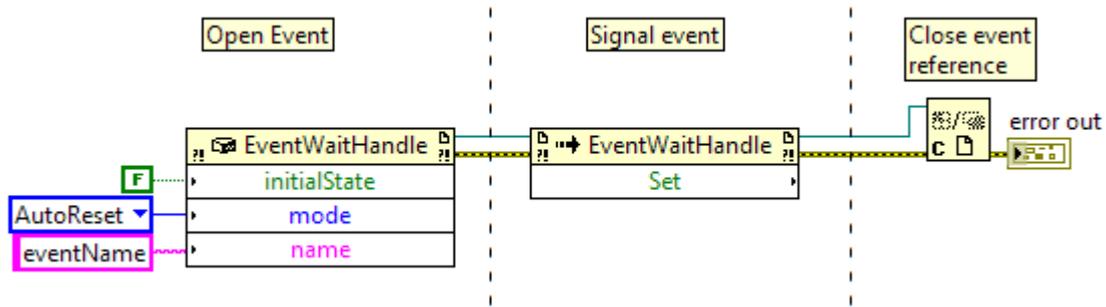
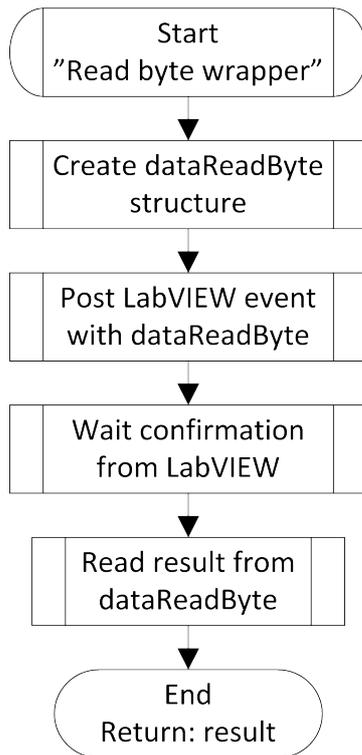


FIGURE 20. Signalling an Event Object from LabVIEW

Using the EventWaitHandle class in LabVIEW is simple and it only requires three parameters. These parameters include the initial state for the Event Object, reset mode and name of the Event Object. Initial state parameter is only used if an Event Object is created. In Wrapper for BIF SW Engine this is not the case because the Event Object is created in C/C++ code. The mode parameter describes the reset behaviour after the event object is signalled. If the reset mode is set to manual, the event stays signalled until cleared. For the purpose used here the automatic reset is desired. Lastly the name parameter describes the name of the Event Object. The event name is a identifier that connects the Event Objects between C/C++ and LabVIEW. After the Event Object has been opened, it can be signalled with a Set method like shown in figure 20.

#### 4.4.3 Read Byte Callback Wrapper Function

With the Event Objects from Windows and User Events from LabVIEW the callback functions from the BIF SW Engine can be wrapped for LabVIEW. The read byte wrapper function is shown in figure 21. Only the read byte wrapper function is presented here because all the remaining wrapper functions are very similar with the only difference being that they have different parameters.



*FIGURE 21. A wrapper function for a read byte callback function*

The read byte wrapper function execution begins with a call from the BIF SW Engine. The BIF SW Engine gives an address as a parameter and expects it to be read by the wrapper function.

The wrapper function stores the address to a transportation structure called `dataReadByte` and sends it with the user event to LabVIEW. In LabVIEW the address is retrieved and read through I<sup>2</sup>C. The result is written to a result buffer pointed by the `dataReadByte` structure. After that, LabVIEW uses `EventWaitHandle` class to signal a confirmation event back to the wrapper function. After receiving the confirmation event, the wrapper function can continue its execution and return the result to the BIF SW Engine.

The `dataReadByte` structure used to transfer data between LabVIEW and the read byte wrapper function is a local variable for the read byte wrapper function. The local variable is usable as long as the function returns a value.

Race conditions while accessing this shared variable should not arise because the wrapper function execution is halted until a confirmation from LabVIEW is received. Also the CLN calls for the BIF SW Engine are synchronized with the error line from LabVIEW.

#### 4.5 Architecture for Wrapped Battery Interface Software Engine

Using the BIF SW Engine in LabVIEW does not require anything else than storing the state machine variables and handling the callback functions. The full architecture the Wrapper for BIF SW Engine wrapper for LabVIEW can be seen in figure 22.

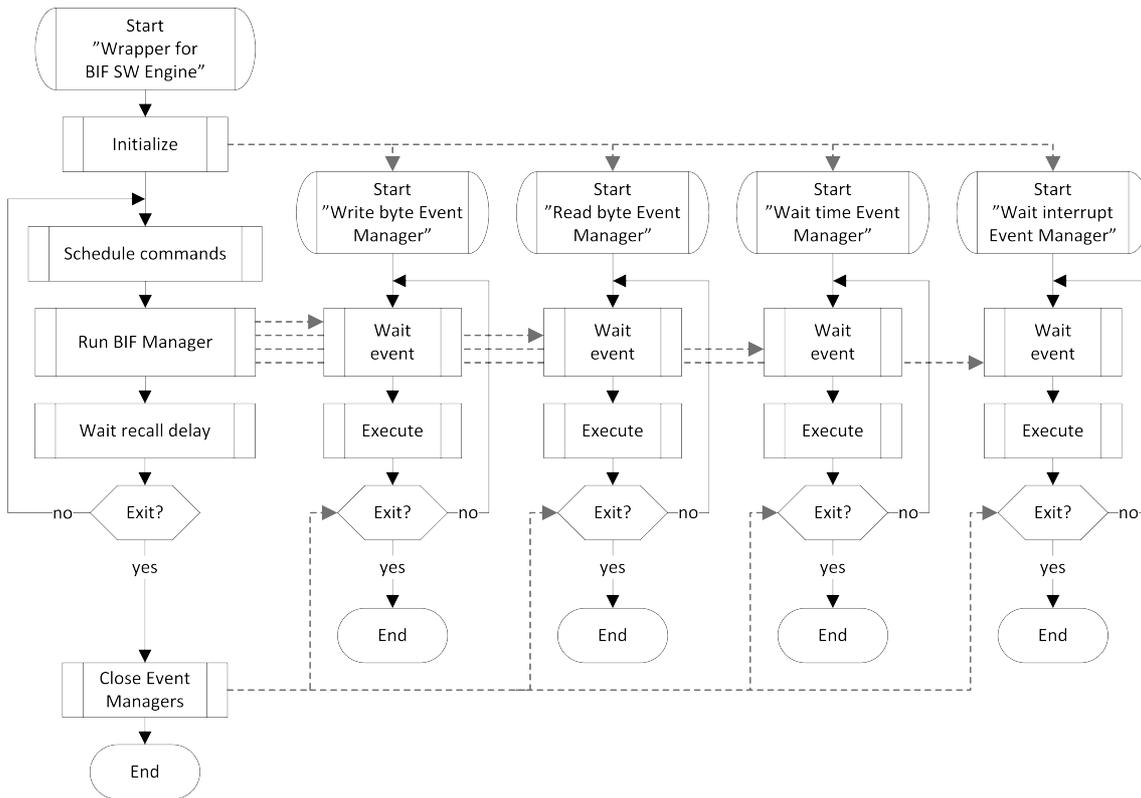


FIGURE 22. A full flowchart for the Wrapper for BIF SW Engine

The program follows the same flowchart that was used for the BIF SW Engine (see figure 17). The difference now is that the callback functions are executed in LabVIEW. Executing the callback functions in LabVIEW requires four different Event Managers to be started in parallel to the BIF SW Engine (see figure 17).

Because the event managers are started asynchronously in parallel to the main LabVIEW program, there needs to be a way to stop them at the end. This can be done with the notification system in LabVIEW. When the main program in LabVIEW is nearing the end, a closeBif.vi SubVI is called. The closeBif.vi SubVI sends stop notifications to all of the asynchronous Event Managers.

In all of the four event managers there is a timeout routine. The timeout routine constantly checks for a “stop” string from the notifier system. If a stop notification is found, the event manager quits.

## 4.6 LabVIEW Interface to Battery Interface Software Engine

Controlling the BIF SW Engine is separated into multiple SubVIs with each one having its own purpose. The LabVIEW interface closely resembles the original BIF SW Engine functions because it is only a wrapper. All the SubVI's used to control LabVIEW wrapper are listed in table 1.

Table 1. LabVIEW interface to the BIF SW Engine

SubVI	Description
initBif.vi	Initializes the BIF SW Engine
commandBif.vi	Schedules commands for the BIF Manager
memoryAllocate.vi	Allocates memory to be used with the commandBif.vi
freeMemory.vi	Frees allocated memory
runBif.vi	Runs the BIF Manager state machine
getRecallDelayBif.vi	Retrieves a recall delay
closeBif.vi	Closes the BIF SW Engine
resetBif.vi	Resets the BIF SW Engine

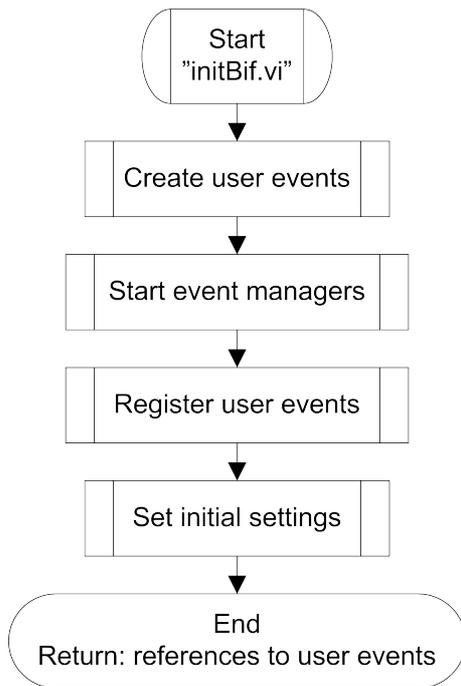
### 4.6.1 Initializing Battery Interface Software Engine

The BIF SW Engine is initialized with the initBif.vi SubVI (see figure 23). The initialization procedure returns a cluster of LabVIEW User Event references. The User Event references must be routed to the closeBIF.vi SubVI in the end.



FIGURE 23. SubVI for initializing the BIF SW Engine

Initialization procedure for the `initBif.vi` SubVI is shown in figure 24. First the User Events are created for the BIF SW Engine (write byte, read byte, wait interrupt and wait time). Next, event managers for all the User Events are started. The User Events are only registered after the event managers are successfully started. Lastly some initial configuration for the BIF Manager is done.



*FIGURE 24. A flowchart for `initBif.vi` SubVI*

#### **4.6.2 Scheduling Commands for Battery Interface Software Engine**

Commands can be scheduled for the BIF SW Engine with the `commandBif.vi` SubVI (see figure 25). The input parameters are routed to the original `bifManagerCommand` function from the BIF SW Engine library. All parameters for this SubVI are listed in table 2.

Table 2. The parameters for commandBif.vi SubVI

Parameter	Type	Description
dataSize	Unsigned 16 bit integer	Size of dataIn or dataOut buffer
dataIn	Pointer	Pointer to dataIn buffer
dataOut	Pointer	Pointer to dataOut buffer
command	Enumerated LabVIEW variable	Specifies the command
batterySlot	Unsigned 16 bit integer	Selected slaves.
parameter	Unsigned 16 bit integer	Parameter for the given command
error in	LabVIEW error	Error in node
error out	LabVIEW error	Error out node

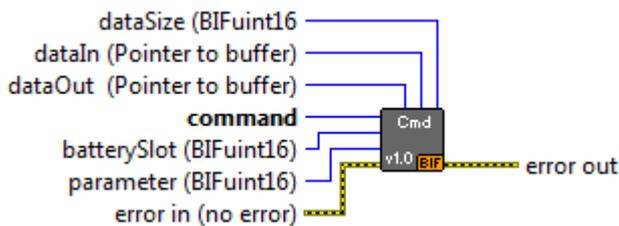


FIGURE 25. The SubVI for scheduling commands

There are two parameters for commandBif.vi which are pointers (dataIn and dataOut) as shown in table 2. Not all of the BIF commands use these buffers, but for those that do, **it is necessary to allocate memory beforehand**. Memory can be allocated with the memoryAllocate.vi SubVI.

Scheduling commands to the BIF Manager can result in errors with improper input parameters. These errors are returned from the DLL as BIFResults which are 32 bit integers. BIFResults are converted to regular LabVIEW errors by translateBifResult.vi SubVI.

### 4.6.3 Allocating Memory for dataIn and dataOut Buffers

Memory allocation requires two parameters (see table 3). The first parameter is sizeof and it tells the amount of bytes one element takes and second parameter is the number of elements which tells the number of elements in the allocated buffer. After the allocated memory is not needed any more, it can be freed with the memoryFree.vi SubVI.

Table 3. The parameters for memoryAllocate.vi SubVI

Parameter	Type	Description
sizeof	Enumerated LabVIEW variable	Size of allocated element
number of elements	32 bit Integer	Amount of allocated elements
error in	LabVIEW error	Error in node
error out	LabVIEW error	Error out node

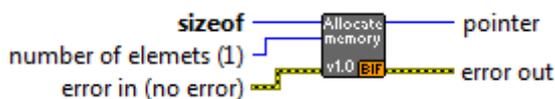


FIGURE 26. The SubVI for allocating memory

The memory allocated by memoryAllocate.vi SubVI is reserved under LabVIEW until it is freed. If the memory is not freed, it will stay reserved until LabVIEW exits. Because of this, it is good practice to free the memory explicitly.

#### 4.6.4 Reading Memory With Call Library Node

Memory can be read with LabVIEW's CLN if the CLN's Library name is set to LabVIEW and the MoveBlock function is selected. The MoveBlock function needs 3 parameters: ps, pd and size (17). The first parameter ps is the pointer to the source from where the bytes will be moved. The second parameter pd is a pointer to the destination. The third parameter size defines the amount of bytes to be moved. For example, there could be a situation where a 16 bit unsigned integer would have to be read from a given address (see figure 27).

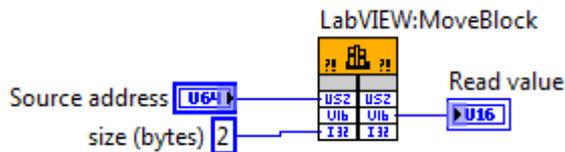


FIGURE 27. Reading a 16 bit unsigned integer from a given address

To read the given memory address the ps parameter in the CLN must be configured as an unsigned pointer sized integer which is then passed as a value. Next, the pd parameter must be configured as a 16 bit unsigned integer passed as a pointer to the value. This effectively makes the CLN pass a pointer instead of the actual value to the MoveBlock function. At the last step, the size parameter is configured to be a 32 bit integer passed as a value.

When the CLN is properly configured, the source address can be wired to the ps input of the CLN. The pd input can be left empty because LabVIEW generates a temporary variable to hold the result. For the size input number 2 should be wired because 16 bits is 2 bytes.

#### 4.6.5 Freeing Allocated Memory

Memory allocated with the `memoryAllocate.vi` SubVI can be freed with `freeMemory.vi` SubVI (see figure 28). The SubVI only takes the memory pointer as a parameter.

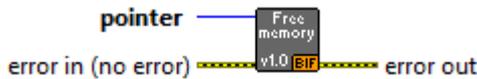


FIGURE 28. The SubVI for freeing allocated memory

#### 4.6.6 Running the Battery Interface Manager State Machine

The BIF Manager state machine can be run with the `runBif.vi` SubVI (see figure 29). The SubVI calls the original `RunBifManagerMachine` function from the BIF SW Engine library. The only difference with the original function and the LabVIEW wrapper function is that the `bifMgrMachineData` parameter is given internally in the DLL by the wrapper function.

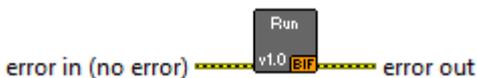


FIGURE 29. The SubVI for running the BIF SW Engine state machine

The `RunBifManagerMachine` function from the BIF SW Engine library returns a `BIFresult` error variable. This `BIFresult` error variable is converted to a LabVIEW error with the `translateBifResult.vi` SubVI.

#### 4.6.7 Retrieving Recall Delay from Battery Interface Software Engine

After every time the BIF Manager state machine is ran, a recall delay is retrieved and the amount of time it describes is waited. The recall delay can be retrieved with the `getRecallDelayBif.vi` SubVI presented in figure 30.

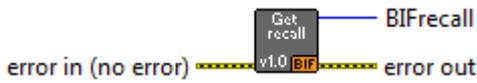


FIGURE 30. The SubVI for querying a recall delay

#### 4.6.8 Closing the Battery Interface Software Engine

In the end the BIF SW Engine can be closed with the closeBif.vi SubVI (see figure 31). For input parameters the closeBif.vi SubVI only takes the cluster of User Event references given by the initBif.vi SubVI.



FIGURE 31. The SubVI for closing the BIF SW Engine

Flowchart for the closeBif.vi SubVI is shown in figure 32. The user event managers are explicitly closed with stop notifications as shown in the flowchart.

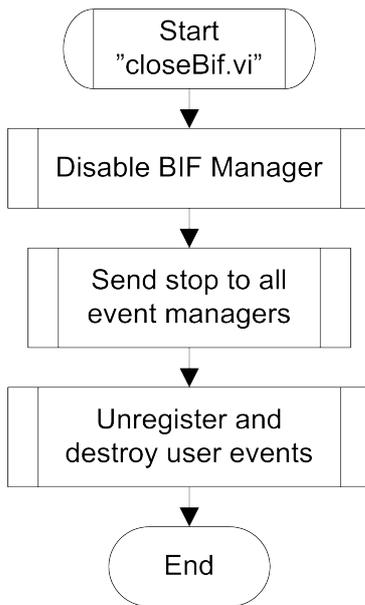


FIGURE 32. The flowchart for closeBif.vi SubVI

#### 4.6.9 Resetting the State Machine

The BIF Manager state machine can be reset with the resetBif.vi SubVI (see figure 33). Resetting the BIF SW Engine does not require any parameters.



FIGURE 33. The SubVI for resetting the state machine

#### 4.6.10 Translating Battery Interface Result into LabVIEW Error

Most of the BIF SW Engines functions return a BIFresult variable. This BIFresult variable contains error information from the function execution. The error information can be merged with the LabVIEW error line through translateBifResult.vi (see figure 34).



FIGURE 34. The SubVI for merging a BIFresult into a LabVIEW error.

## 5 SIMULATED BATTERY INTERFACE SLAVE

The Simulated BIF Slave is located to the bottom of the BIF Master Architecture (see figure 35). Physically the Simulated BIF Slave is located at the DIB that has an FPGA running a Nios II SoPC system. The SoPC system runs a C program which interprets the commands sent through USBTMC by LabVIEW. This C program also controls the Simulated BIF Slave on the DIB.

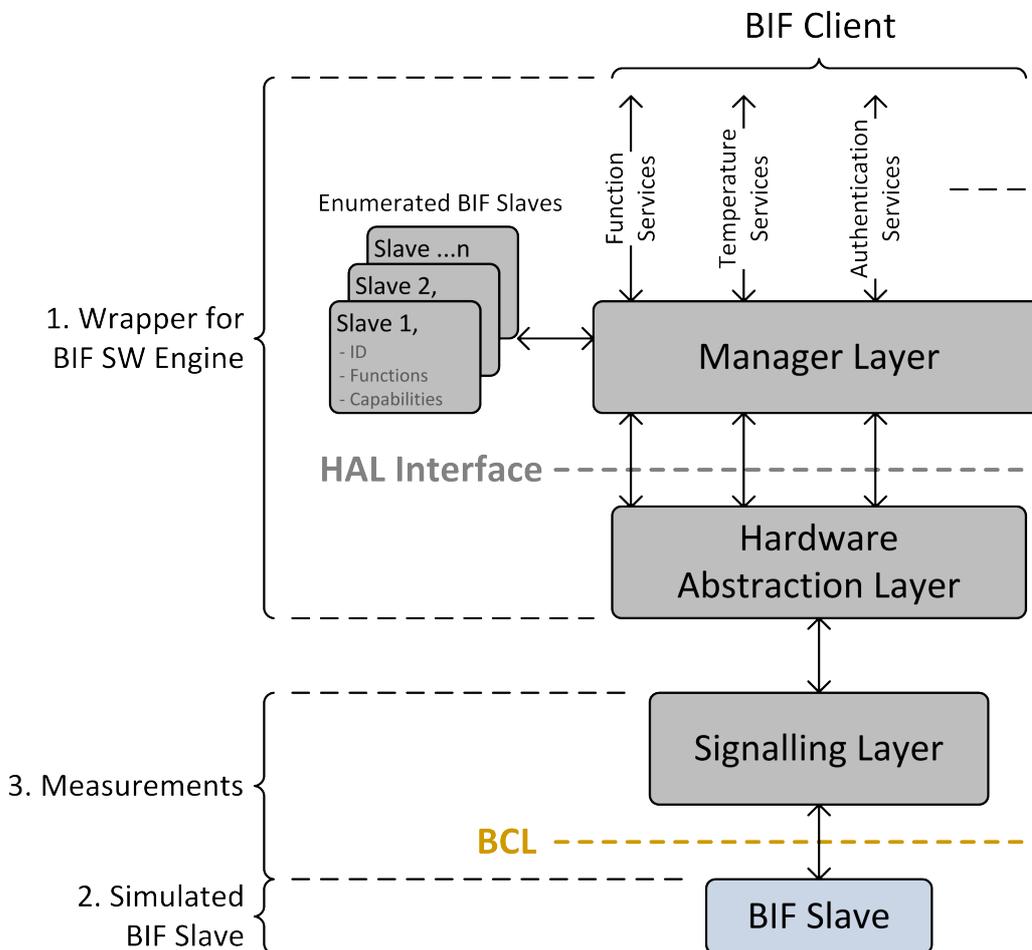


FIGURE 35. Simulated BIF Slave on the BIF Master Architecture.

The Simulated BIF Slave is a key component of the Tester for BIF Master because it is used to generate time critical responses for the BIF Master. The response from the Simulated BIF Slave is used to test the receiver of the BIF Master. A Block diagram for the Simulated BIF Slave is illustrated in figure 36.

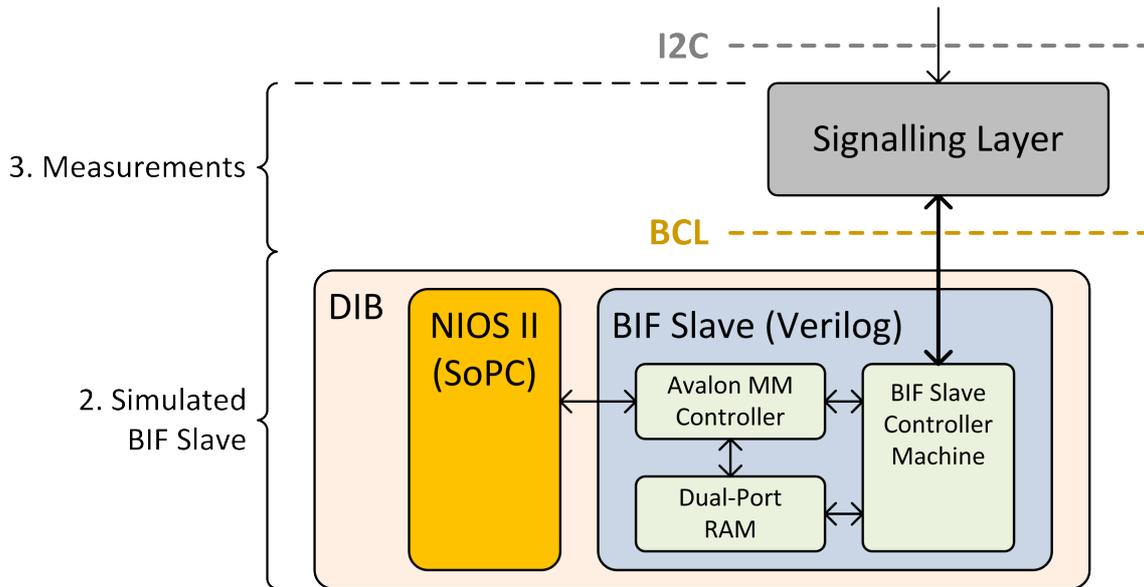


FIGURE 36. A block diagram of the Simulated BIF Slave

The simulated BIF Slave consist of a BIF Slave Controller, a Dual-Port RAM and an Avalon MM (Memory Mapped) Controller. The BIF Slave Controller is a high level state machine for the BIF Slave and it controls the BIF Slave's behaviour.

A single-port RAM was used in the original BIF Slave design but it was replaced with a dual-port version. The dual-port RAM can be accessed simultaneously by the Avalon MM Controller and the BIF Slave Controller Machine.

The Avalon MM Controller is an interface controller between the Nios II and the Simulated BIF Slave. It consists of a register map and control logic. The Simulated BIF Slave can be then controlled through the register map provided for the Nios II system.

## 5.1 Battery Interface Slave Controller

The Simulated BIF Slave is a Verilog design synthesized to the DIB. The Verilog design of the Simulated BIF Slave originates from another project and it has been reused in this thesis. The original BIF Slave was also used for testing a BIF Master. The original BIF Slave Verilog design had to be slightly modified in order to fit in Tester for BIF Master. The block diagram for Simulated BIF Slave is shown in figure 37.

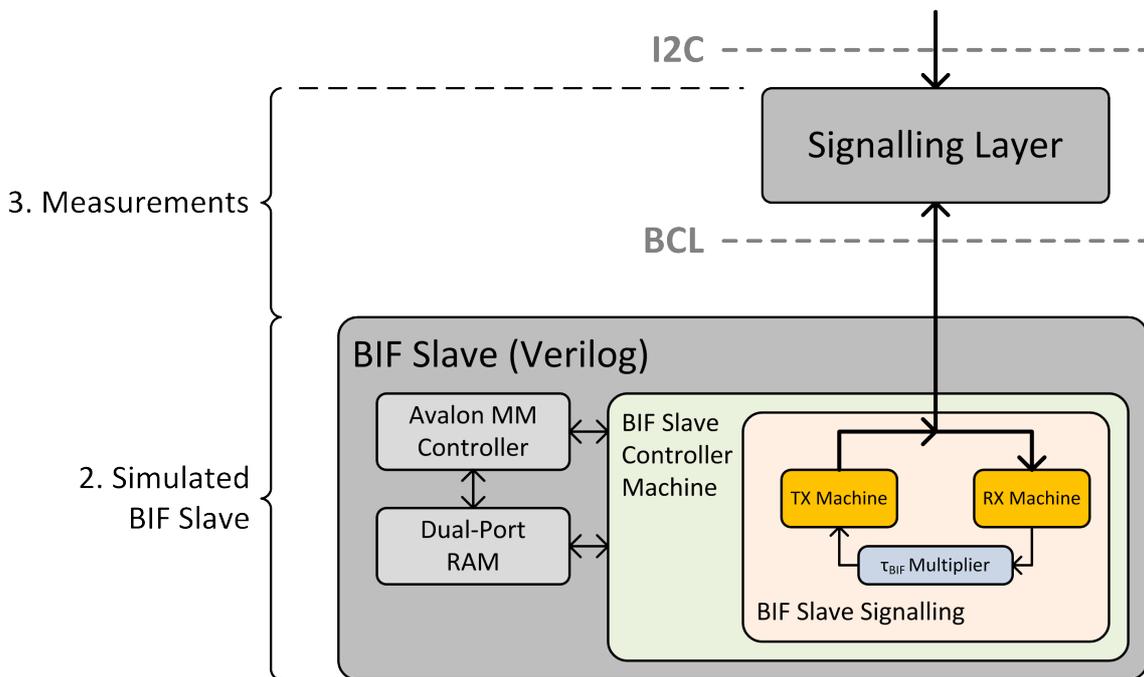


FIGURE 37. A simplified block diagram of the BIF Slave

The BIF Slave used for the Simulated BIF Slave consists of a BIF Slave Controller Machine, a BIF Slave Signalling block and TX & RX Machines (see figure 37). The BIF Slave Controller is a state machine that controls the operation of the BIF Slave at a high level. The BIF Slave Signalling block glues both the TX and RX state machines and some logic to the BIF Slave Controller.

One of the requirements for the Simulated BIF Slave is that the timebase  $\tau_{\text{BIF}}$  of the response can be modified. As was stated earlier, the BIF Master decides the transmission speed with each BIF word sent. The BIF Slave then measures the training sequence from the BIF word and replies with the same timebase. This means that the RX Machine inside the BIF Slave measures the training sequence and makes it available for the TX Machine. To alter the  $\tau_{\text{BIF}}$  a multiplier was added between the RX and TX Machines (see figure 37). The multiplier is configurable by the Avalon MM Controller.

Even with a short  $\tau_{\text{BIF}}$  a fine adjusting resolution was required from the response of the Simulated BIF Slave. To achieve the fine resolution, the clock frequency for the Simulated BIF Slave had to be increased. The clock frequency was increased from 2 MHz to 48 MHz. The 48 MHz clock frequency was chosen because it was the same frequency used by the rest of the SoPC. By using the same clock frequency, no additional synchronization was required for the Avalon MM Interface. Because the clock frequency was 24 times bigger, several counters had to be widened to prevent overflows.

## 5.2 Timebase Multiplier

The  $\tau_{\text{BIF}}$  multiplier was implemented as a Verilog function. The function has input parameters for the multiplier and the measured clock divider representing the  $\tau_{\text{BIF}}$ . With these, the function multiplies the  $\tau_{\text{BIF}}$  for the TX Machine in a range of 0...200%. Because the resulting clock divider can be double in the maximum case, the function returns the clock divider one bit wider. The new MSB (most significant bit) can be used as an overflow indicator because the currently used clock divider counter can already hold a lot bigger value than the maximum  $\tau_{\text{BIF}}$  in the BIF specification.

The function implementation itself is quite simple and contains only one multiplication and a few bit shifts. First the input clock divider is incremented by one to represent the  $\tau_{\text{BIF}}$ . This is done because the offset between the clock divider and  $\tau_{\text{BIF}}$  is 1. Then, the  $\tau_{\text{BIF}}$  is multiplied with the multiplier. With the incrementation the multiplication is done to the  $\tau_{\text{BIF}}$  period and not to the clock divider.

After the multiplication, the quotient and the remainder are separated by bit shifting. The bit shift amount is decided so that with the maximum multiplier the  $\tau_{\text{BIF}}$  quotient is double the input value. This effectively sets the adjustment range between 0...200%. Because of the bit arithmetic, the maximum value is a bit less than the 200% and gets even smaller if the bit widths are increased. The multiplication result is separated in a quotient and a remainder to make the process easier to follow. Before the result is returned, the quotient is rounded to the right direction according the remainder. The rounding also decrements the value by 1 so that the returned value is the clock divider and not the  $\tau_{\text{BIF}}$ .

### **5.3 Avalon Memory Mapped Interface**

In Nios II systems an Avalon Interface is used to connect embedded devices to the processor. Avalon MM Interface is one of the seven Avalon interface types available. The Avalon MM Interface is a memory mapped interface where the control is based on read and write operations to different addresses. (2, p. 1)

The Simulated BIF Slave is configured as an Avalon MM Slave device to the Nios II system. The Avalon MM Controller interprets the Avalon MM Slave signals from the Nios II system. Signals used in the Avalon MM Slave interface are listed in table 4.

Table 4. Signals for an Avalon MM Slave

Signal	Description
clk	Clock
reset_n	Active low reset
address [8:0]	Memory address to be read or written
chip_select	Enables the interface
write	Writes write_data to given address
write_data [31:0]	Write data where only the low byte is used
read_data [31:0]	Read data that updates as soon as the address changes

The clock signal is used to synchronize the communication in the Avalon MM Slave Interface. The reset signal is used to reset the Simulated BIF Slave to a known state. In the address bus, a memory location to be read or written is given. When data is written, the address is prepared beforehand and the write signal is pulsed for one clock cycle. In the write operation only the lowest byte is used. This is because the Nios II processor is 32 bits wide but the register map is 8 bit oriented. If the write\_data bus is forced to 8 bits, automatic interconnection logic is generated by the Avalon Interface. If the interconnection logic is used, a 32 bit wide write operation results in four 8 bit write operations. In the Simulated BIF Slave this issue was solved by using a 32 bit write\_data bus where only the lowest byte was used.

A read operation does not require a read signal because the read\_data is updated after one clock cycle when the address changes. Because of the read\_data update delay, a delay of one clock cycle must also be added to the read operation in the SoPC Component editor at Quartus.

## 5.4 Dual-Port RAM

A Dual-Port RAM is a RAM module with two ports that can be used simultaneously. The Dual-Port RAM shown in figure 36 is an interface between the BIF Slave and the Nios II system. The Simulated BIF Slave uses the RAM for read and write memory operations. With the Dual-Port RAM, the RAM contents of the BIF Slave can be modified through the Avalon MM Controller by LabVIEW. Signals for Dual-Port RAM are shown in table 5.

*Table 5. Signals for the Dual-Port RAM*

Signal	Description
clk	Clock
address_a [7:0]	Port A address
write_a	Port A write signal
write_data_a [7:0]	Port A write data
read_data_a [7:0]	Port A read data
address_b [7:0]	Port B address
write_b	Port B write signal
write_data_b [7:0]	Port B write data
read_data_b [7:0]	port B read data

With this Dual-Port RAM, simultaneous write and read to the same address results in the new value being read, but two simultaneous write operations to the same address results in an undefined condition. Depending on the synthesis, one write is chosen over another. However, a race condition like this should not happen because the only time the Simulated BIF Slave accesses the RAM is when LabVIEW uses the BIF Master to read a value from the RAM. Additionally, when the RAM is configured by the Nios II system, it is done by the request of LabVIEW. This effectively means that the access to the RAM is synchronized by LabVIEW.

## 5.5 Nios II Interface

In the software of the DIB SoPC, there is an infinite loop written in C. The infinite loop interprets the commands sent from LabVIEW with USBTMC. A simple interface between the Nios II and the Simulated BIF Slave was implemented to the existing loop. A simple interface was chosen because the driver logic could be easily implemented in LabVIEW. Also, because every driver function added to C code requires a matching counterpart from LabVIEW.

The interface consists of four commands which are control write, control read, RAM write and RAM read. With these four commands all the control registers and the RAM of the Simulated BIF Slave can be accessed. These four commands require only four interface SubVIs in LabVIEW. The driver logic in LabVIEW uses these four basic SubVIs to control the Simulated BIF Slave.

For example, a control write command is sent as a string through the USBTMC and it is formatted as `bif:ctrl_write:aa:dd` where the `aa` is the address and the `dd` is the data. Every control register available can be written through this command. Both the data and address are in hexadecimal format with zero padding. Registers can be read with a similar command string which is formatted as `bif:ctrl_read:aa`. In the read operation the `aa` is also in hexadecimal form with zero padding. The read operation result is returned through the USBTMC as a hexadecimal string. Controlling the RAM follows the same pattern but the control strings are formatted as `bif:ram_write:aa:dd` and `bif:ram_read:aa`.

## 5.6 Simulating Battery Interface Slave

The original BIF Slave Verilog design was simulated and studied with ModelSim hardware simulator. The Verilog description was short enough to be studied just by reading it. However, in order to test and confirm modifications to the design, a simulator with an appropriate Verilog test bench was used.

All transactions in BIF are initiated by the BIF Master (4, p. 6). This means that an appropriate test vector is required in order to get a valid response from a BIF Slave. This was achieved by extracting the TX state machine from the existing BIF Slave. The TX state machine was then used in the test bench to create test vectors for the BIF Slave. Some example waveforms are shown in figure 38.

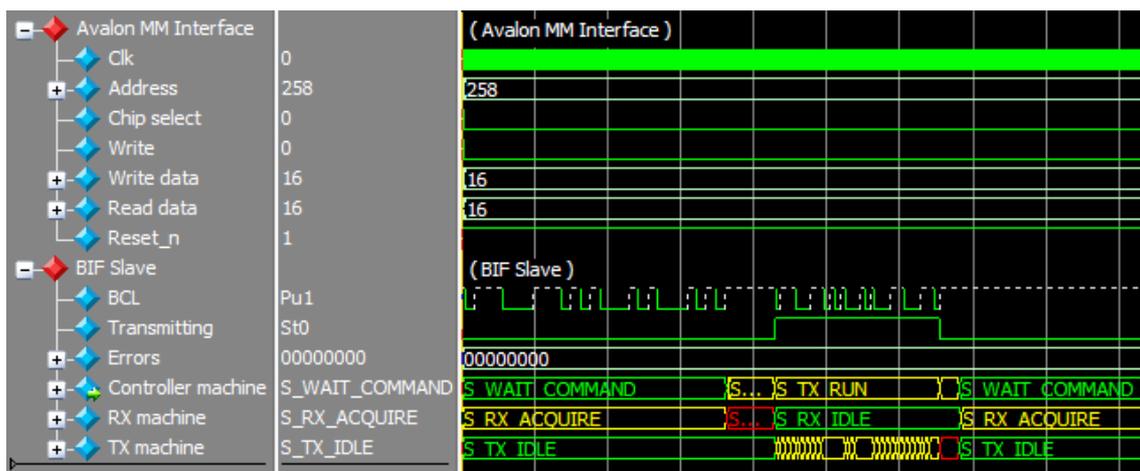


FIGURE 38. A print screen from simulated waveforms in ModelSim

In the waveform window, signals in Avalon MM Interface and the BIF Slave are separated into different groups. The clock signal seen here has a frequency of 48 MHz. A brief description of the figure 38 signals are in table 6.

*Table 6. Descriptions for the signals presented in the print screen*

<b>Signal</b>	<b>Description</b>
Clk	Clock signal for the BIF Slave
Address	Selects control registers or RAM
Chip select	Enables Avalon Interface
Write	Pulses to trigger a write operation
Write data	Write data
Read data	Data read from the address
Reset_n	Active low reset
BCL	Battery Communication Line
Transmitting	Asserts when the Simulated BIF Slave transmits
Errors	Signals various error conditions
Controller machine	Controller machine state
RX machine	RX machine state
TR machine	TX machine state

## 5.7 Control Registers

The Simulated BIF Slave is connected to the Nios II system as an Avalon MM Slave. This means that the BIF slave is visible to rest of the Nios II system as a part of the memory space. The register map of the BIF slave can be seen at table 7.

Table 7. The register map for the Simulated BIF Slave

Addr. [8]	Addr. [7:0]	Name	MSB 7	6	5	4	3	2	1	LSB 0	R/W
0	00	ERRORS	RESERVED	SCALER	STOP	PARITY	BCF	INV	TIMING	ALL	R/reset
0	01	SCALER_HIGH	SCALER_HIGH								R/W
0	02	SCALER_LOW	SCALER_LOW								R/W
0	03	CONTROL	SIGNAL_INT	INT_STATUS	RESET	RESERVED				R/W	
0	04	RX_TAUBIF_HIGH	RX_TAUBIF_HIGH								R
0	05	RX_TAUBIF_LOW	RX_TAUBIF_LOW								R
0	06	STATES1	TX_MACHINE_STATE				RX_MACHINE_STATE				R
0	07	STATES2	RESERVED				CONTROL_MACHINE_STATE				R
0	08..FF	RESERVED	RESERVED								
1	00..FF	BIFRAM									R/W

The memory space used to control the Simulated BIF Slave is 9 bits wide as shown in table 7. The address space is divided into two 8 bit wide parts which are the control registers and the RAM. The MSB of the address is used to select between the control registers and the RAM.

### 5.7.1 Errors Register

All the error signals of the Simulated BIF Slave are collected into ERRORS register shown in table 8. The purpose of the register is to indicate the status of all errors. The status of each error can be read but any write operation to this register will clear the error status of all bits. The reset operation is desired and it is used to clear the errors by LabVIEW. Individual error bit descriptions are listed in table 8.

Table 8. Description for error bits

Register	ERRORS	
Bit	Name	Description
7	RESERVED	Reserved for future use
6	SCALER	Signals overflow if scaled $\tau_{BIF}$ overflows
5	STOP	Stop symbol was too short
4	PARITY	Parity error
3	BCF	Error in BCF
2	INV	Inversion bit error
1	TIMING	General timing error
0	ALL	Is asserted if any errors exist

### 5.7.2 Timebase Multiplier Register

The  $\tau_{BIF}$  multiplier setting can be read and written at the registers shown in table 9 and table 10. In the SCALER\_HIGH register the upper bits are not used by the multiplier and will be discarded when written. These leftover bits will always have a zero value.

Table 9. The control register for the  $\tau_{BIF}$  multiplier's high nibble

Register	SCALER_HIGH	
Bit	Name	Description
7:0	SCALER_HIGH	Bits [11:8] of the $\tau_{BIF}$ multiplier. Overflowing bits will be discarded and will always read 0.

Table 10. The control register for  $\tau_{BIF}$  multiplier's low byte

Register	SCALER_LOW	
Bit	Name	Description
7:0	SCALER_LOW	Bits [7:0] of the $\tau_{BIF}$ multiplier

### 5.7.3 Control Register

Simple interrupt control and reset is available from the control register shown in table 11. Operations can be triggered by writing logical 1 to the appropriate bit. Only the INT\_STATUS bit is readable and others will read 0.

Table 11. The control register for the Simulated BIF Slave

Register	ERRORS	
Bit	Name	Description
7	SIGNAL_INT	Writing to this register triggers interrupt
6	INT_STATUS	The interrupt status for the interrupt query
5	RESET	Writing 1 to this register resets the BIF slave
4:0	RESERVED	Reserved for future use

### 5.7.4 Measured clk\_div

The RX machine measures system clock ticks for every received BIF word. The measured clk\_div can be read from registers shown in table 12 and table 13. The clk\_div is a clock divider for a  $\tau_{BIF}$ . The  $\tau_{BIF}$  value calculation in seconds is explained in chapter 5.8.

Table 12. High bits of the measured clk\_div

Register	RX_TAUBIF_HIGH	
Bit	Name	Description
7:0	RX_TAUBIF_HIGH	Bits [13:8] of the clk_div. Overflowing bits will be dismissed and will always read 0.

Table 13. Low bits of the measured clk\_div

Register	RX_TAUBIF_LOW	
Bit	Name	Description
7:0	RX_TAUBIF_LOW	Bits [7:0] of the clk_div

### 5.7.5 State Machine Registers

The BIF Slave is implemented as digital logic and it is built with state machines. States for the three state machines are visible in the control registers. The state registers provide only a read access and writing to them is ignored. The first register STATES1 (see table 14) contains states for both the TX machine and the RX machine. The possible states for the TX & RX machines are shown in table 15 and table 16.

Table 14. The STATES1 register

Register	STATES1	
Bit	Name	Description
7:4	TX_MACHINE_STATE	The current state of TX machine
3:0	RX_MACHINE_STATE	The current state of RX machine

Table 15. The possible states for the TX state machine

Value	TX Machine State
Hex	Name
0	S_TX_IDLE
1	S_TX_WAIT_0
2	S_TX_WAIT_1
3	S_TX_NEXT
4	S_TX_STOP

Table 16. The possible states for the RX state machine

Value	RX Machine State
Hex	Name
0	S_RX_IDLE
1	S_RX_ACQUIRE
2	S_RX_ANALYZE
4	S_RX_STOP

The STATES2 register (see table 17) contains the state for the Controller machine. The possible states of controller machine are listed in table 18.

Table 17. The STATES2 register

Register	STATES2	
Bit	Name	Description
7:4	RESERVED	Reserved for future use
3:0	CONTROL_MACHINE_STATE	The current state of the Controller machine

Table 18. The Possible states for the Controller state machine

Name	RX Machine State
Bit	Name
0	S_WAIT_COMMAND
1	S_ANALYSE_COMMAND
2	S_WAIT_ALL_READY
3	S_WRITE_TO_RAM
4	S_TX_WAIT_RX_DONE
5	S_READ_FROM_RAM
6	S_TX_RUN
7	S_TX_NEXT
8	S_STANDBY
9	S_INTERRUPT_WAIT
A	S_INTERRUPT_PULSE

## 5.8 LabVIEW Interface

LabVIEW controls the Simulated BIF Slave through the Nios II interface that was discussed in previous chapters. The Simulated BIF Slave is controlled by writing and reading the control registers. Control registers and the RAM of the Simulated BIF Slave can be read or written with the four interface SubVIs that include control write, control read, ram write and ram read functions. The LabVIEW interface (see table 2) is based on these four basic interface SubVIs.

*Table 19. The LabVIEW interface to the Simulated BIF Slave*

SubVI	Description
control_read.vi	Reads the control registers
control_write.vi	Writes the control registers
ram_read.vi	Reads the RAM
ram_write.vi	Writes the RAM
error_get.vi	Reads errors
error_clear.vi	Clears all errors
reset.vi	Resets to a known state
interrupt_trigger.vi	Triggers interrupt from Simulated BIF Slave
interrupt_set_status.vi	Sets the interrupt status
tauBif_get.vi	Reads $\tau_{BIF}$ measured by RX Machine.
tauBif_scaler_get.vi	Reads $\tau_{BIF}$ multiplier
tauBif_scaler_set.vi	Sets $\tau_{BIF}$ multiplier
state_get_controller_machine.vi	Reads the current state for Controller machine
state_get_rx_machine.vi	Reads the current state for RX machine
state_get_tx_machine.vi	Reads the current state for TX machine

### 5.8.1 Reading and Writing Control Register

SubVIs for reading and writing the control register are shown in figure 39 and 40. They are used as a low level communication interface to the control registers in the Simulated BIF Slave. The control\_read.vi SubVI takes an 8 bits wide address as an input parameter and returns an 8 bits wide data as a result. The applicable addresses were discussed in chapter 5.7 Control Registers.

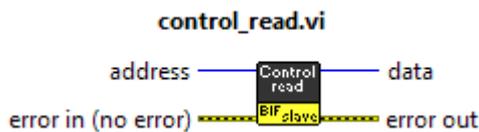


FIGURE 39. The SubVI for reading the control registers

Writing to the control registers is done with control\_write.vi SubVI shown in figure 40. The parameters are very similar to reading but here the address and data ports are inputs.

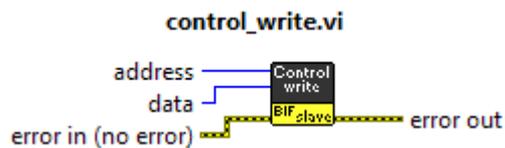


FIGURE 40. The SubVI for writing the control register

### 5.8.2 Reading and Writing Random Access Memory

RAM on the Simulated BIF Slave can be read and written with ram\_read.vi SubVI and ram\_write.vi SubVI (see figures 41 and 42). These two SubVI are also a part of the low level communication interface to the Simulated BIF Slave and they are not used in the test programs as such. The parameters for reading and writing the RAM are exactly same as for the control registers.



FIGURE 41. The SubVI for reading the RAM

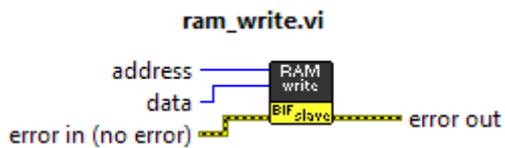


FIGURE 42. The SubVI for writing the RAM

### 5.8.3 Reading and Clearing Errors

Errors from the Simulated BIF Slave can be read with the error\_get.vi SubVI (see figure 43). Reading errors with error\_get.vi SubVI has no parameters and returns a cluster of Boolean variables representing the errors. The errors are translated to Boolean values as defined in table 8.

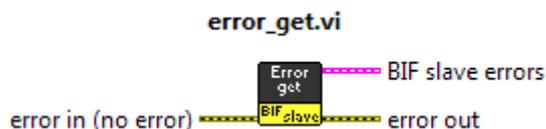


FIGURE 43. The SubVI for reading errors

Errors can be cleared with error\_clear.vi SubVI (see figure 44). The SubVI writes to the error register which effectively clears it. The value written to the error register does not matter because only the address and the write pulse are needed to clear the errors in the Avalon MM Controller.

The error\_clear.vi SubVI has a Boolean input to enable the error clearing. The Boolean input is useful because it can possibly remove an extra case structure from a LabVIEW program. The Boolean input defaults to true, so it can be left empty if is not used.

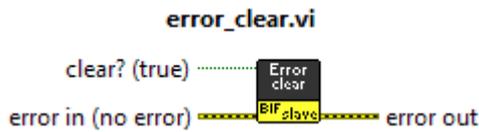


FIGURE 44. The SubVI for clearing errors

### 5.8.4 Reset

The Simulated BIF Slave can be reset to a known state with the reset.vi SubVI (see figure 45). The reset.vi SubVI has a Boolean input that defaults to true for enabling the reset. Resetting the Simulated BIF Slave does not reset the RAM but as it only contains 256 elements, it can be easily cleared with a loop of RAM writes.



FIGURE 45. The SubVI for reset

### 5.8.5 Configuring Interrupts

When a BIF Master enables interrupt mode, all communication on BCL is halted until an interrupt is sensed or the mode is exited. An Interrupt from the Simulated BIF Slave can be triggered with the interrupt\_trigger.vi SubVI (see figure 46). The interrupt\_trigger.vi SubVI has a Boolean enable for the trigger.

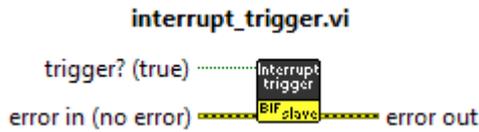


FIGURE 46. The SubVI for triggering an interrupt

After an interrupt is signalled, all slaves are scanned to determine which BIF Slave signalled the interrupt. The reply for the interrupt status query can be set with the interrupt\_set\_status.vi SubVI (see figure 47). The interrupt status must be set before triggering the interrupt.

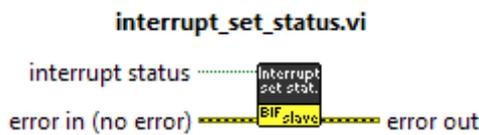


FIGURE 47. The SubVI for settings the interrupt status

### 5.8.6 Controlling Timebase

A BIF Slave updates its  $\tau_{BIF}$  every time a BIF word is received from a BIF Master. The last measured  $\tau_{BIF}$  can be read with tauBif\_get.vi SubVI (see figure 48).

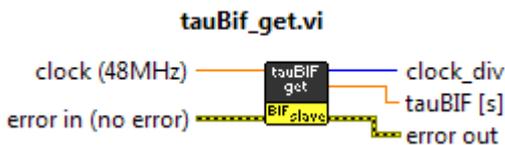


FIGURE 48. The SubVI for reading the  $\tau_{BIF}$

The SubVI returns the  $\tau_{BIF}$  as a clock\_div and in seconds. The SubVI automatically calculates the  $\tau_{BIF}$  in seconds based on the given clock frequency. If the clock parameter is not given, the calculation uses a default clock frequency of 48 MHz. The time constant  $\tau_{BIF}$  can be calculated by using equation 1.

$$\tau_{\text{BIF}} = \frac{\text{clock\_div} + 1}{f_{\text{clock}}} \quad (1),$$

where

clock\_div is the clock divider for  $\tau_{\text{BIF}}$  and

$f_{\text{clock}}$  clock frequency for Simulated BIF Slave

### 5.8.7 Reading Timebase Multiplier

The current  $\tau_{\text{BIF}}$  multiplier can be read with tauBif\_scaler\_get.vi SubVI (49). The SubVI returns the value in percents and as the  $\tau_{\text{BIF}}$  multiplier.

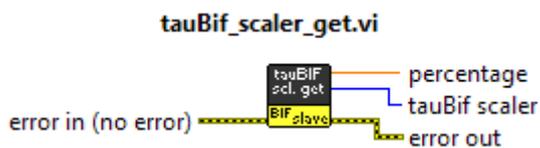


FIGURE 49. The SubVI for reading the  $\tau_{\text{BIF}}$  multiplier

The 12-bit  $\tau_{\text{BIF}}$  multiplier can be turned into percents with equation 2.

$$\text{percentage} = \frac{\tau_{\text{BIF\_MULTIPLIER}}}{20,48} \quad (2),$$

where

$\tau_{\text{BIF\_MULTIPLIER}}$  is the current 12-bit  $\tau_{\text{BIF}}$  multiplier

### 5.8.8 Writing Timebase Multiplier

The  $\tau_{\text{BIF}}$  multiplier can be written with the tauBif\_scaler\_set.vi SubVI (see figure 50). The SubVI has one input parameter called percentage. The percentage must be wired with a LabVIEW double type variable in the range of 0-200. This percentage sets the response timebase of the Simulated BIF Slaves compared to the last received BIF word.



FIGURE 50. The SubVI for writing the  $\tau_{BIF}$  Multiplier

### 5.8.9 Reading State Machine States

The state machine states of the Simulated BIF Slave are routed to the control registers. These states can be used to help debugging if a problem arises. States of each state machine can be read with the `state_get_controller_machine.vi` SubVI, the `state_get_rx_machine.vi` SubVI and the `state_get_tx_machine.vi` SubVI (see figures 51, 52 and 53).

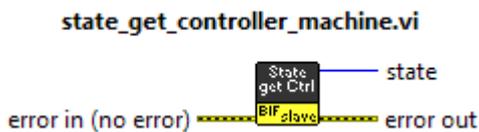


FIGURE 51. The SubVI for reading the controller state machine state



FIGURE 52. The SubVI for reading the RX state machine state

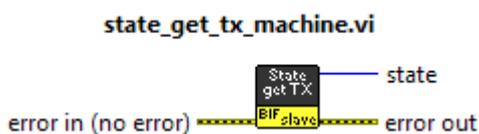
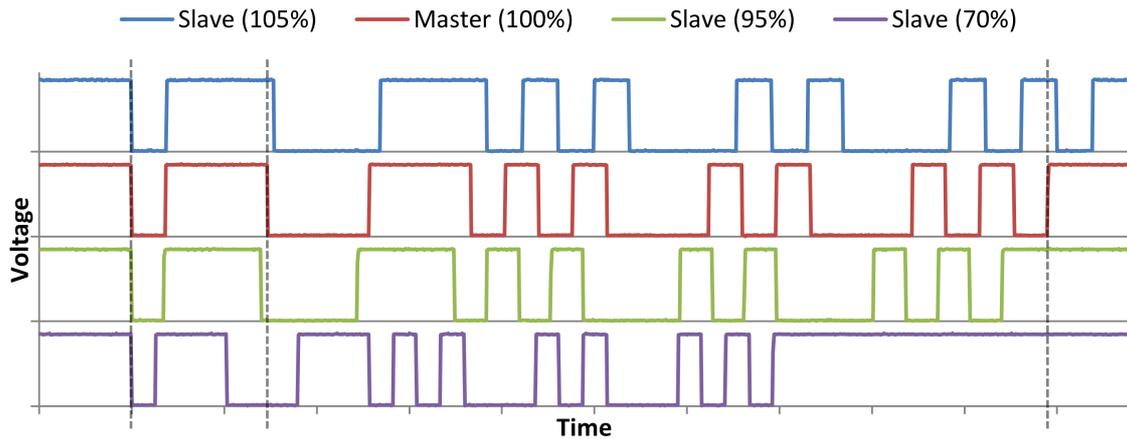


FIGURE 53. The SubVI for reading the TX state machine state

The states for all the SubVIs are returned as enumerated LabVIEW variables. The state machine states are translated into enumerated LabVIEW variable as defined in tables 18, 16 and 15.



It is difficult to see the timebase difference of the Master and the Slave in figure 54 waveform. But if the Master and the Slave words are extracted from the waveform, and the time origin of the words are aligned to start from zero, it is easier to inspect the response. Several response words of the Simulated BIF Slave are shown in figure 55. The red waveform in figure 55 is the read request sent by the BIF Master.



*FIGURE 55. Various  $\tau_{BIF}$  responses from the Simulated BIF Slave compared to the read request from the BIF Master*

At the top of figure 55 there is a Slave response with 105%  $\tau_{BIF}$  compared to the 100%  $\tau_{BIF}$  of the BIF Master. Under the BIF Master's waveform there are two Slave responses with the  $\tau_{BIF}$ s being 95% and 70%. From these three results only the 70% case created an error in the BIF Master, which is an expected result. The Slave response timebase deviation of 70%  $\tau_{BIF}$  is clearly outside of the BIF specification limits (14, p. 77).

Next, the  $\tau_{BIF}$  was swept across from 70% to 130% with a step of 1%. With every step the BIF Master reads a value from the Simulated BIF Slave. After the read operation, the BIF Master was checked for errors. An array of errors was created for the full  $\tau_{BIF}$  sweep.

The sweep was then done multiple times and the resulting error arrays were summed column-wise. The sum array was then divided by the maximum value found in the array. This effectively scales the results between 1 and 0. The scaled sum array represents the probability of a successful transmission at the given  $\tau_{BIF}$ . An example graph with a slow 32kHz clock is given in figure 56.

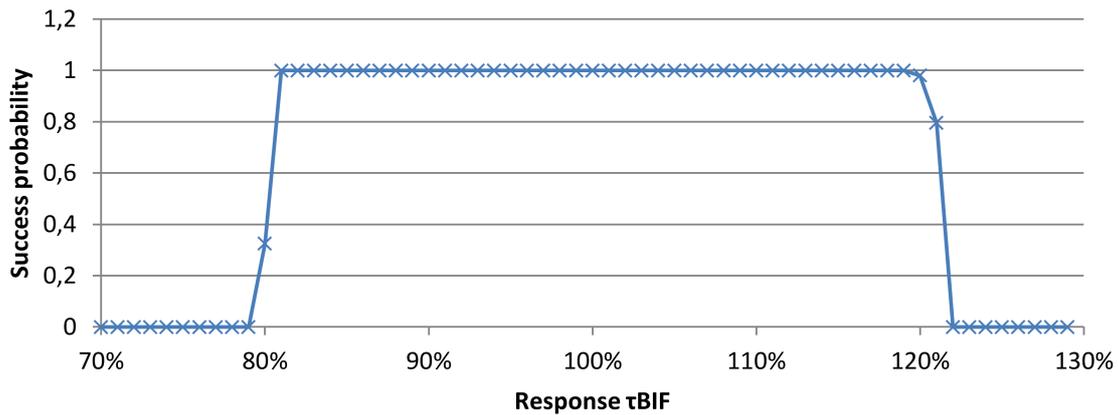


FIGURE 56. The success probability of the BIF Master's receiver with  $\tau_{BIF}$  Sweep

The  $\tau_{BIF}$  sweep was done with only one  $\tau_{BIF}$  period at one temperature. At the next phase, this test would be performed over all possible  $\tau_{BIF}$ s, temperatures and process corners. Testing the BIF Master with all these parameters guarantees that the BIF Master operates correctly in all required conditions.

## 6.2 Temperature Measurement with Real BIF Slaves

The high level control of BIF SW Engine was tested with some real BIF Slaves. These BIF Slaves had a temperature function implemented. The temperature function was used with the BIF SW Engine to measure the temperature of the different BIF Slaves. In this test case the measured temperature was the ambient, but it in a real use case it could be the battery pack temperature. A pseudo code of the temperature demo is shown in figure 57.

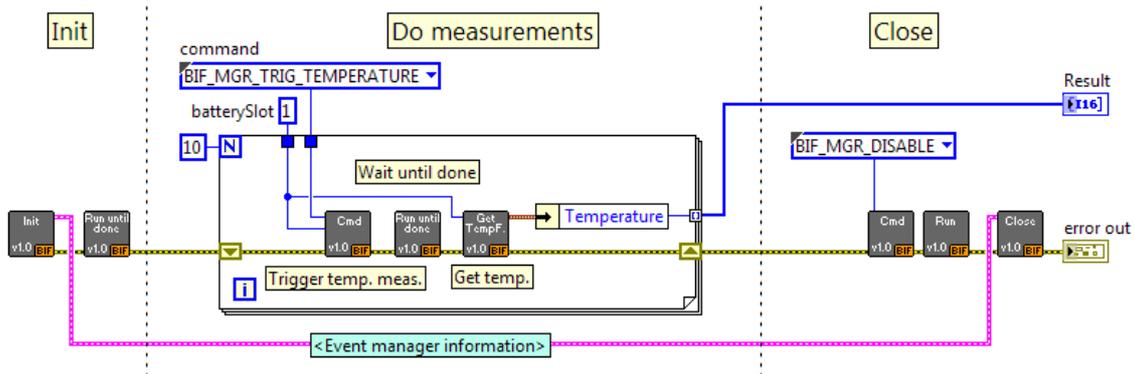


FIGURE 57. A pseudo code of the Temperature demo

The test begins with the BIF SW Engine initialization with certain SubVIs as shown in figure 57. Then, the temperature measurement is triggered and the measurement program waits for completion. After the BIF SW Engine asserts its ready flag, the temperature cluster is retrieved and the temperature is extracted. The measurement is repeated 10 times and the result is delivered to the Result indicator.

In the actual LabVIEW temperature measurement demonstration a value of 111<sub>B</sub> was used for the batterySlot value to trigger the temperature measurement in all three of the BIF Slaves. A loop was used to retrieve the individual temperatures from the three slaves (see figure 58). The actual LabVIEW program also contains some plotting logic in order to present all the measured temperatures in a real time plot. The looping and plotting logic are not shown here because they do not belong to the actual BIF SW Engine.

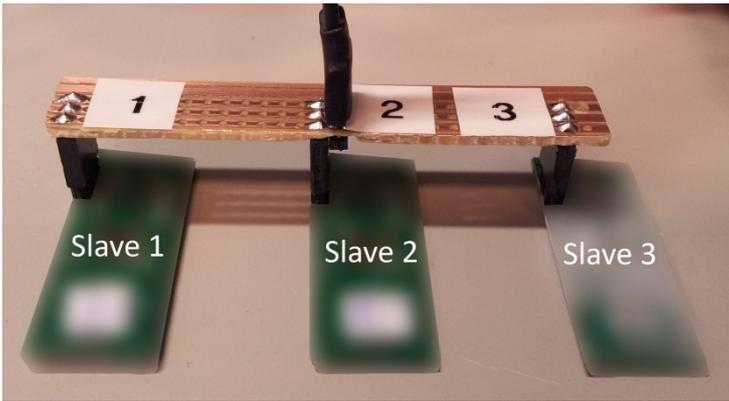


FIGURE 58. The BIF Slaves used for the temperature measurement (the image is blurred intentionally)

An example plot of the temperature measurement is shown in figure 59. The dips in the temperature are caused by cold spray that was applied on each of the Slave devices during the temperature measurement session.

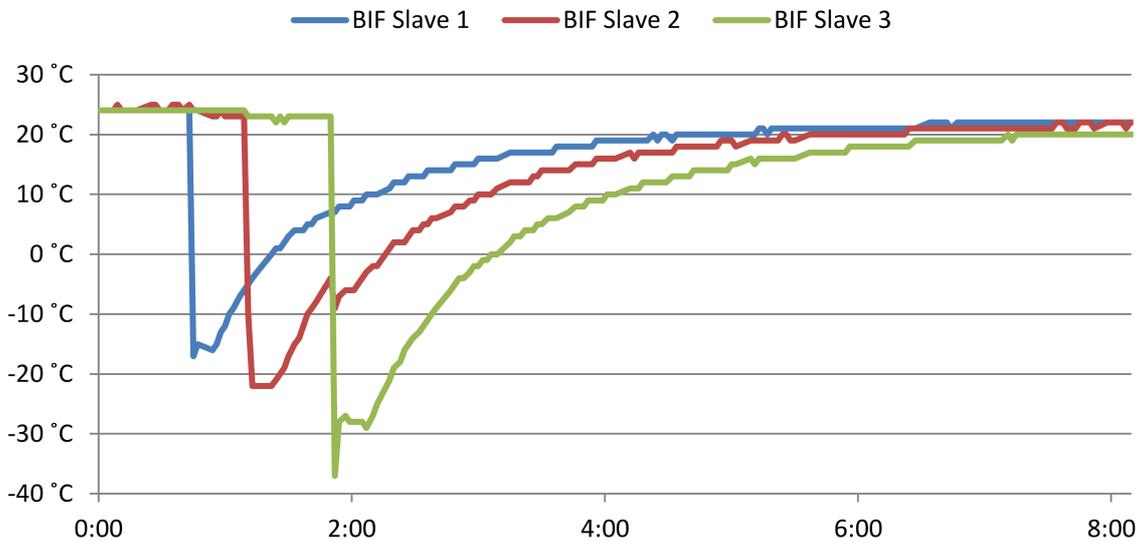


FIGURE 59. The temperature measurement

## **7 THOUGHTS AND CONCLUSIONS**

The purpose of this thesis was to create a test environment for a BIF Master. The test environment was required to be able to send timebase skewed responses to the BIF Master. The test environment was also required to be able to reproduce real world use cases. The Tester for BIF Master was separated into three parts: Wrapper for BIF SW Engine, Simulated BIF Slave and Measurements.

### **Wrapper for Battery Interface Software Engine**

The Wrapper for BIF SW Engine is the part that can be used to replicate real world use cases. In the beginning it was not perfectly clear how the BIF SW Engine should be wrapped into LabVIEW. However, the chosen DLL method proved to be a good solution. This is because the BIF SW Engine itself did not require any modifications in order to be compiled as a DLL. However, the callback functionality used by the BIF SW Engine required more improvised implementation of LabVIEW code in order to work. After all, the Wrapper for BIF SW Engine turned out as a clean and working implementation without significant compromises, and the designed callback wrapper functions perform very well.

### **Simulated Battery Interface Slave**

The Simulated BIF Slave is the part that creates the timebase skewed responses for BIF Master. The work for the Simulated BIF Slave begun with an existing BIF Slave Verilog design that had to be modified in order to support the new features required. As a language the Verilog was new to me but as I had studied VHDL on my own before the thesis, learning Verilog was easier. Mainly two modifications had to be done for the BIF Slave. Firstly, a finer resolution for the Simulated Slave timing was needed and therefore the clock frequency had to be increased significantly from the original 2MHz. Secondly, a multiplier was added between the RX & TX state machines. After Verilog code modifications, the Simulated BIF Slave worked as planned on the first go after synthesis.

## **Measurements**

The whole system was integrated and test measurements were done with both the Simulated BIF Slave and the real BIF Slaves through the BIF SW Engine. This part could have been more extensive but I am happy that we were able to squeeze it in and the whole system could be tested. The integration and measurement step provided important feedback from the work that was done in the previous steps.

The whole system works as expected and it can be used by ST-Ericsson for future development. The test environment can be used as a standalone system or integrated on the ATE (Automatic Test Environment) testers if needed.

## 8 LIST OF REFERENCES

1. Altera. 2008. Cyclone II Device Handbook, Volume 1. Date of retrieval 1.4.2013, [http://www.altera.com/literature/hb/cyc2/cyc2\\_cii51001.pdf](http://www.altera.com/literature/hb/cyc2/cyc2_cii51001.pdf).
2. Altera. 2011. Avalon Interface Specification. Date of retrieval 10.4.2013, [http://www.altera.com/literature/manual/mnl\\_avalon\\_spec.pdf](http://www.altera.com/literature/manual/mnl_avalon_spec.pdf).
3. IEC and IEEE. 2004. Higher Performance Protocol for the Standard Digital Interface for Programmable Instrumentation-Part 1: General. Date of retrieval 16.4.2013, <http://ieeexplore.ieee.org/servlet/opac?punumber=9646>.
4. Littow, M., Chun, S. & Schaecher, S. 2012. BIF Whitepaper. Date of retrieval 31.3.2013, [http://www.mipi.org/sites/default/files/BIF\\_whitepaper\\_020112\\_final.pdf](http://www.mipi.org/sites/default/files/BIF_whitepaper_020112_final.pdf).
5. Microsoft Developer Network. 2012. CreateEvent function. Date of retrieval 2.4.2013, [http://msdn.microsoft.com/en-us/library/windows/desktop/ms682396\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms682396(v=vs.85).aspx).
6. Microsoft Developer Network. 2012. EventWaitHandle Class. Date of retrieval 2.4.2013, [http://msdn.microsoft.com/en-us/library/system.threading.eventwaithandle\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/system.threading.eventwaithandle(v=vs.85).aspx).
7. Microsoft Developer Network. 2012. Image has Safe Exception Handlers. Date of retrieval 11.4.2013, [http://msdn.microsoft.com/en-us/library/9a89h429\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/9a89h429(v=vs.110).aspx).

8. Microsoft Developer Network. 2012. Interprocess Synchronization. Date of retrieval 2.4.2013, [http://msdn.microsoft.com/en-us/library/windows/desktop/ms684123\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms684123(v=vs.85).aspx).
9. Microsoft Developer Network. 2012. OpenEvent function. Date of retrieval 2.4.2013, [http://msdn.microsoft.com/en-us/library/windows/desktop/ms684305\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms684305(v=vs.85).aspx).
10. Microsoft Developer Network. 2012. Struct Member Alignment. Date of retrieval 11.4.2013, [http://msdn.microsoft.com/pt-br/library/xh3e3fd0\(v=vs.110\).aspx](http://msdn.microsoft.com/pt-br/library/xh3e3fd0(v=vs.110).aspx).
11. Microsoft Developer Network. 2012. WaitForSingleObject function. Date of retrieval 15.4.2013, [http://msdn.microsoft.com/en-us/library/windows/desktop/ms687032\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms687032(v=vs.85).aspx).
12. Microsoft Developer Network. 2012. When to Precompile Source Code. Date of retrieval 11.4.2013, [http://msdn.microsoft.com/en-us/library/9d87zb00\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/9d87zb00(v=vs.71).aspx).
13. MIPI Alliance. 2013. Battery Interface Working Group. Date of retrieval 24.4.2013, <http://www.mipi.org/working-groups/battery-interface>.
14. MIPI Alliance. 2012. Specification for Battery Interface. MIPI Alliance internal document.
15. MIPI Alliance. 2012. Specification for BIF Hardware Access Layer. MIPI Alliance internal document.

16. National instruments. 2011. LabVIEW 2011 Help: Configuring the Call Library Function Node. Date of retrieval 1.4.2013, [http://zone.ni.com/reference/en-XX/help/371361H-01/lvexcodeconcepts/configuring\\_the\\_clf\\_node/](http://zone.ni.com/reference/en-XX/help/371361H-01/lvexcodeconcepts/configuring_the_clf_node/).
17. National Instruments. 2011. LabVIEW 2011 Help: MoveBlock (LabVIEW Manager Function). Date of retrieval 4.4.2013, <http://zone.ni.com/reference/en-XX/help/371361H-01/lvexcode/moveblock/>.
18. National Instruments. 2011. PostLVUserEvent (LabVIEW Manager Function). Date of retrieval 15.4.2013, <http://zone.ni.com/reference/en-XX/help/371361H-01/lvexcode/postlvuserevent/>
19. NXP. 2012. I<sup>2</sup>C-bus specification and user manual. Date of retrieval 1.4.2013, [http://www.nxp.com/documents/user\\_manual/UM10204.pdf](http://www.nxp.com/documents/user_manual/UM10204.pdf).
20. USB Implementers Forum. 2003. Universal Serial Bus Test and Measurement Class Specification (USBTMC). Date of retrieval 1.4.2013, [http://www.usb.org/developers/devclass\\_docs/USBTMC\\_1\\_006a.zip](http://www.usb.org/developers/devclass_docs/USBTMC_1_006a.zip)
21. Wikipedia. 2013. Hamming code. Date of retrieval 31.3.2013, [http://en.wikipedia.org/wiki/Hamming\\_code](http://en.wikipedia.org/wiki/Hamming_code).
22. Wikipedia. 2013. I<sup>2</sup>C. Date of retrieval 1.4.2013, <http://en.wikipedia.org/wiki/I2C>.
23. Wikipedia. 2013. IEEE-488. Date of retrieval 1.4.2013, <http://en.wikipedia.org/wiki/IEEE-488>.
24. Wikipedia. 2013. Process corners. Date of retrieval 31.3.2013, [http://en.wikipedia.org/wiki/Process\\_corners](http://en.wikipedia.org/wiki/Process_corners).

25. Wikipedia. 2013. Virtual Instrument Software Architecture. Date of retrieval  
1.4.2013,  
[http://en.wikipedia.org/wiki/Virtual\\_Instrument\\_Software\\_Architecture](http://en.wikipedia.org/wiki/Virtual_Instrument_Software_Architecture).

## Compilation Guide for Battery Interface Software Engine

The BIF SW Engine can be compiled with Visual Studio by Microsoft. This compilation guide illustrates the steps that are needed in order to recreate and compile the Wrapper for BIF SW Engine Visual Studio 2012 project. The guide begins with an explanation how to create a new project for Visual Studio and then continues with configuring the project for compilation.

### Creating New Project

The compilation process begins by creating a new project named BIF SW Engine for the Wrapper for BIF SW Engine. Other names can also be used, but then the wrapper source files BIF SW Engine.c and BIF SW Engine.h must be named accordingly. The project type for the Wrapper for BIF SW Engine should be selected as Win32 Project under the Visual C++ category (see figure 60).

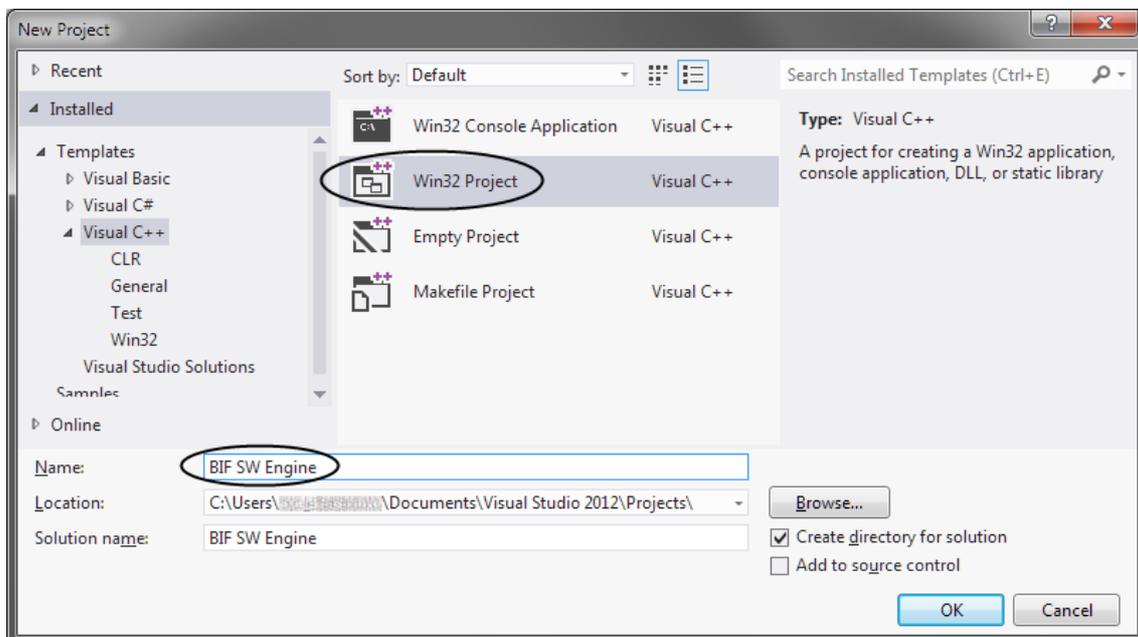
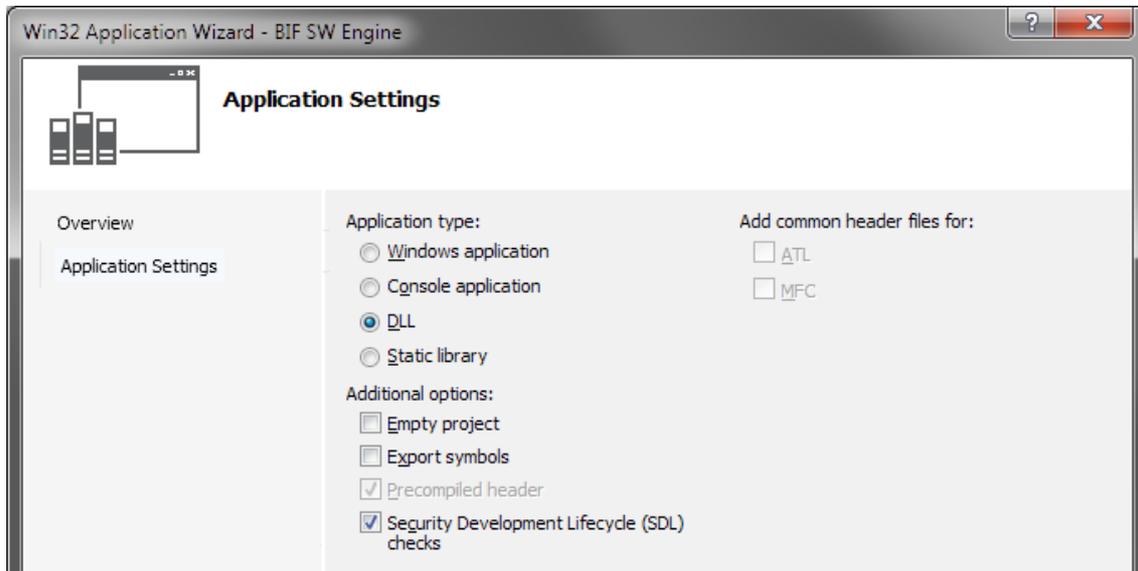


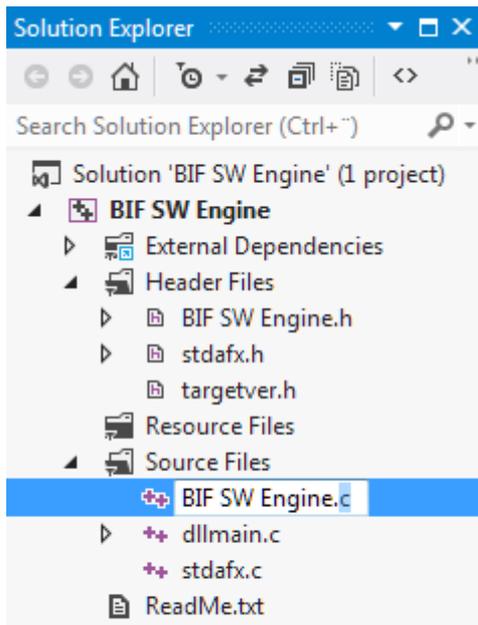
FIGURE 60. Selecting the project type

After selecting the project type, a configuration screen for the application type (see figure 61) comes into view. From this screen the DLL option should be selected. No other changes are required.



*FIGURE 61. Selecting the application type*

Next, Visual Studio does some initial project set-up. After that, the wrapper source files BIF SW Engine.c and BIF SW Engine.h need to be copied to the local project directory. The project directory was shown in figure 60. The copied files must also be added to the project. This can be done by right clicking Source Files/Header Files folder and selecting Add Existing Item. After this, the files stdafx.cpp and dllmain.cpp must be renamed as .c files (see figure 62).



*FIGURE 62. Renaming the file types to .c*

Next, the source files regarding the BIF SW Engine can be added to the project. The BIF SW Engine files should be added to Filter folders like shown in figure 63. Filter folders can be created by right clicking the host folder and selecting the Add New Filter option. The BIF SW Engine files do not have to exist in the local project directory and they can be added from a remote location.

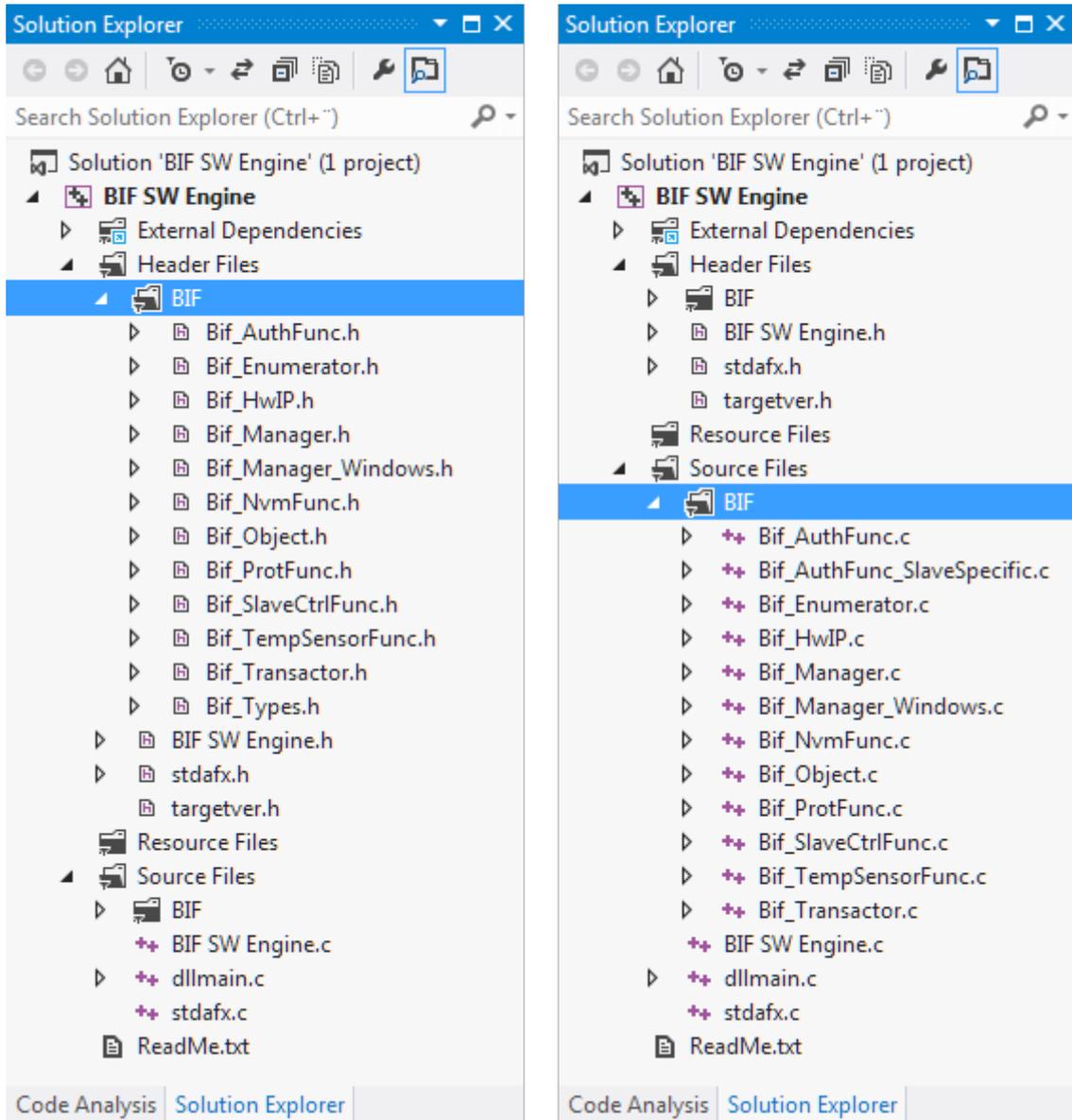


FIGURE 63. The BIF SW Engine files added to the project

### Configuring Project Settings

After the project is created and all the required files are added, some configuration must be done before the Wrapper for BIF SW Engine can be compiled. All the required configurations can be done from the project properties. Project properties can be accessed by right clicking the BIF SW Engine from the solution explorer and selecting the properties option. Before making any adjustments, the Configuration drop box must be adjusted so that all build configurations are included (see figure 64).

First some include directories must be added to the compiler. These additional directories can be added like shown in figure 64.

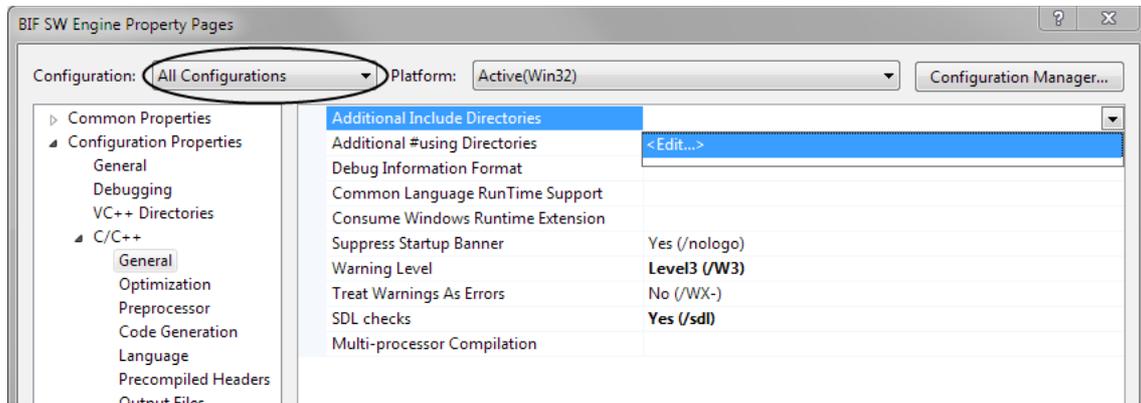


FIGURE 64. The changes for project properties

There are three additional include directories that must be added. The first one is the LabVIEW's cintools folder which contains headers for LabVIEW related functions. The second and third ones are the two BIF SW Engine source file folders that must be added. The BIF SW Engine contains one folder for headers called incl and one folder for sources called src. All the required include directories with example paths are shown in figure 65.

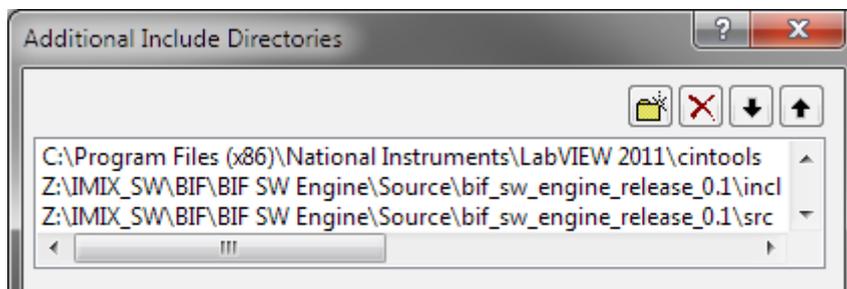


FIGURE 65. The additional Include Directories

Next, the labviewv.lib must be linked to the project in order to use the LabVIEW functions. The labviewv.lib can be added to the Additional Dependencies list which is shown in figure 66.

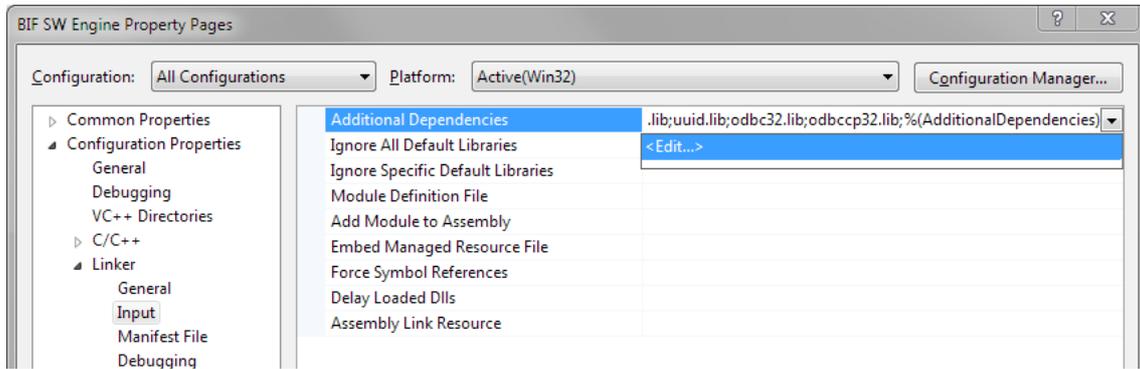


FIGURE 66. Additional Dependencies for the linker

After selecting the <Edit...> option from the Additional Dependencies, a window shows up where these additional dependencies can be added (see figure 67). Figure 67 also contains an example path for the labviewv.lib. For some reason the Visual Studio does not provide an explorer to add the directories and they must be typed or copied by hand.

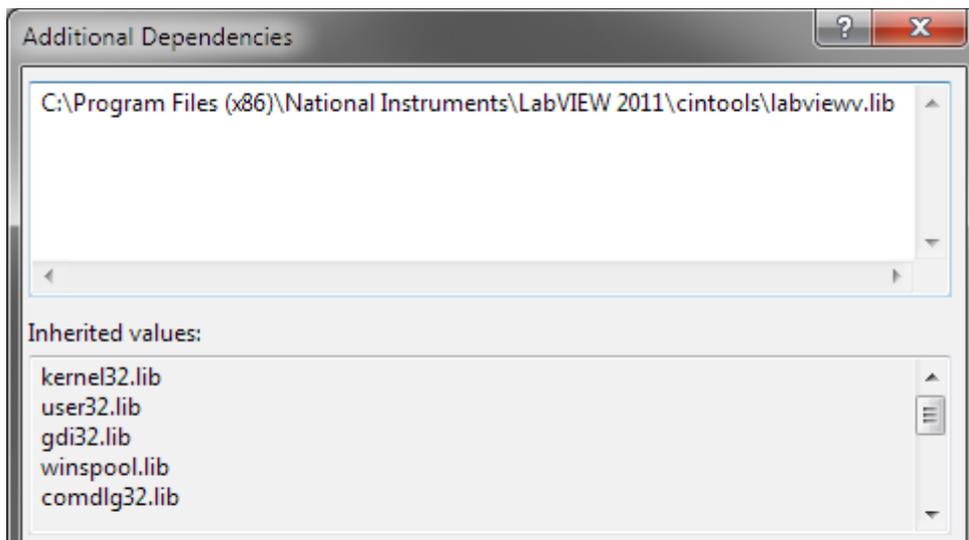


FIGURE 67. The additional Dependencies for the linker

Next, the way Visual Studio stores structures in memory must be changed. By default the structure data alignment is set to 8-bytes which means that extra padding will be added to structure items that are smaller than 8-bytes (10). This is important because the BIF SW Engine returns results as pointers to structures. These pointers are then read byte by byte with LabVIEW’s MoveBlock function. If the structure contains padding, the result will be misaligned and corrupted. Structure padding can be disabled by selecting the Struct Member Alignment as 1 Byte (see figure 68).

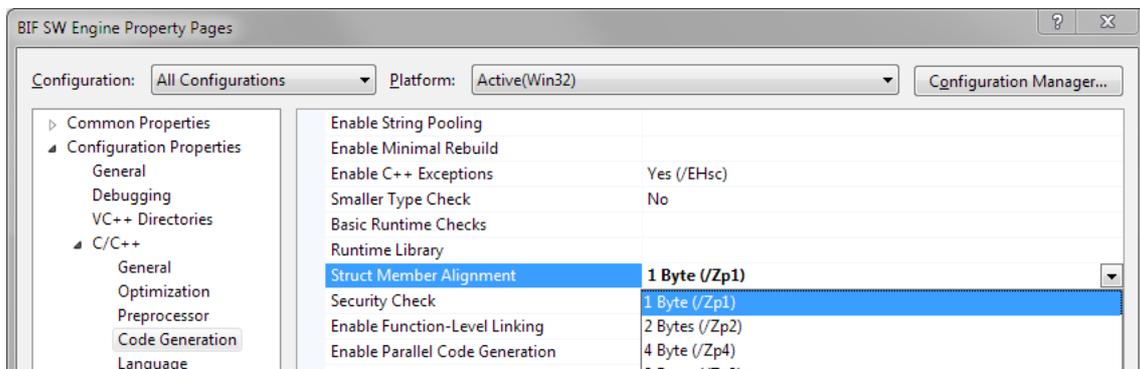


FIGURE 68. Setting the Struct Member Alignment

Because the Wrapper for BIF SW Engine is quite small, the precompiled header feature from Visual Studio is not used (12). Precompiled headers can be disabled from project properties like shown in figure 69.

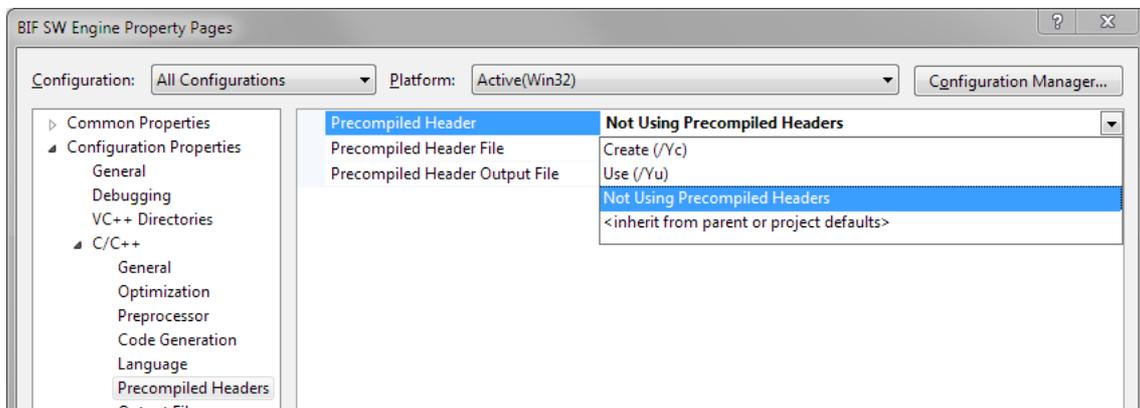


FIGURE 69. Disabling precompiled headers

For some reason the labviewv.lib throws a safe exception handle check error when compiling. According to MSDN (Microsoft Developer Network), the most common reason for this to happen is because the module in question does not support safe exception handles (7). Because of this, the check for safe exception handles is disabled in order to compile (see figure 70).

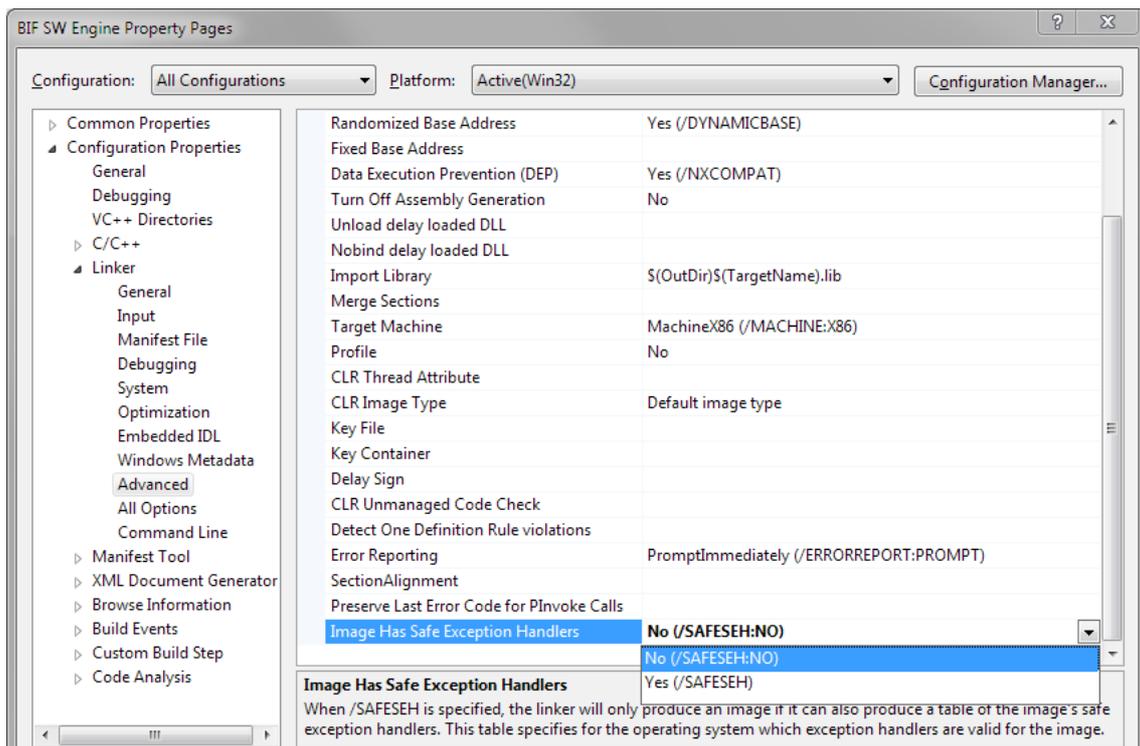


FIGURE 70. Disabling the Safe Exception Handles check

All the required configurations are done now and the project properties window can be closed with the OK button. Before compiling, the active build configuration needs to be changed from debug to release. The build configuration can be changed from the top center of the main Visual Studio window.

The Wrapper for BIF SW Engine can be now compiled from the build menu or by pressing the F7 key. The compiled DLL file should appear in the Release folder under the project folder. This DLL can then be placed to an appropriate location where every computer that uses it can reach it.