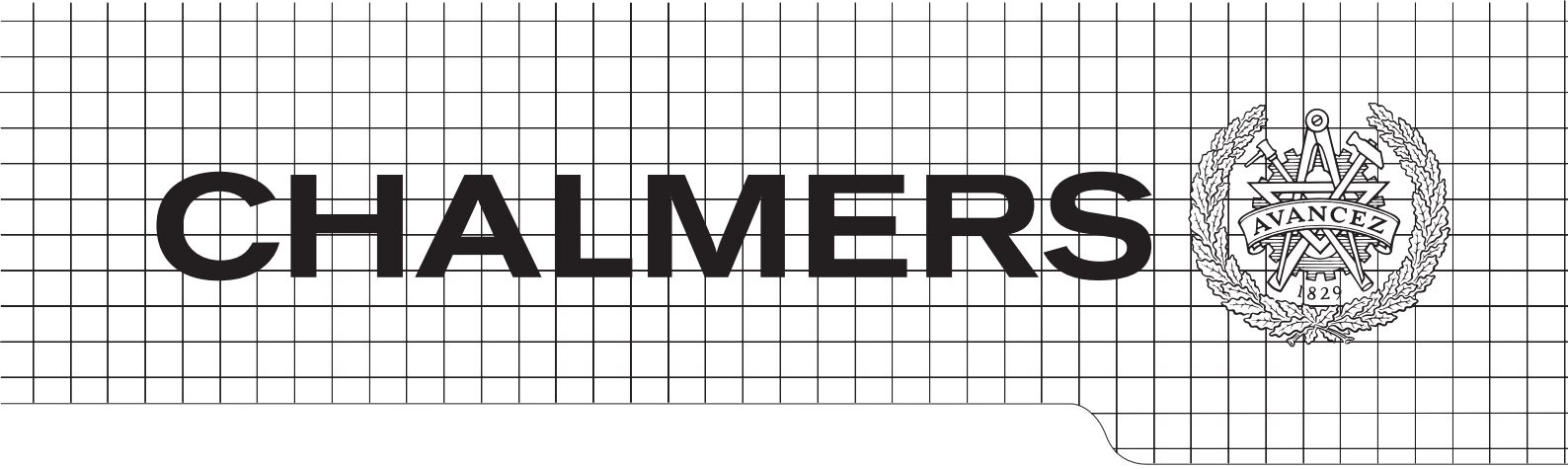


DC/DC PMBus Compliance Tester

Master Thesis in Computer Science & Engineering

NICKLAS PERSSON

Department of Microtechnology and Nanoscience
Chalmers University of Technology
Gothenburg, Sweden 2012

Abstract

Digital power management involves the ability to manage and configure a DC/DC converter with commands through a communication system. The Power Management Bus (PMBus) is an open standard protocol that can be implemented in a power management system for controlling the power converters on a printed circuit board. The advantage of this standard is that the system designer can easily transmit commands to any PMBus device without having to consider which device is used. Another advantage is that power converters can be replaced by other converters as long as they are compliant with the PMBus standard. Also, an previously used PMBus implementation can be reused and another PMBus device can be substituted as long as the standard is followed. It is therefore important that the chosen PMBus device is compliant with the standard and will behave as commanded. A PMBus device should be tested early in a product design phase for PMBus compliancy otherwise problems may be introduced late in the design phase and that can be costly. Therefore, a PMbus compliance tester would be beneficial. However, such test system does not exist today.

This report presents a PMBus compliancy tester, i.e. an automated test system that checks and verifies if a PMBus device is compliant with the standard. The system is modifiable and tests are modified by the use of script files. For this purpose an new script language has been defined, offering the test engineer the possibility to adjust each script file independently. The script file defines the commands and the parameters to be tested. When the script file is executed by the test system a log file with all the desired test results is produced. This flexibility makes the test system developed in this project superior to any existing automated test system. The tester is implemented as a LabView program featuring a graphical user interface. Verification measurements are performed via a GPIB interface that controls an oscilloscope, a power supply and an electronic load. Each script command is implemented as a separate sub-program and a main program calls the sub-program when the corresponding script command is identified by a parser. This approach makes it possible to substitute an instrument without having to redo more than one sub-program. Therefore the modularity of this tester is believed to be an benefit in a laboratory environment where instruments are frequently replaced and occupied.

Acknowledgements

First of all, the author would like to thank Magnus Johansson and Peter Hohlfält for giving the opportunity to do the thesis work at Ericsson Power Solutions.

Furthermore a special thanks to Henrik Isaksson and Björn Olsson for assistance during the thesis work. Henrik was the thesis supervisor and assisted with the planning of the work and with design decisions. Björn assisted with technical details about the PMBus specification and with programming guidance.

Finally, I want to thank the thesis examiner Kjell Jepsson for report writing feedback and structure guidance.

Nicklas Persson, Gothenburg 2012

Contents

1 Introduction **8**
- 1.1 Background . . . 8
- 1.2 Purpose . . . 9
- 1.3 Objectives . . . 9
- 1.4 Constraints . . . 10
- 1.5 Related work . . . 10

2 Power Management Bus **13**
- 2.1 Introduction to the Power Management Bus . . . 13
- 2.2 The underlying bus protocol SMBus . . . 14
- 2.3 Commanding PMBus devices . . . 15
- 2.4 PMBus operating memory and startup routine . . . 17
- 2.5 Turning a PMBus device on and off . . . 19
- 2.6 Configuring the output voltage . . . 19
- 2.7 PMBus fault management . . . 20

3 System approach **22**
- 3.1 Implementing the PMBus . . . 22
- 3.2 Transmitting PMBus commands . . . 23
- 3.3 Test instruments . . . 23
- 3.4 Controlling the PMBus and test instruments . . . 24

4 Test system **26**
- 4.1 Main program and graphical user interface . . . 26
- 4.2 Script language . . . 29
- 4.3 Transmitting PMBus commands . . . 29
- 4.4 Instruments and GPIB . . . 31
  - 4.4.1 LabView and electronic load GPIB-interface . . . 33
  - 4.4.2 LabView and oscilloscope GPIB-interface . . . 35
  - 4.4.3 LabView and power supply GPIB-interface . . . 38
- 4.5 System summary . . . 38

5 Verification test programs **40**
- 5.1 Verifying the output DC level . . . 40
- 5.2 Verifying rise and fall times . . . 42
- 5.3 Verifying ON and OFF delays . . . 43
- 5.4 Verifying transition rates . . . 45
- 5.5 Verifying outgoing currents . . . 46
- 5.6 Verifying status registers . . . 47
- 5.7 Summary of compliance test programs . . . 49

6 Results **50**

7 Discussion **51**

8 Conclusion **52**

1 Introduction

Ericsson has a vision to be the prime driver in an all-communicating world. They believe that better connectivity improves quality of life and has become basic infrastructure for economic growth and prosperity [1]. They develop power circuit boards that deliver power to radio base station systems. Such power boards should be low powered and environmental sustainable. Digital power converters can adjust the board power consumption to use a power saving scheme. This kind of power saving scheme is important when designing power efficient electronic products that is environmentally sustainable. Studies have shown that the environmental impact of the manufacture phase is less than the actual energy consumption throughout the whole product's life-cycle [2] [3]. To be as competitive as possible the industry needs to adopt to intelligent power management systems since large energy consumers are increasingly demanding higher value for their energy investment. Energy saving is also without doubt the most effective strategy for environmental protection [4].

The Power Management bus (PMBus) can be used to control and monitor power converters. PMBus is a power management bus standard for digital power converters that includes a set of commands for control and monitor all power converts connected to the bus. For example, the rise and fall time for a changing output voltage level can be configured. A system controller is bus master and can be reused to other power converters if they are compliant with the PMBus standard.

This report describes the implementation of a compliance tester that can be used to verify that a given power converter follows the PMBus standard. In this chapter the background and purpose to the problem are given.

1.1 Background

Usually a circuit board contains a number of integrated circuit chips, each chip needing its own voltage level for power up. The number of power converters needed on the board is then the same as the number of voltage levels. The power distribution can therefore be complex and expensive as the number of voltage levels on a board increase. One solution would be to place multiple power devices on the board, which distribute different power levels and communicating with a centralized system controller. Some kind of interaction is needed to configure, control and monitor the power devices. The power devices are DC/DC converters and the system controller can be a microcontroller, a dedicated chip, a laptop etc [5].

This is known as digital power management and was first introduced into telephone central office power systems in the 1980's [6]. The advantage with digital power management is that changing the output voltage from a power converter is done by sending a digital command message to the device, compared to the analog system where one has to actually pick up a soldering iron and manually add resistors.

In 2004, semiconductor and power supply companies grouped together to develop an open standard called PMBus for communicating with DC/DC converters [7]. This has become a standard bus in industry for configuration of power converters. By implementing the PMBus for a system, one can minimize the component count and power consumption. The ability to

adjust the power during normal operation makes it easy to implement power saving schemes [1].

When designing with power devices, the PMBus makes it possible to implement the power management once and reuse for next design with another type of power device following the same standard. Also, the designer does not need to know how the power converter exactly works but can be sure that it operates as the PMBus commands control it.

However, if a power converter is expected to follow the PMBus standard but is not fully compliant, problems may arise late in the design phase and delay the product time to market. A power converter that is supposed to be used in a system needs to be fully tested and configured before it is integrated with the system. A test suit where a PMBus converter's functionality and PMBus compliance could be easily verified would be beneficial.

1.2 Purpose

In a power board design project it is important to be able to trust the power converter manufacturers that the chosen power devices truly comply with the PMBus standard. Detecting that a power device is not PMBus compliant late in the design phase can cause high cost and significant delays. To avoid this, a PMBus compliance tester would be beneficial to actually verify the PMBus converters early in the design phase.

1.3 Objectives

The main objective is to implement a PMBus compliance tester that verifies if a PMBus converter complies with the standard. It should be possible to attack the converter with PMBus commands and observe the expected behavior of that converter. A graphical user interface (GUI) will provide the test result, which should be easy accessed and should clearly state which commands that failed and which passed the test. This tester should be possible to be used early in the design phase to verify that the converter will behave and function as expected. This main objective is divided to sub-objectives as described in the following list.

1. **Understand the PMBus standard.**
   It is necessary to first get a basic understanding of PMBus. A brief description of all the important aspects of the PMBus should be documented and reported in this report.
2. **Select system components.**
   An analysis of what kind of system component is needed is supposed to be done. How the system is going to be implemented should be known when this sub-objective is reached. The problem on how to make the tester modifiable and automated should be answered.
3. **Interface to the test instruments.**
   An electronic load, an oscilloscope and a power supply is needed. Since the tester is going to be automated, these instruments needs to be remotely configured. A communication system needs to be implemented so the tester can communicate with the instruments.

4. **PMBus configuration tool.**
   The tester should be able to configure the test PMBus converter by transmitting PMBus commands to it. Therefore, each command should be implemented independently so that the each command can be issued. After this sub-objective the test converter should be fully configurable.
5. **Develop a graphical user interface.**
   A graphical user interface (GUI) should be implemented. The test engineer should be able to control and observe the execution of each test using this GUI.
6. **Verification programs.**
   To actually verify a functionality of the PMBus converter, some test programs are needed. These programs should be used to verify the converter's properties and presents the result to the GUI. For example, after a PMBus converter has been configured to output 5 volts a verification program should confirm that the newly configured output is 5 volts.
7. **Provide a test program.**
   The tester is going to be fully modifiable and each test case is built for independently test converters. The user could modify the tester so that it matches his needs. However, in this report a fully function test program that verifies a randomly chosen PMBus converter should be provided. The user can then use the provided function or modify it if needed.

1.4 Constraints

The tester is constricted to only write test programs for a subset of all PMBus commands. The subset is chosen based on what is believed to be the absolute minimum subset needed for Ericsson's power boards. The tester will only test the functionality of a PMBus converter based on these commands.

The converter is first configured and then the new configured functionality is verified. There is therefore only a command-level tester. The tester can therefore not test bit-level bus transmission or bus timing issues.

1.5 Related work

In this section related work and papers are presented. PMBus verification work has not been found since this standard is relatively new and not widely used yet. However, other bus standards have some related work that has been interesting and inspiring for preparing the implementation of the PMBus compliance tester.

For example, Baccolini and Offeli [8] have done a program to generate bus-test programs for microprocessor systems. The illustrated program utilizes criteria to find sequences of bus instructions which are able to detect a bus fault. A user interface is made so that the user is able to choose a set of instructions to be tested.

The CAN-bus is widely used in the automobile industry and previous work could be used as inspiration. Chengqun [9] has done a CAN-bus test based on the Volkswagen PASSAT B5. The test bench setup is similar to the one used in this report, where oscilloscope, multimeter, PC and a CAN-bus interface card platform is used and the test software is written in C++. However, the GUI only lets the user test one device operation at a time and needs to be configured for each test.

A LIN-bus has also been tested [10] in a similar approach as in this report, by offering the user to insert data on the bus for testing and debugging purposes using microcontroller and written software in C code. This was the inspiration that lead to the PMBus tester to be modifiable. Another research describes a similar test suit using an FPGA instead [11].

It is also possible to use formal methods for compliance verification. Nguyen et al. [12] formally specified on-chip bus protocols and protocol compliance of communication blocks in System-on-chip designs. A type of recorder is implemented to remember previous bus states is designed independent and can be re-used on any design implementing the same protocol. A formal verification of bus-bridges has also been done [13]. Bus-bridges are used to connect different types of protocols.
A formal method has also been used to verify compliancy of the PCI-bus [14]. One major advantage with formal verification is that the test can be easily re-configured with a textual configuration file for the bus specification. This has been the inspiration for using script files to make the PMBus tester modifiable. Formal methods are not considered in this report.

Cram [15] has a similar approach to the test suit as in this report when testing GPIB compliance. A test script is written in C that generates test sequences and generates a pass or fail indication for every test sequence tested and a log file to interpret the result. The test suit is also independent of the product and strictly black box. A similar log file generation feature is used for the PMBus tester.

Instead of having a test suit, a real-time on board bus testing approach has been tested. Floyd and Perry [16] shows that it is possible to have a test logic device that generates meaningful test results for a bus added to an ASIC itself. This could be interesting for future work of this report.

Raju et al. [17] propose an automated testing tool for testing satellite communication systems with instrumentation and LabView algorithms. The method they used was to categorize all tests to sub-tests. For each one of the tests a separate LabView program was coded accordingly. The tests include measurement of currents, internal voltage, temperature measurement etc. The instruments are controlled with NI PXI modules. This tester setup is very closely related to the tester in this report.
Zhao [18] has implemented an even more related tester, since a DC/DC converter is tested using an automated measurement tool in LabView. The converter performance is recorded and analyzed with experiment data. The voltage regulation and converter efficiency is tested and the result is provided as an excel file. An electronic load, power meter and a power supply is controlled through a GPIB interface. The GUI used in this LabView program has many advanced features and includes waveforms of the converter output.

Neskovic and et al. [19] has also implemented an automated testing tool using LabView. They test multiplexer devices with PXI instruments connected through a GPIB interface. The GUI they implemented makes it possible to adjust test parameters from a list of parameters.

However, the user is then strictly limited by the set of parameters provided by the GUI. The tester presented in this report enables the user to configure any test parameter. The test results are stored in a MS Word document with information about which test was performed and at what time and date of the test execution. The time and date information is also used in the log files for the tester presented in this report. However, a MS Word document is considered as a restricted type of document since the user has to have MS Word installed. A regular text file is easier to move from computer to computer and can be opened with any type of document software.
Neskovic and et al. [19], Raju et al. [17] and Turley and Wright [20] expresses the great advantages with LabView which has been used as an argument for using LabView for the tester presented in this report.

2 Power Management Bus

In this section the Power Management Bus (PMBus) is introduced. First, a brief background and purpose is given in section 2.1 before moving on to the more technical details of the bus protocol. After the introduction the SMBus is described in section 2.2. This is needed since PMBus lies on top of this protocol. In section 2.3 the PMBus command language is introduced and the most usual commands are presented. In section 2.4 it is described how PMBus devices operates on configured parameters that can be saved on the operating memory and later re-loaded for the next time the device is powered on. The following section describes how to turn-on a device and how to configure the output voltage. Finally, the fault management that can be used with advantages for a fault tolerant system is presented in section 2.7. Notice that these sections are just introducing the PMBus. For the full details the PMBus specification [21] and [22] can be studied.

2.1 Introduction to the Power Management Bus

Power management is about configuration, controlling, fault management and monitoring of one or several power devices. These functions can be a combination of analog and digital. Digital power management implies that all functions are implemented digital via a data communication bus. It offers several benefits. For example, temperature, voltage and output current can be monitored to regulate cooling fans and power saving schemes. Digital power management has a structure with several power devices communicating with a control system. The power devices are DC/DC converters and the control system can for example be a PC or microcontroller. The power devices are called bus slaves and the control system is called bus master.

As the number of voltage levels on a circuit board increases the power control becomes complex. Voltage sequencing, ramp times and delays need to be controlled for both normal start-up and shutdown operation as well for some fault handling. This is straightforward with digital power management.

The communication bus between the converters and the control system can be implemented with the Power Management Bus (PMBus) protocol. It is owned by the System Management Interface Forum and the membership is open to all interested parties and the PMBus specification is freely distributed [5].

The protocol is based on the System Management Bus (SMBus), which is based on the widely used Inter-IC Bus(I2C Bus), but is supposed to give more functionality for power control. This is why the PMBus is a great choice for the bus in a power management system. By implementing the PMBus one can minimize the component count and makes it easy to implement a scheme that delivers system-level power savings [23].

In addition to the I2C bus, PMBus protocol also implements the optional alert response signal that could be used in presence of a fault. The faulty device then pull the alert signal low at any time to notify the control system that it needs to communicate. The control system then read the faulty device's status register. Also, an additional control line can be used to turn a slave device on or off. The PMBus provides 255 commands that the control system can instruct the

slave devices. Another featured is the Packet Error Checking (PEC) that increases reliability and robustness by using an algorithm to validate the integrity of a transaction [24].

Further technical and specification related details are given in the official specification, [21] and [22].

2.2 The underlying bus protocol SMBus

Before one can understand the PMBus technical details it is necessary to get the basics of the underlying layers. PMBus is based on the SMBus, which is derived from I2C. In this section, a short introduction to the PMBus basics is given. For further technical details the reader is adviced to the specification [21][22].

SMBus has two required signal wires that has derived from the I2C bus. The first signal, $SMBCLK$, is the clock generated by the master device at 10 kHz to 100 kHz. The second signal is the $SMBDAT$ which is the data communication link between the master and the slaves.
In addition to the I2C bus, there is also an optional signal wire. The $\overline{SMBALERT}$ interrupt signal is optional and used by the slaves to alert the master device that a fault has occured.

The master controls the clocking by drive the $SMBCLK$ signal. During transactions the $SMBDAT$ signal may change when the clock is low and needs to be stable when the clock is high.

Typically, the bus master initiates a transfer to a slave when it puts a start condition on the bus to inform the slave devices that a transaction is about to start. The start condition is made when the $SMBDAT$ signal go from high to low while the $SMBCLK$ is high. The bus is busy until the transaction is finished. In a transaction, the bytes are transmitted starting with the lowest ordered byte first and ending with the highest ordered byte. For example the value 0xABCD would be transferred as two bytes with the first byte 0xCD followed by 0xAB. Each byte is transferred with the most significant bit first.

After the start condition the master puts a seven bit unique address to the receiving slave on the bus. Each device has a seven bit address. It is very important that all devices have unique addresses to avoid conflicts. The address is hardware wired to the device by the system designer and not configurable through the bus. The eight bit is used as a $READ/\overline{WRITE}$ bit that notifies the receiving device that the master is supposed to write to or read from the device.
All the slaves on the bus will compare the address with its own and the matching slave will acknowledge the address with an ACK condition on the bus and continue to listen on the bus. The master detects the ACK and continues with the byte transaction by toggling the clock. When the receiving device has received eight bits, it is supposed to pull the $SMBDAT$ line low to ACK or pull it high to not acknowledge (NACK) the byte on the ninth transaction bit.

When the transaction is finished the master puts a stop condition on the bus. If the master device notifies an NACK it ends the transaction and initiates the error response that is implemented by the system programmer. The stop condition is always used when the transaction is finished. The master puts a stop condition by pull the $SMBDATA$ signal from low to high

while the $SMBCLK$ is high. In figure 1 an illustration of how the SMBus protocol issues a byte transfer is given.

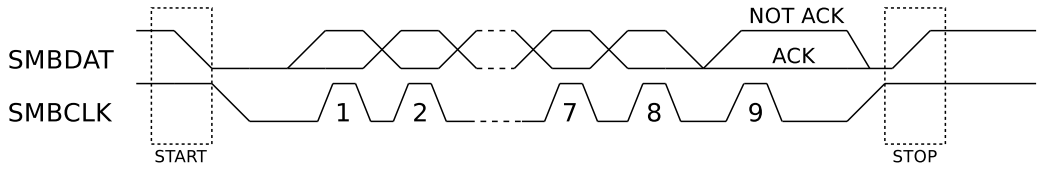


Figure 1: *An illustration of how a start condition is generated on the SMBus. Then an byte is transferred and the receiver either ACK or Not ACK the byte. The data transmission is finnished with the stop condition.*

One feature of the SMBus is that it allows clock stretching. This means that if a device needs more time it can stop the transaction temporary. For example the device may need some time to process the received data or need to wait for some slow memory when asked for data. So, the device hold the clock line low. To avoid problems with clock stretching the master device needs to be fully compliant with the SMBus specification and it is up to the system engineer to assure that any microcontroller used as bus master is SMBus compliant [25].

2.3 Commanding PMBus devices

In this section we extend the basics given in the previous section about the SMBus and introduce the additional features that makes the PMBus. A basic introduction how to issue PMBus commands is given. Commands that uses data values as parameters need some kind of technique to interpret the data. This is where the two data formats is necessary. The data format LINEAR and DIRECT is introduced in this section. The PMBus is only briefly introduced and the reader is advised to read the full PMBus specification ([21] and [22]) for further details. As mentioned earlier, the PMBus is based on the SMBus with some extensions. The technique behind transaction of bytes is still the same as in SMBus.

The extensions that makes the PMBus is the defined command language used to control, status monitoring, fault management and information storage on power devices. The PMBus uses the SMBus signals $SMBDAT$, $SMBCLK$ and $\overline{SMBALERT}$. Also, the PMBus has a fourth signal named $CONTROL$ that is hardwired and used to turn a device on or off. Figure 2 illustrates how the PMBus signals are used in a typical power management system.

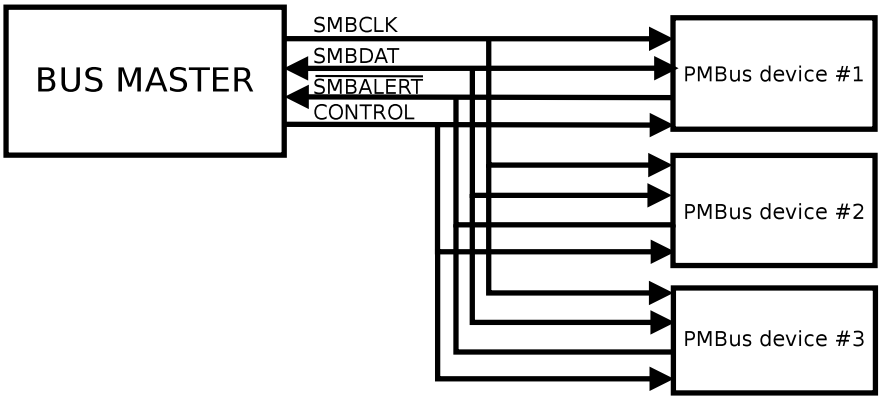


Figure 2: *An example of a PMBus system with the signal lines wired. Three devices act as slaves and another device is the bus master.*

During transactions the master issues a start condition to notify the slaves that a transaction is about to start. Then the master transmits the same seven bit unique address technique as in SMBus. However, in PMBus there is also the possibility to broadcast to all bus slaves by addressing the address 0x00. After the address the $READ/\overline{WRITE}$ bit is included. After the address has been acknowledged the master send the PMBus command. Each PMBus command is a hexadecimal byte code which gives 256 possible combinations of commands. Each command has therefore a number and a name. The number is used for the PMBus devices to recongnize the command and the name is used for humans to recognize the command. In this report each command is represented by its name written with capital letters and the hexadecimal code is written as index to the name (example $VOUT\_COMMAND_{0x21}$). Listing of PMBus commands and their command codes are listed in table 29 in the PMBus specification [22]. A PMBus slave receives a command by the master and start executes the command as soon as the stop condition is issued by the master. The slave device does not wait for an ”execute” command. The PMBus command transaction is illustrated in figure 3.

Figure 3: *Illustratin of how data bytes are transmitted on the PMBus. First a stop condition (S) is generated by the master that then puts the receiving slave's address on the bus with an read/write bit (W). The receving slave acknowledge (A) the address. The master puts one or several data bytes on the bus where each byte is acknowledged by the receiving slave. When the transaction is finnished the master puts the stop condition (P) on the bus.*

PMBus devices usually use two types of data formats when receiving and reporting data values, LINEAR and DIRECT. The LINEAR data format is a two byte value typically used for commanding and reporting physical units such as voltage, current, temperatures, time and energy. The data format can be thought of as a floating point number. Figure 4 shows how the two data bytes are structured. The mantissa, Y, is an eleven bit two's complement binary integer. The exponent, N, is a five bit two's complement binary integer. The complete data value is given as $X = Y \cdot 2^N$ [26].

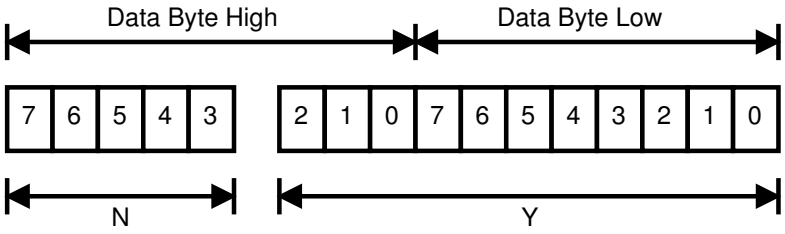


Figure 4: *Two data bytes are used for the LINEAR data format. Y is the mantissa and N is the exponent.*

The DIRECT data format may be used for any command that sends or reads a parametric value. The DIRECT data format eliminates the need for the PMBus device to process a floating point number. DIRECT data format uses equations to encode and decode data values. To encode data equation (1) is used.

$$Y = (mX + b) \cdot 10^{R} \tag{1}$$

Where X is the value to be encoded, m is the 16 bit two's complement byte slope coefficient, b the 16 bit two's complement offset coefficient, R is the 8 bit two's complement scaling coefficient and Y is the 16 bit two's complement value transmitted over the bus. To decode data equation (2) is used.

$$X = \frac{1}{m} \cdot (10^{-R} \cdot Y - b) \tag{2}$$

The coefficients are device independent and decided by the manufacturer. The coefficients are available by issuing a $COEFFICIENTS_{0x30}$ command to the given PMBus device which will respond with the correct coefficient values.

In this section we have discussed the general basics about the PMBus and how to use the commands. How the two data formats are used was also introduced. This basic introduction is supposed to give the reader enough understanding for this report.

2.4 PMBus operating memory and startup routine

A PMBus device operates on configurable parameters such as output voltage, fall time, rise time, transition rate, rise delay, fall delay and etc. These values are stored in a volatile memory. This memory is called the *Operating* Memory. From the beginning, when the device is powered on, this memory is empty and needs to load parameters before the device can start operate normally. The routine to get the device up and running with an Operating Memory loaded with parameters is described in this section.

First, the PMBus device checks if there is any hard coded parameters.

Second, the device checks if there is any pin programmed parameters. If any of the pin parameters are the same as the earlier hard coded parameters, the pin programmed values

overwrites the previous loaded hard coded value. This is a general rule, when parameters are loaded they will overwrite any previous values.

Third, the device loads parameters from a non-volatile memory called the *Default Store* memory. Here, the device manufacturer can decide upon how the device should operate when it is in its default mode. It is up to the manufacturer if it should be possible or not to load values to the Default Store memory.

Fourth, the device loads any parameter from another non-volatile memory called the *User Store* memory. The user can load and restore parameters to this memory anytime. By storing a copy of the operating memory in this memory the device will resume operation with the last working set stored by the user when the device is turned on.

Finally, the startup routine is finished and the device can start to accept PMBus commands on the bus. Therefore, any PMBus command can change the operating memory and previous parameters will be overwritten, even the hard coded parameters. In figure 5 the startup routine is illustrated.

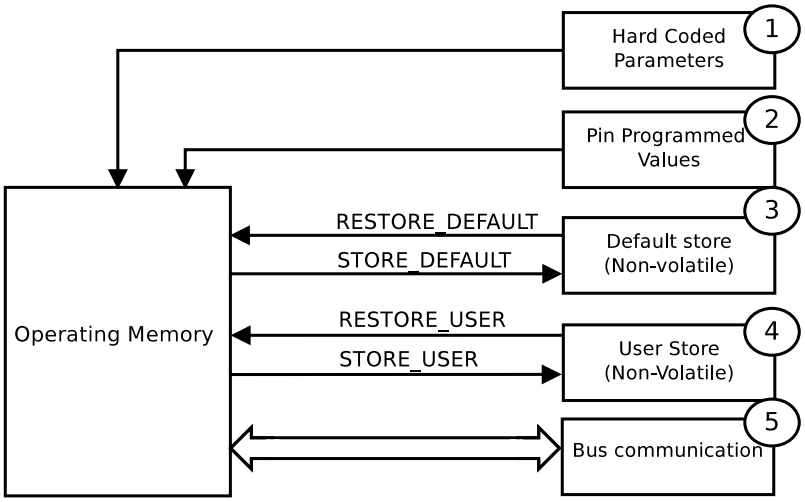


Figure 5: *An illustration of how a PMBus device's memory is restored during a startup.*

To store the volatile operating memory to any of the two non-volatile memories there is four commands. To copy all of the operating memory to the User Store memory the $STORE\_USER\_ALL_{0x15}$ command is used and $STORE\_DEFAULT\_ALL_{0x11}$ to store the Operating Memory to the Default Store memory. There is also the possibility to just store one parameter with $STORE\_USER\_CODE_{0x17}$ and $STORE\_DEFAULT\_CODE_{0x13}$. This could be used to modify an already working copy.

To load any of the memories to the Operating Memory the commands $RESTORE\_USER\_ALL_{0x16}$ and $RESTORE\_DEFAULT\_ALL_{0x12}$ are used. And if the user wants to just load one parameter from any memory the $RESTORE\_USER\_CODE_{0x18}$ and $RESTORE\_DEFAULT\_CODE_{0x14}$ commands is used.

2.5 Turning a PMBus device on and off

A PMBus device can be configured to behave differently when powered on or off. The device can use sequencing which means it can output increasing or decreasing voltage when turned on or off. The sequencing behavior is defined by the $ON\_OFF\_CONFIG_{0x02}$ and the $OPERATION_{0x01}$ command actually turns the device on or off. The device can be configured to be turned off immediately or by using a programmed turn-off delay and turn off fall time. The command to configure the fall time and the turn-off delay is $TOFF\_FALL_{0x65}$ and $TOFF\_DELAY_{0x64}$. The same principles are true for turning the device on. The turn-on delay is configured with the TON_DELAY_{0x60} command and the rise time is configured with the TON_RISE_{0x61} command. The turn-on and turn-off with sequencing and delays are illustrated in figure 6.

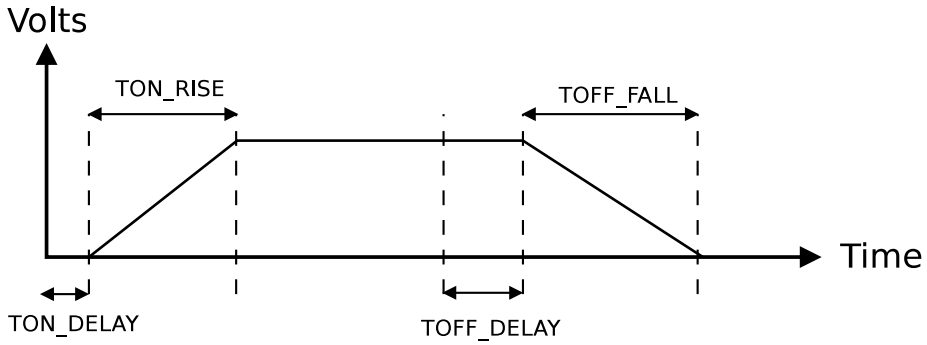


Figure 6: *A PMBus device's output. When the device is powered on or powered off a delay could be introduced. How fast a device will turn on or off depends on the rise and fall delay.*

2.6 Configuring the output voltage

In this section a conceptual description how to control and monitor the output voltage of a PMBus device is given. The actual implementation is up to the device manufacturer and is not described in this report.

First the data format (LINEAR or DIRECT) that is used when defining the output voltage level is set by the $VOUT\_MODE_{0x20}$ command. Then the actual voltage level is set with the value defined by the $VOUT\_COMMAND_{0x21}$ command. To set the output voltage level there is several commands involved. Figure 7 shows how all of these commands are used. The process starts by loading one of the parameters that can be controlled with one of the commands: $VOUT\_COMMAND_{0x21}$, $VOUT\_MARGIN\_HIGH_{0x25}$ or $VOUT\_MARGIN\_LOW_{0x26}$. These values are passed to a multiplexer and one of these parameters is selected by the $OPERATION_{0x01}$ command.

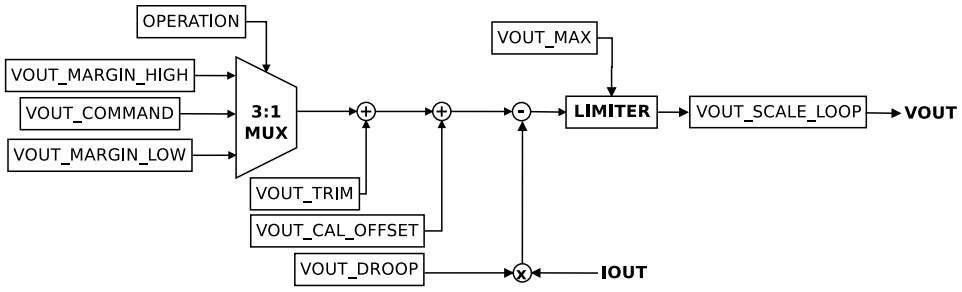


Figure 7: *Illustration of how a PMBus device's voltage level is configured.*

In the next step an offset value is added to the output voltage. This offset value can be used by the user to trim the output voltage after the PMBus device has been assembled into a system. The command to change the offset value is the $VOUT\_TRIM_{0x22}$ command. A calibration offset value is added to the output voltage which is typically used by the PMBus device manufacturer to adjust the output voltage in their factory. The command to change this second offset value is the $VOUT\_CAL\_OFFSET_{0x23}$ command.

Next, if one choose to have an output voltage droop characteristic it is applied. To configure the voltage droop coefficients the $VOUT\_DROOP_{0x28}$ command is used. The coefficients are always greater or equal to zero and is multiplied with the output current. The result is subtracted from the voltage command. With this characteristic the output voltage decreases with increasing output current and increasing with decreasing output current.

It is possible to set a maximum allowed voltage value by the command $VOUT\_MAX_{0x24}$. The calculated voltage is compared to this maximum limit and if the voltage is below this value everything is okay. However, if the value is above the limit, the output voltage will be the maximum allowed and fault has occured. The PMBus fault management is described in section 2.7.

The last step is to multiply with a scaling factor that is configured with the $VOUT\_SCALE\_LOOP_{0x29}$ command. Even though several commands can be used to set the voltage level it is usually just enought to use the $VOUT\_COMMAND_{0x21}$ when everything else is configured and calibrated.

2.7 PMBus fault management

The PMBus protocol specifies a comprehensive set of actions that can be performed during faults. A typical fault can for example be if the host sends an unsupported command or an invalid data. Faults can also be output related such as under voltage and over temperature. The fault management is a huge advantage for the PMBus. For a power conversion device each physcial property can be configured with limits for generating warnings and limits for generating faults.

Each physical property such as temperature and output voltage can be read by a read command. Also, every device stores status registers where the user can retrieve information about the current state of the device. If a warning or fault has been introduced it is representing

in one of the status registers. All status information is stored in a two-byte word register where the most critical faults are indicated by the lower ordered byte. The command $STATUS\_WORD_{0x79}$ retrieves the two-byte word register and $STATUS\_BYTE_{0x78}$ retrieves the lower order byte. When a fault is generated the corresponding bit in a status register is set and can be read by the system controller. There exists other registers and a PMBus device can be compliant with up to eight different status registers.

As mention in the beginning of this section it is possible to manage the warning and fault limits. A warning condition notifies that there is something wrong but it is possible to continue. A fault is more critical and a device needs to take immediate actions to prevent the risk of damaging the load or device. The action to perform during fault is configurable.

A fault can be detected in several ways. One approach is for the host to continue poll the PMBus device for the status registers. The other approach is to let the device notify the host by the $\overline{SMBALERT}$ interrupt signal or take over as bus master and send a notice to the host that a fault has occurred. How to configure the response to a fault and general fault management is described in more detail in the PMBus specification [21]. Example of typical fault responses are that the device performs a number of retries or that the device turns off the output.

3 System approach

A PMBus tester is supposed to be implemented. It will be a system built from several system components. The PMBus tester is going to transmit and receive data through a PMBus and therefore require a component that implements the bus communication. Also, to actually perform some analysis a number of instruments are required to verify the behavior of a test device. In figure 8 an overview of the system components is illustrated.

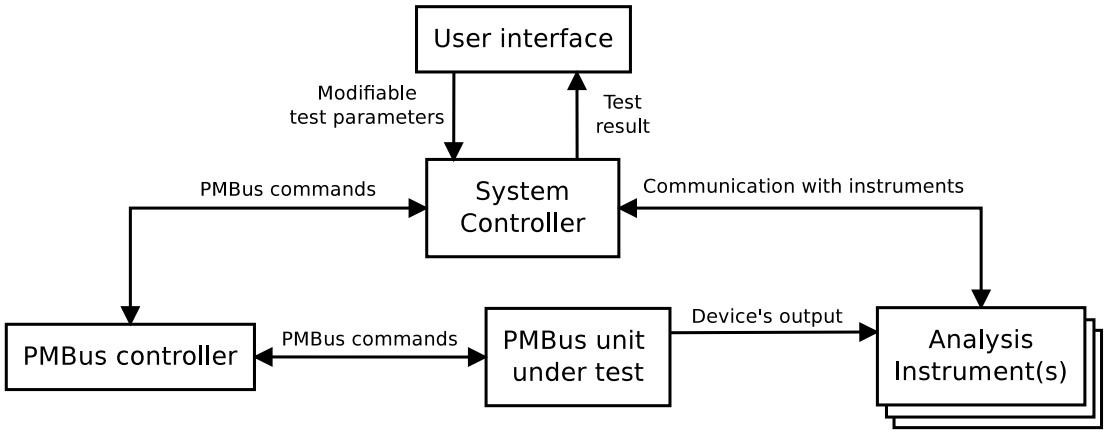


Figure 8: *A block diagram of the needed components for the tester.*

In this section the system components are selected based on some reasoning about their advantages and disadvantages from several alternatives. A brief introduction to the meaning of a system component is given, some proposed solutions are also presented and then an explanation of why the solution was chosen is given.

3.1 Implementing the PMBus

The actual PMBus can be implemented in several approaches. The most straightforward approach is to build it by hand with some cables and directly connect to the PMBus device that is going to be tested. All test instruments are also connected directly to the device. The implemented bus needs to be fully tested and verified to make sure that a problem that does occur is not because of a failure in the bus. To switch PMBus device, one need to redo the whole procedure and re-connect all cables.

The other approach is to use a factory implemented development board that implements the bus and has routed outlets for PMBus devices. This kind of boards makes it easy to switch a device and also to connect test instruments to the output of the device. One board that has been available for this report is the 3E Evaluation Board from Ericsson. This board is not widespread so it is could be hard to get hands on. However, it is available for this thesis and is therefore used.

3.2 Transmitting PMBus commands

For testing a DC/DC converter's compliancy with PMBus it is required to be able to transmit data over the bus to the converter. Two approaches to send data over a PMBus are described here.

One approach is to use a microcontroller. Then, the PMBus protocol is implemented by hand as a software program downloaded to the microcontroller. The PMBus data signals can then be completely under control and bus timing issues can be tested. One can simulate how to transmit corrupted bits since one has total control over what is transmitted on the bus. A microcontroller is relatively cheap and easy programmed.
However, this solution is time consuming and can be a project by itself. Also, when the test program is included it can be hard to verify if there is a compliance problem with the test device or the actual software implemented PMBus is imperfect.
The microcontroller also needs some communication with the other system components, so based on what the other components are this bus communication can be complex.

The second approach is to use a factory complete solution. In this case it could be a dongle that connects a PC with the PMBus through USB. Compared to the microcontroller solution, this is easier and does not take any considerable amount of time since it is already a complete product. However, with a dongle one has no control of the bus lines and it is not possible to transmit any corrupted data since the dongle always transmit PMBus data messages correctly.

Two kinds of dongles are considered in this report. There is the 1613A [27] and the USB Interface Adapter from Texas Instruments. These two dongles are almost the same in terms of features and design. The dongles implements a PMBus communication, uses a USB interface to a PC and includes evaluation software. The evaluation software makes it easy to transmit a single data to the bus. However, in this report there should be an automatic test that transmits several data without any interaction. To the USB Interface Adapter from Texas Instruments one can download a DLL-file called "USB Adapter Driver.dll" that includes two functions for receiving and transmitting data over the PMBus. A DLL-file can be used outside of the evaluation software and makes it possible for writing software that automatically uses the provided functions without any interactions. Therefore, the USB Interface Adapter is used in this report for transmitting and receiving data over a PMBus.

3.3 Test instruments

The test environment requires several lab instruments. Each PMBus device that should be tested needs to be powered by a power supply. A standard power supply that statically outputs the same voltage all the time is just not good enough. The power supply is required to be adjustable and remotely reconfigurable. This is because there is one test case where the input power is changed and the behavior of the device is monitored. This test case is going to remotely adjust the power supply.
Also, different devices could be powered differently, for example the BMR453 [28] has 36-75V and the BMR450 [29] has 4.5-14V as input voltage.
One power supply that fits these requirements is the Xantrex XKW80-13 [30] which is remotely configured with GPIB bus and operates up from 0 to 80 volts. This power supply is used in

this report.

The device's output needs to be monitored and analyzed to perform verification on the device's output characteristics. A voltmeter could be used to monitor the DC level. However, with a voltmeter is not possible to acquire the volts remotely. Another requirement is that the instrument that monitor the output should be able to look at timing characteristics such as transition rates, rise and fall time. This is another reason for why a voltmeter is not enough. Instead, a typical digital oscilloscope with up to four channels and some feature with remote access will meet the requirements. In this report, the Wavepro 960 [31] from Lecroy is used. With this oscilloscope one can use a GPIB bus for remote access. Since this is the same technique as the power supply in this report they could operate on the same GPIB bus. Therefore, one can reduce implementation time by also use GPIB for the oscilloscope. This oscilloscope is able to perform FFT on the input signals, which is not required in this report but could definitely be used for future work.

A PMBus converter drives a load and therefore an electronic load is used as load. The requirements are that it is remotely accessible and can sink current up to 60A. The Chroma 6312 is a mainframe that stores two electronic loads. The mainframe is remotely accessed with GPIB as the oscilloscope and the power supply. By having all instruments using GPIB the development time is reduced. The major advantage for choosing the Chroma 6312 as load is that the electronic loads it stores can be substituted while still controlling the mainframe with the same GPIB implementation. So, if the requirement on the sink current would change it is possible to put another load into the mainframe without having to redo the GPIB communication system.

3.4 Controlling the PMBus and test instruments

The system needs a main program and a controller component that controls the other instruments to work together. In this section several approaches are presented and finally one is chosen to be the approach used in this report.

The first approach to discuss is to use a microcontroller. Then the main program is a software downloaded into the memory of the microcontroller. The microcontroller needs to implement a GPIB communication with the instruments and a PMBus communication with the test object. The advantage of using a microcontroller is that one gets total control of everything that is transmitted on the PMBus. Another advantage is that with this approach it is possible to make a small complete product and make a printed circuit board with the complete product. However, it is required to give some test result as a log file to the user and this can not be done in an easy way without a complex solution. The complexity of the main program with a microcontroller approach is of course a major disadvantage.

Instead of dealing with the complexity of writing the software code by hand one could instead use automated generated code tools, such as Matlab or LabView. These tools help engineers to develop a program much faster than traditional hand coding. However, the generated code has more complexity [32].

The major advantage for choosing LabView in this report is that LabView feature instrument drivers for hundreds of instruments. Because National Instruments distributes LabView

drivers in source code one can easily modify the code for a particular instrument or combine code for several instruments. LabView feature instrument drivers for both the oscilloscope and the power supply that was discussed in previous section. Another advantage is that LabView contains a graphical editor one can use to graphically build a user interface. From menus, one can select predefined buttons, knobs, slides, graphs, charts, etc. This editor makes it easy to make a GUI [33].

Matlab supports GPIB communication through the Instrument Control Toolbox [34]. However, there are not available complete instrument drivers for the oscilloscope and the power supply. This means that instrument drivers need to be developed if Matlab is used. Finally, the Matlab program can only run on Matlab platform so that portability is poor and the function of graphical user interface is not flexible enough [35].

In table 1 all the advantages and disadvantages are presented. The advantages with available instrument drivers are clearly a good reason why LabView is chosen in this report as a system controller.

	Advantages	**Disadvantages**
Microcontroller	Total control of PMBus. Area, make a small complete circuit for the whole test.	Very Complex solution and time consuming.
LabView	Instrument drivers are available. Easy to create a GUI.	Complex generated code. A PC is needed.
Matlab	Supports GPIB communication	Complete instrument drivers are not available. Not easy to build a GUI. A PC is needed and the portability is weak.

Table 1: *Comparison of different system controllers.*

4 Test system

In this chapter the basic software components that builds the system is presented. In the previous section the components were chosen and the final system is illustrated in figure 9.

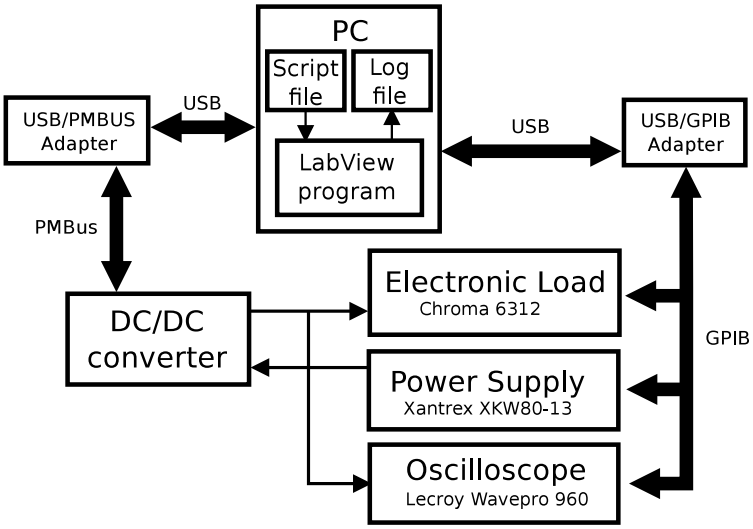


Figure 9: *A block diagram of the system.*

The PC in the system runs a LabView program and that program controls what tests should be issued based on a script file. The script file contains a list of script commands that defines the test scenario and test parameters. The script language is described in section 4.2. After the program has executed the test results are presented in a separate log file. This LabView program is the main program that runs continuously and is presented in more detail in section 11.
Test instrument that are used (electronic load, power supply and oscilloscope) needs necessary LabView drivers so that they can be remotely configured with the GPIB interface. These drivers are described in this chapter.

4.1 Main program and graphical user interface

The main program controls the execution of the program by calling other programs, writes the test results to a log file and it includes the GUI which is presented in figure 10.

Figure 10: *Graphical User Interface (GUI) for the tester.*

In the GUI the test engineer can choose which script file should be executed, choose what name the log file should have and see how far the execution has come in terms of percentage. The percentage is calculated by looking at what line in the script file is executing and how many lines there are in total. In this section the main program and its functionality is described. The main program block diagram is given in figure 11. As seen in the block diagram, the main program is basically a while-loop that for all loop iterations parses a script file, executes a command and writes the result to a log file.

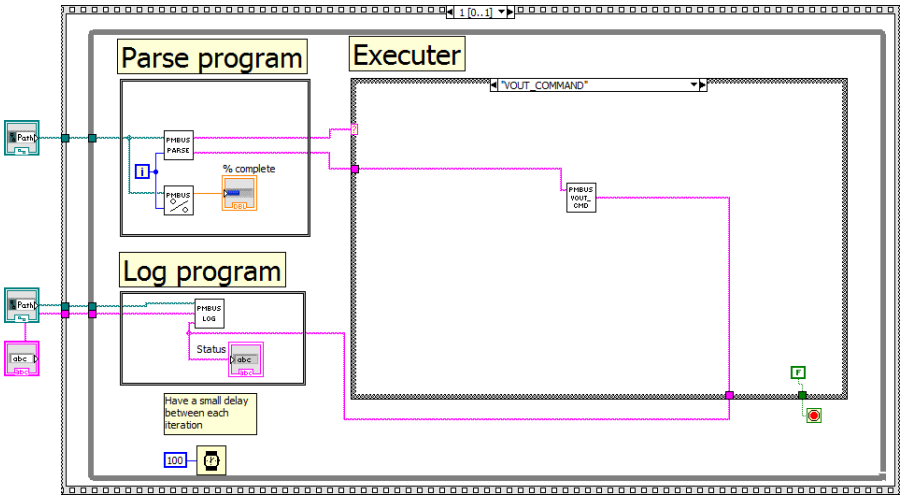


Figure 11: *The main program. A parser reads a script command and executes the corresponding program. A log program stores all the test results to a text file.*

All PMBus commands that could be issued are implemented as separate sub-programs. Each PMBus command sub-program is needed to be able to configure the test device and perform compliance tests to verify the device functionality. There are also sub-programs that perform analytical and verification tests on the device's output.
Sub-programs are called from the main program and when they are called they will start to execute. The order of how the sub-programs are called is decided by a script file that lists commands on separate lines. The main program needs to read the script file to decide what sub-program should be called. This is done by a parser program that is illustrated in figure

12.
The parser program reads the script file and picks the line that should be parsed. The line that should be parsed is determined by an input from the main program which knows how far the script has already been executed. The parser then search for a space punctuation in that line since a command is separated with its parameters with a space punctuation. The parser sends the command and its parameters back to the main program.

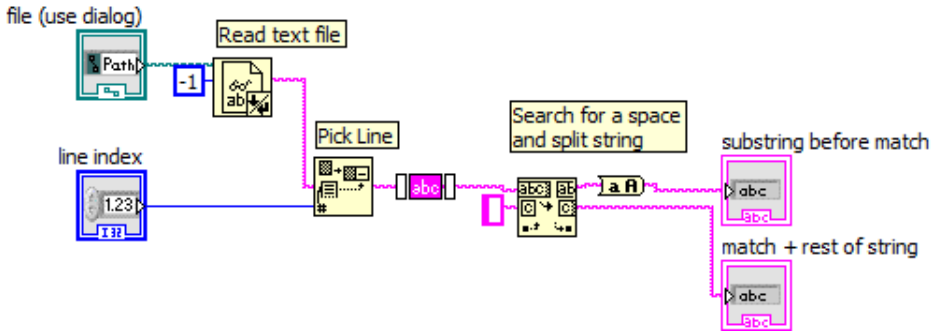


Figure 12: *A parser that parse a script command. From the main program the script file and the script line number to read is given. The line is read from the script file and then the first space punctuation is searched for. The program then returns the search match before the space punctuation as an output and the rest of the line as another output. The first output is the script command name and the second output is the command parameters.*

After the parser program has executed, the command is input to the case-sequence and the main program starts the corresponding sub-program. The command parameters are passed to the sub-program where the sub-program has to parse the parameters by itself. So, the main program only parses the command and passes the parameters to the sub-program that should be executed. By doing this, each command can has an individual number of parameters and a new command can be implemented as a new sub-program without having to change anything in the main program parser.

When a sub-program has finished its execution, it passes a text string with result back to the main program. The main program then passes the text string forward to a log program which writes the text string to the end of the log file. The log program's block diagram is given in figure 13.

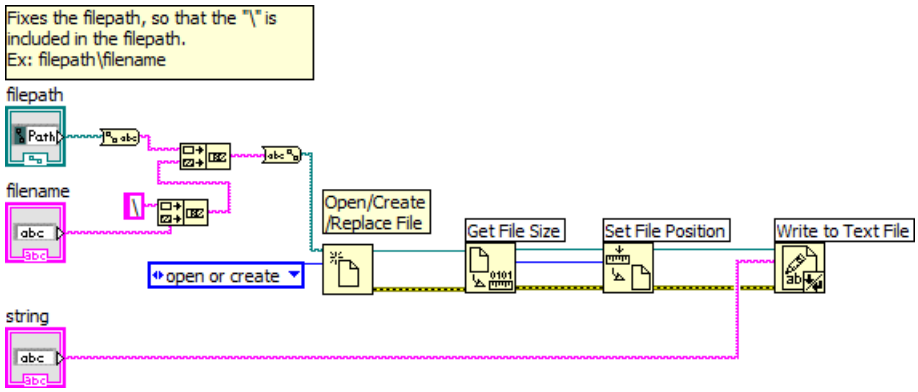


Figure 13: *The log program writes a text string to the end of the log file.*

The main program has finished its execution when the END script command is parsed by the parser and the execution block stops the while-loop.

4.2 Script language

As mentioned earlier, the script file determines the execution order of the test program and each test program can be easily modified by rewrite the script file. A script file consists of several script commands that are given in a list where each command will be executed in top-to-bottom order. A command can be a PMBus command, a test related command, instrument configuration command or an analytical command. What type of command it is does not matter and several can be combined in all kinds of way as long as each command is on a separate line. Each command is defined by a command name that is the first word in a line. All commands in the script language needs an arbitrary number of parameters and each is given on the same line after the command name. A script command is given on the form in (3).

$$Command\ \ parameter_1\ \ parameter_2\ \ parameter_3\ \ ...\ \ parameter_N \tag{3}$$

The main program parser in figure 12 parses the command name to find out which sub-program to execute and the rest of the line after the command name is passed to the sub-program where each parameter is parsed. This makes it possibly to have different number of parameters for different commands. The main program executes its while-loop until the mandatory END script command is parsed. This script command is the only command that does not need any parameters and is usually at the end of a script. It could also be placed wherever in the script to comment out the rest of the consecutive commands.

4.3 Transmitting PMBus commands

In this section a description of how to transmits data messages over the PMBus are given. When the main program parses a PMBus command it starts the corresponding sub-program and also passes the containing parameters to the sub-program. The sub-program then parses

the parameters and transmits the PMBus command using the USB Interface Adapter that was introduced in section 4.3.
Each PMBus command can be divided into four categories; read-byte, write-byte, read-word and write-word. In reality, each PMBus is implemented as separate programs but they are almost implemented in the same way. So, for simplicity reasons this section describes how these four categories are implemented in general.
The USB Interface Adapter comes with a DLL file that should be located in the same file directory where the sub-programs are located. Then a LabView "Constructor Node" block is used to create an instance of the DLL file and get access to functions that could be used with the adapter. The functions that are used from the DLL file are *pmbusRead* and *pmbusWrite*.

The byte commands transmits or reads a byte from a PMBus device. The device is determined by the address. So, a script command that transmits data to the PMBus needs to define the device's address and the data message as script command parameters. The main program sends these parameters and they are received by the sub-program's parser, which is illustrated in figure 14.

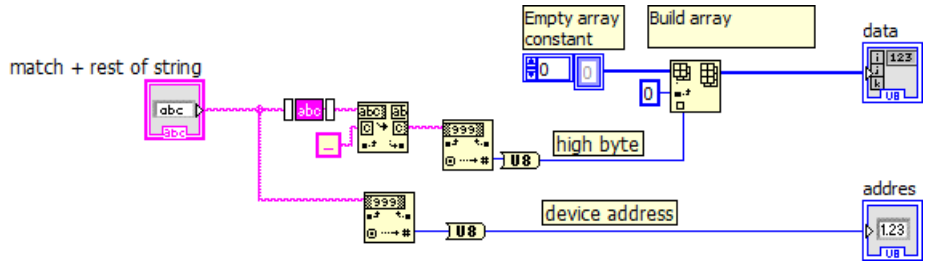


Figure 14: *A program that parses a script command's parameters to extract the PMBus device's address and the data byte that should be transmitted. The input to this program is the list of parameters that was given from the previous parser.*

The difference from reading and sending a byte with a word is that an extra parameter should be parsed and converted to an eight bit data. The parser for this is given in figure 15.

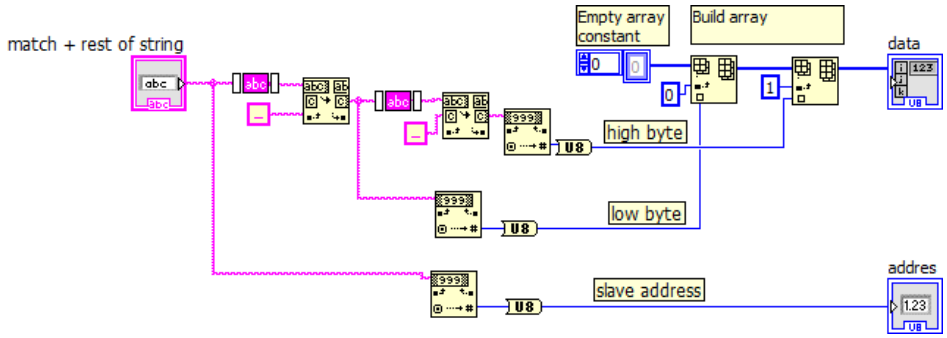


Figure 15: *A program that parses a script command to extract the PMBus device's address and the two data bytes that should be transmitted. The input to this program is the list of parameters. The input to this program is the list of parameters that was given from the previous parser.*

The parsers searches the text given by the main program for the device address and the data that should be written to the device. The address and the data are converted from text to eight bit data bytes. Then these two values are passed to the sub-program that will do the actual PMBus communication. Note that this is only a generic parser for PMBus commands and for other commands there are other similar parsers.

A ”Constructor Node” block is used to create a PMBus instance from the USB Adapter Driver.dll file. Then an ”Invoke Node” is used to call the function *pmbusWrite* or *pmbusRead*. To read a byte the *pmbusRead* function is used and to write a byte to a device the *pmbusWrite* function is used. These two functions have four inputs that need to be defined. The first input defines the device address and is wired to the address that comes from the parser. The second input defines which PMBus command should be issued and this is made as constants in each sub-program. The third input is the data that should be written, this is not necessary for the *pmbusRead* function. The data is wired to the data that comes from the parser. The fourth input is a constant that will be set for each sub-program and constant differs from when reading or writing a byte or a word. If a byte is read then this constant is *readByte* and *writeByte* when a byte is written. If a word is read then this constant is *readWord* and *writeWord* when a word is written. In figure 16 four block diagrams represents how to read and write data with the USB Interface Adapter.

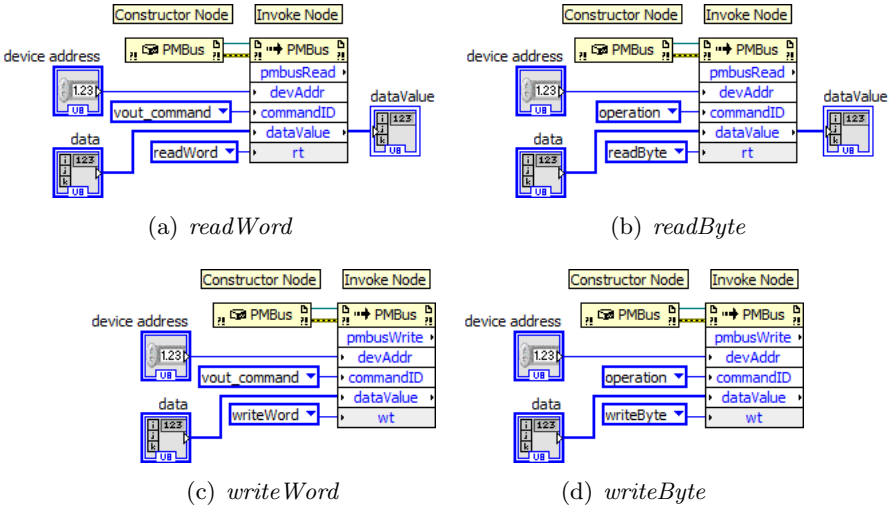


Figure 16: *The programs that are used to transmit and receive data on the PMBus.*

4.4 Instruments and GPIB

In this section an introduction to the *General Purpose Interface Bus (GPIB)* interface is given. After the introduction the following sections presents how to use the GPIB interface together with the instruments used in this report.

The formal name of the GPIB is the ANSI/IEEE Standard 488.1 and is a standard for communication with instruments [36]. It is the primary instrument control interface due to its proven reliability and it is estimated that there are over 10000 different GPIB instruments in use today [37]. GPIB is an eight bit parallel communication interface that supports transfer

rates of 1 Mbyte/s and higher. There is a system controller and additional instruments. In this report the system controller is a computer and there are three additional instruments. In figure 17 the GPIB system used in this report is illustrated.

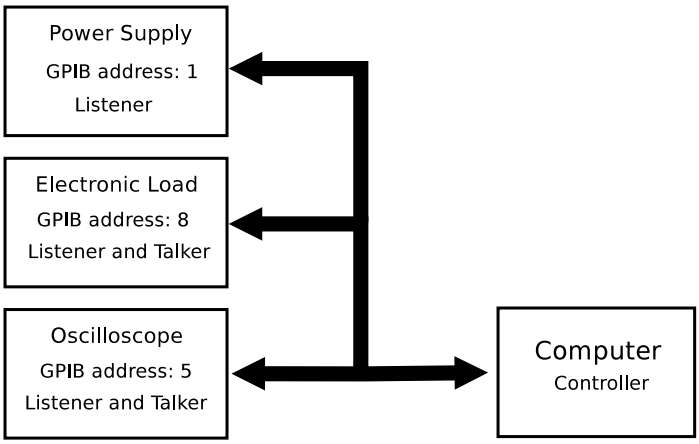


Figure 17: *A block diagram presenting the GPIB system used for the tester.*

Instruments that support GPIB can be *talkers*, *listeners* or *controllers*. A controller manages the communication and the information on the bus, a talker sends out data to the bus that a listener receives. In figure 17 the computer is a controller, the power supply is a listener and the oscilloscope and the electronic load are both talker and listeners.

All GPIB devices are uniquely identified by assigned addresses. An instrument can usually be assigned an address by switches on the back of the instrument. The address is eight bits, where the five least significant bits is the unique number that ranges from 0 to 30. Then there is the TA (Talk active) bit that is assigned to instruments that are talkers and the LA (Listener Active) bit that is assigned to instrument that are listeners. Figure 18 shows the address configuration.

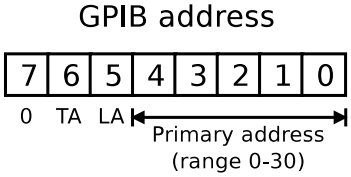


Figure 18: *GPIB address configuration.*

There are 16 signal lines in GPIB bus. Eight lines are the DIO data bus lines that carry the data messages. Three hand-shake signals are used to assist the transfer of each byte. The rest lines are five interface signals that are used to manage an orderly flow of information across the interface. Further details about the GPIB interface are given in IEEE Std 1174-2000 [36].

LabView can be used together with VISA, which is a high-level API that calls a lower level driver. VISA is capable of controlling a GPIB communication and makes appropriate driver calls.
The lower level driver that is called for GPIB communication is the ”NI-488.2 for Windows

DLL" file. VISA can also use two other types of communication techniques, Serial and VXI, which are not discussed here. The advantage with VISA is that whether the interface is GPIB, Serial or VXI the same operations to communicate with the instrument are used [33]. Figure 19 illustrates how VISA provides interface independence. VISA and NI-488.2 needs to be installed together with LabView to use the GPIB-interface presented in this report.

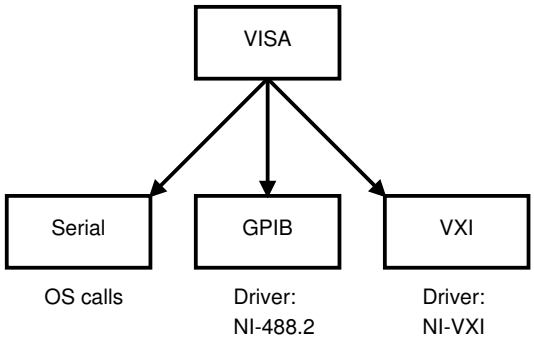


Figure 19: *VISA can control serial, GPIB or VXI instruments. VISA calls the corresponding driver depending on the type of instrument.*

VISA provides many complete functions that can be used in LabView but there are only four that is needed in this report. These functions are *Open*, *Read*, *Write* and *Close*. The *Open* function opens a session and establishes a communication with the instrument. The *Close* function properly closes the opened session. The *Read* and *Write* functions receive and transmits data on the interface (the GPIB bus in this case) [38]. In figure 20 an example program that writes and reads to an GPIB instrument in LabView is given.

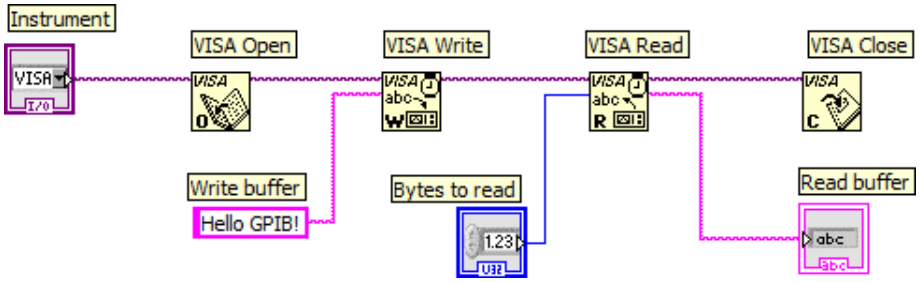


Figure 20: *A VISA example program that reads and writes a text string to the GPIB bus. The instrument involved in this communication is defined by the GPIB address that is given as input.*

4.4.1 LabView and electronic load GPIB-interface

The electronic load's manual [39] describes all GPIB commands that it features. In this report, only a couple commands are used and they are described in this section. There is no available LabView driver for the instrument so a LabView program has to be programmed by hand. One advantage is that it is relatively easy to implement since all that is needed is to look up the appropriate command in the instrument manual and perform a VISA Write

function call and write that command to the GPIB bus.
The instrument needs to be configured to sink a current that is given as an input to the LabView program. Also, the instrument is a mainframe that stores several electronic load channels so the LabView program needs to have an additional input that defines the load channel. This functionality is all that is needed for the LabView program in this case.

For the first configuration, the specific load channel in the mainframe should be defined. The GPIB command ":CHAN N;" does exactly this, where N is an integer and determines the exact mainframe channel id. For the LabView program this text string is appended with the channel input, which is converted to a string data type. The complete text string is then written to the correct GPIB address. A block diagram of this is shown in figure 21. After this command, every succeeding command to the instrument will only affect the mainframe channel that was configured with this program.

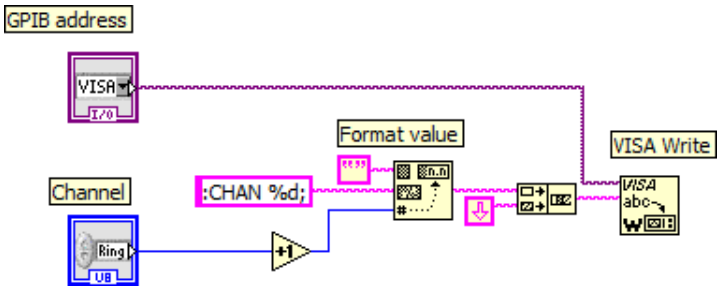


Figure 21: *Configure electronic load channel.*

The next step is to configure the load to sink a given current. This is not as straightforward as configure the load channel, since the operation mode first needs to be defined. There is three available operation modes; *constant current (CC)*, *constant resistance (CR)* and *constant voltage (CV)*. In CC mode the load sinks a given current regardless of the input voltage, in CV mode the load sinks current to control a voltage at a given level and in CR mode the load will sink a current linear proportional to the input voltage. The CC mode is the appropriate operation mode in this case. It can be programmed to sink current in two current ranges, Low and High. The Low range provides better resolution but the range is to low and could not sink current that is appropriate for a general DC/DC converter. Therefore, the operation mode is CC with High range. The GPIB command for configure this operation is ":MODE CCH;". This command is easily transmitted to the instrument by write the command as a text string to the GPIB bus and a block diagram is presented in figure 22.

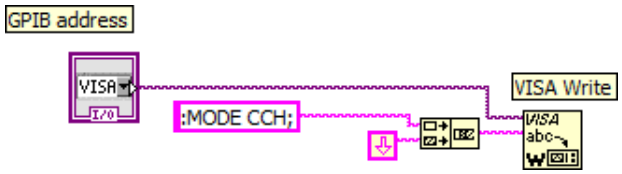


Figure 22: *Configure electronic load to use the constant current operation mode with high range.*

When the operation mode is configured one can configure the instrument's sink current.

The sink current can be configured to operate static (STAT) or dynamic (DYN). The static operation is the typical operation and is used in this report. These two operation functions are described in the manual but it is not necessary to understand them to continue reading this section. Also, there is the possibility to have to current levels (L1 and L2) that the load is switching between. However, since only a static current is needed, only the first level (L1) needs to be configured.
The GPIB command that is transmitted to the instrument is ":CURR:STAT:L1" followed by the current. The current is given as an input in the double data type and needs to be converted to a string data type with the "Format Value" function. The command is then transmitted to the GPIB bus with the "VISA Write" function. The block diagram in figure 23 configures the sink current level to the load.

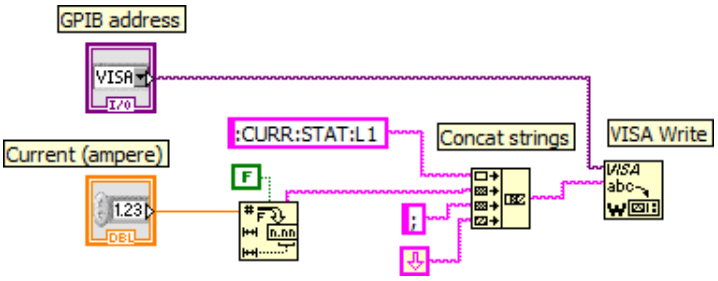


Figure 23: *Configure the electronic load current to be static.*

When the load is configured correctly it is should be activated with the new configuration. This is done by sending the ":LOAD ON;" command to the instrument. Figure 24 shows how to activate the load.

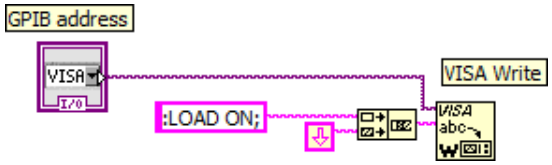


Figure 24: *Configure the electronic load to start sink current.*

These block diagrams that has been described in this section is combined into one LabView program that configures the load instrument. A separate parser is implemented to parse a script command to the inputs used in this program. The main program calls this program when it parses a "CONFIG_LOAD" script command (4). To this script command the current is given as a parameter.

$$CONFIG\_LOAD \;\; current \tag{4}$$

4.4.2 LabView and oscilloscope GPIB-interface

The oscilloscope features a LabView driver [40] that is used in this report. The driver includes functions that configures the oscilloscope trigger, acquires measurements and other configu-

ration setups. Each driver function performs necessary GPIB commands automatically. The driver must be installed to the LabView directory before it can be used. There are several functions from the driver that is used and they are described in this section.

A GPIB communication session is started with the ”lcwpxxx Initialize” function and it is then closed with the ”lcwpxxx Close” function. The specific GPIB address is given as input to these functions so that the driver knows which address is associated with the oscilloscope.

In figure 25 a program that initialize the oscilloscope is illustrated. After the GPIB has been initialized the oscilloscope is configured to continually acquire signals with the ”lcwpxxx Configure Initiate Continuous” function. Then the oscilloscope acquisition type is configured which could be in *normal* or *sequence mode*. Normal acquisition mode is the typical operation mode and used here. The next step is to choose which oscilloscope channel that is for interest. That is, which oscilloscope probe is connected to the specific signal that is supposed to be observed should be choosed. If several oscilloscope channels are used each channel needs to be enabled and configured. In the same step the signal can be configured to have an offset and attenuation. The signal can be zoomed in by changing the vertical range. This is all configured with the ”lcwpxxx Configure Channel” function. The last step is to close the GPIB session so that VISA is available for another session.

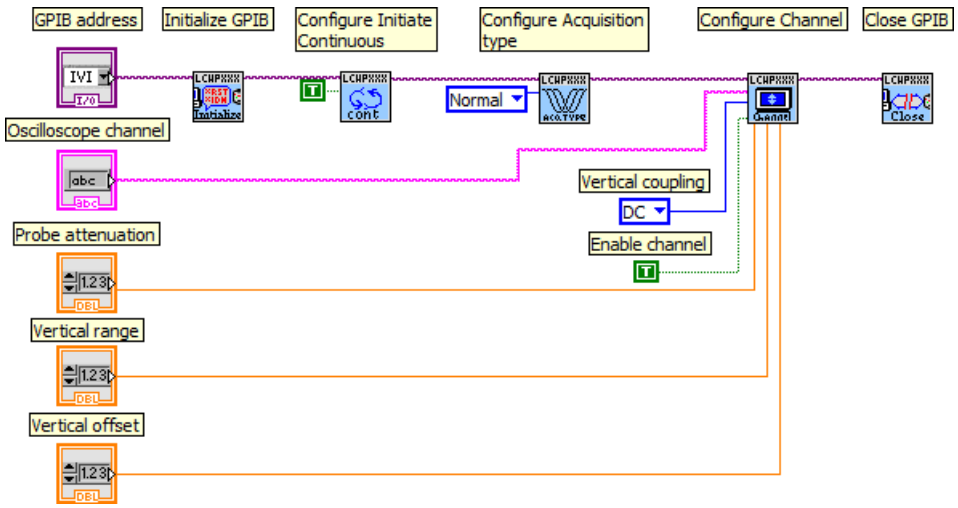


Figure 25: *LabView program that configures the oscilloscope. The oscilloscope channel is enabled and configured.*

For the compliance testing of a PMBus device it is necessary to perform measurements on the device’s output. This is done with assistance from the oscilloscope which then needs to be configured to trigger on the special measurement function. Typical oscilloscope measurement functions are fall time, rise time, amplitude, voltage average, delay, frequency etc. In figure 26 an example program illustrates how to configure the trigger type and in this example the trigger is configured to capture a positive rising edge. Typical trigger types used in this report are rising edge, falling edge, positive and negative slew. The trigger type is configured with the ”lcwpxxx Configure Trigger” function. Each trigger type has a separate configuration function that should be called after the trigger type has been configured. This is because each trigger type needs to be configured differently. In this example the edge trigger needs to

be configured as positive or negative edge and at which trigger level it should trig. Also, the edge trigger needs to know which oscilloscope channel that is the interesting signal for the trigger.

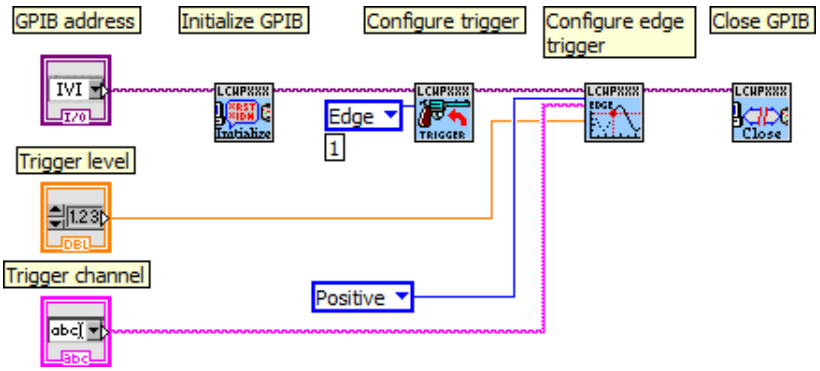


Figure 26: *A LabView program that configures the oscilloscope. The oscilloscope trigger is configured as an edge trigger.*

Now, the oscilloscope is configured and the trigger is configured to capture the signal characteristics that are supposed to be measured. The measurement is retrieved from the oscilloscope with the "lcpxxx Fetch Waveform Measurement" function. Note that the waveform has to be captured before calling this function so a delay could be introduced to wait for the trigger to capture the waveform. The function takes as input the oscilloscope channel that has the signal of interest and the measurement function that should be performed. The function then outputs the result from the measurement function. In figure 27 a program that retrieves the fall time from the oscilloscope is given.

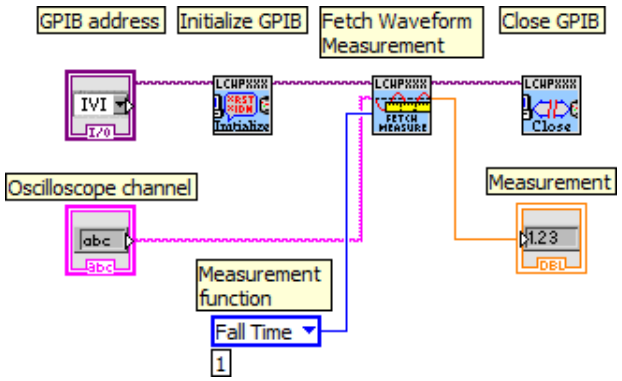


Figure 27: *The waveform is fetched from the oscilloscope through the GPIB interface. In this case, the fall time measurement is retrieved.*

In this section the LabView driver for using the oscilloscope was introduced. How to configure the trigger and the oscilloscope in general was described. A program that retrieves measurement from a triggered waveform was presented. The output measurement is used by other programs that performs operations on the measurement.

4.4.3 LabView and power supply GPIB-interface

The power supply should be able to be reconfigured to output another given voltage level. In this section a LabView program that reconfigures the power supply is presented.

The power supply Xantrex XKW80-13 features a LabView driver [41] for controlling the instrument. This means that one does not need to know how the GPIB commands controlling the instrument are defined. Instead, one can use the driver functions that do the GPIB commands. The driver should be downloaded and installed to the local LabView directory before it can be used.

The power supply can be configured to operate in two operation modes, *voltage mode* and *current mode*. The voltage mode is used and configured by having a voltage level higher than the current level [42]. Therefore, the program needs to configure both a voltage level and a current level where both are given as inputs so that the user can adjust the levels.

The power supply features a fault management function to protect against any over voltages. By setting an *Over Voltage Point (OVP)* the power supply will turn off automatically if the output voltage level is higher than the OVP. Figure 28 shows a complete block diagram of the LabView program that configures the power supply. The "Initialize" function opens a GPIB session and the "Close" function closes the session. The other functions sets the OVP, configures the voltage and the current level.

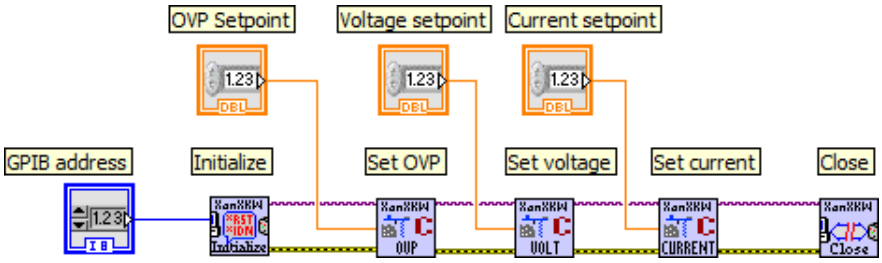


Figure 28: *A LabView program that configures the power supply.*

A separate parser is implemented to parse a script command to the inputs used in this program. The main program calls this program when it parses a "CONFIG_POWER" script command. The script command is given in (5) where *ovp* is the over voltage protection limit, *voltage* is the voltage level and *current* is the current level.

$$CONFIG\_LOAD \;\; ovp \;\; voltage \;\; current \tag{5}$$

4.5 System summary

This chapter described the overview of the total test system. First, in section 4 the main program was presented. The main program is a LabView program that controls the flow of the execution and calls other programs depending on what script line is parsed. After the called program has executed the main program writes the result to a log file. The log file and the script file are selected from the GUI.

In section 4.2 the script language and how to parse a script file was described. Each script file consists of a list of script commands where each command is in one line. For example, a script command can be a PMBus command that configures the test device being tested or a verification test command.

How PMBus commands are transmitted was described in section 4.3. A DLL file is used to transmit and to receive data bytes. Then the instrument control programs were described in section 4.4. The electronic load does not provide any LabView drivers so a GPIB interface program is needed. From the instrument manual the necessary GPIB commands can be found. Then these GPIB commands are implemented in a program that configures the instrument. Both the oscilloscope and the power supply provide a LabView driver that is used to configure the instruments.

5 Verification test programs

In this chapter a set of verification programs are presented. They are important to verify that a PMBus device behaves as commanded. Along with all PMBus commands these verification programs can be issued with script commands. This means that a script file contains both PMBus commands that configure the device and verification commands that verify the behavior of the device. Without these script commands presented in this chapter it would not be possible to actually test a device's output properties automatically.

These programs verify a measurement, a status register or a device's output characteristics and determine if the device has passed or failed the test. The result of the tests is a string containing "Test PASSED" or "Test FAILED" which is passed to the log program that writes the string to the log file.

5.1 Verifying the output DC level

A PMBus device output voltage level could be configured with the $VOUT\_COMMAND_{0x21}$ command as described in section 2.6. The compliance test program needs to provide a test program that verifies the new configured voltage level to be within a defined margin. In this section two such test programs are presented.

The first test program is the CHECK_DC program. This program is called when the main program's parser identifies the "CHECK_DC" label in the script file. The script command for calling this program is defined as

$$CHECK\_DC \;\; expected \;\; margin \;\; channel \tag{6}$$

where *expected* is the expected voltage level, *margin* is the acceptable margin that the voltage level can be within and *channel* is the oscilloscope channel that probes the device's output. This script command can be used immediately after the device's output level has been changed or to verify that a device has responded correctly to a fault by turning off its output.

The approach is to acquire the voltage level from the oscilloscope. This is simple since the oscilloscope does not need to configure any trigger since the voltage level is always presented in the waveform. All that is needed is to request the average voltage level from the oscilloscope via the GPIB bus. This is done by first initialize the bus communication with the "lcwpxxx initialize" function. Second, the voltage level is acquired with the "lcwpxxx read waveform measurement" function and as input to that function is the "Voltage Average" option. Finally, the GPIB session is closed with the "lcwpxxx close" function. The program that acquires the voltage level is presented in figure 29. In this figure, the oscilloscope channel is given from the script command and the other inputs are defined as constants.

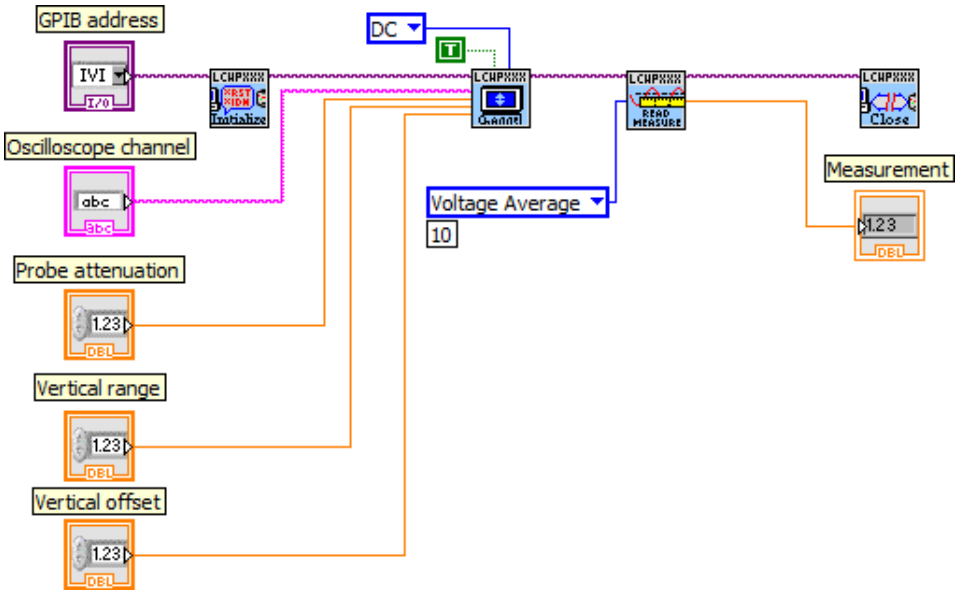


Figure 29: *This program retrieves the average voltage level from the oscilloscope. The voltage level is the result measurement output.*

The test program has not yet verified the voltage level. The measurement is given as input to the program in figure 30 where the actual verification occurs. This program is called the *compare-to-expected* program and is refered to several times in this report. The measurement is compared to an expected value with a defined margin applied. The program returns the string ”TEXT PASSED” or ”TEST FAILED” to the main program that then writes the string to the log file.

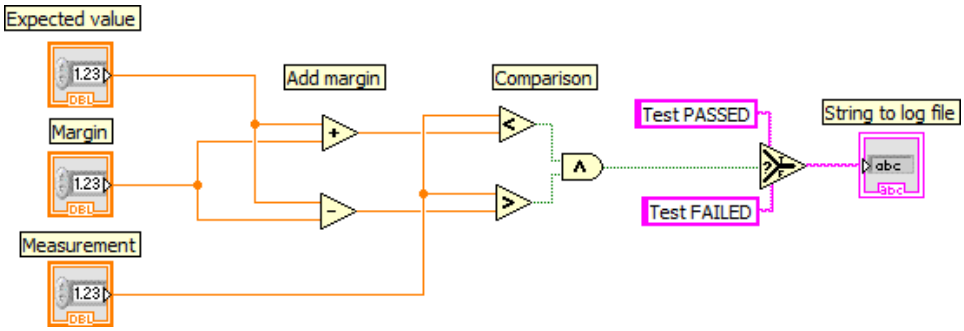


Figure 30: *The Compare-to-expected program. It takes as input an expected value, a measurement value and a margin. The expected value is compared with the measurement value with added margins. The program returns one of two different text strings which depends on the comparison result.*

The voltage level can also be retrieved form the device with the PMBus command $READ\_VOUT_{0x8B}$. The device then has its own voltmeter that senses its output. This command could be used by a system host that monitor and controls all the PMBus devices within that system. The system host needs to know that the device returns the correct voltage level otherwise the

host could take wrong decisions based on faulty voltage levels. Therefore a test program that verifies the $READ\_VOUT_{0x8B}$ command is needed. A test program that is used is the CHECK_READVOUT that is called from the script file with the following line

$$CHECK\_READVOUT \;\; adr \;\; exponent \;\; margin \tag{7}$$

Where *adr* is the PMBus address for the device, *exponent* is the number of bits that are used for the LINEAR data format exponent and *margin* is the acceptable margin.

The CHECK_READVOUT program is given in figure 31. The program first transmits the PMBus command $READ\_VOUT_{0x8B}$ and returns the device's view of the voltage level as a 16 bit word. The 16 bit word has the LINEAR data format that is described in section 2.3. Secondly, the program converts the word to a numeric value that is comparable with a numeric measurement. Finally, the compare-to-expected program in figure 30 is reused to compare a measurement value with an applied margin.

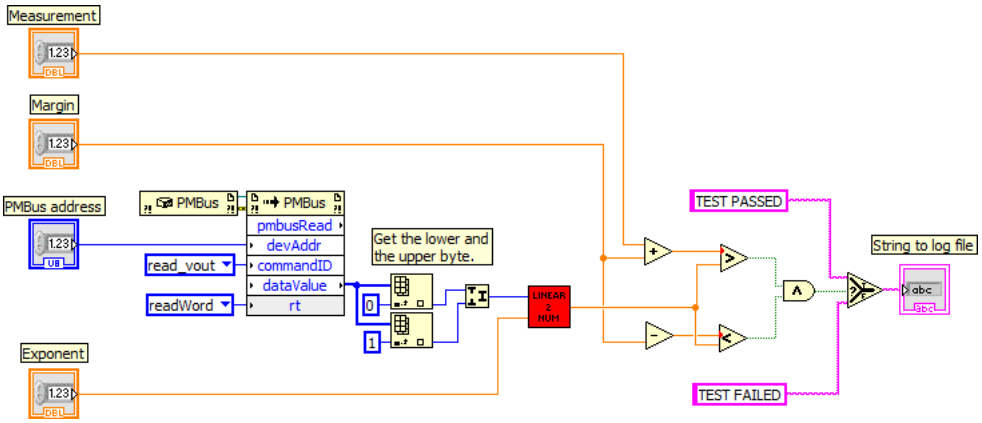


Figure 31: *This LabView program compares an input voltage with the voltage retrieved from the PMBus command CHECK_READVOUT.*

5.2 Verifying rise and fall times

With the PMBus commands TON_RISE_{0x61} and $TOFF\_FALL_{0x65}$ the rise and fall time for the device's output is configured. These two time configurations can be used to turn a device on or off softly. They can also be used for synchronizing several devices. In this section, two test programs that verifies the rise and fall time are presented. The test programs are called with the following script commands:

$$CHECK\_RISETIME \;\; adr \;\; ch \;\; exp \;\; marg \;\; tlo \;\; thi \tag{8}$$

$$CHECK\_FALLTIME \;\; adr \;\; ch \;\; exp \;\; marg \;\; tlo \;\; thi \tag{9}$$

Where *adr* is the PMBus device's address, *ch* is the oscilloscope channel that probes the device's output, *exp* is the expected rise or fall time, *marg* defines the margin that is acceptable and *tlo* is the lower trigger level and *thi* is the higher trigger level. The oscilloscope requires

that the trigger levels are defined and this is why the script commands also require the trigger levels.

Both of the test programs are divided into four steps. The first step configures the PMBus device to use sequencing with the PMBus command $ON\_OFF\_CONFIG_{0x02}$. If the device is not configured to use sequencing the fall and rise time is very fast since the device then tries to change the output level as fast as possible. When the device is configured to use sequencing it uses the rise and fall time configured with the TON_RISE_{0x61} and $TOFF\_FALL_{0x65}$ commands. The device is turned on or off depending on which of the rise or fall time is supposed to be tested. If the fall time is tested, the device is turned on from the beginning and then turned off. The rise time is tested in the opposite way; the device is first turned off and then turned on. This start state is also configured in the first step. In the second step, the oscilloscope is configured to trig for a slew. In figure 32 the oscilloscope is configured for a slew trigger.

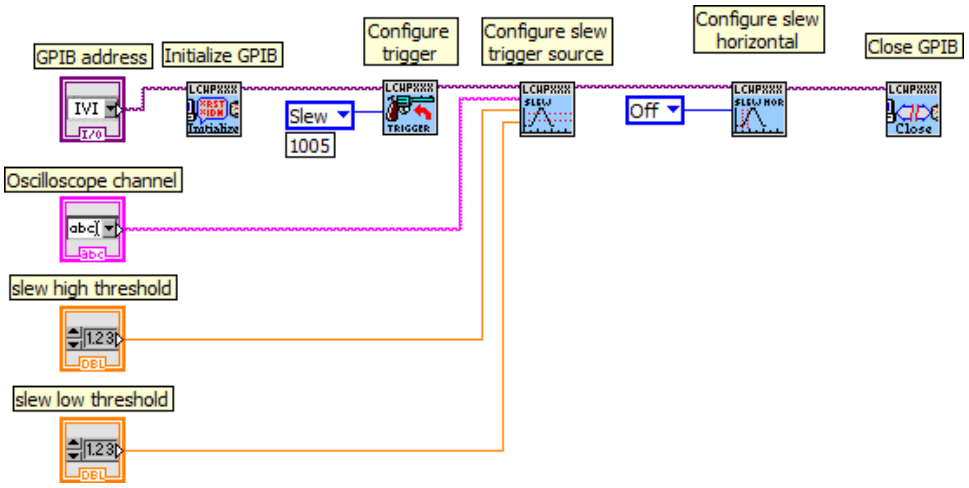


Figure 32: *In this LabView program, the oscilloscope trigger is configured as a slew trigger.*

In the fourth step the oscilloscope trigger is configured and the test program starts the transition. The device's output is turned off for a fall time verification and turned on for a rise time verification. The transition should be triggered by the oscilloscope. The rise and fall time is then acquired and compared to the expected value in the fifth step. In the fifth step a piece of program that reads a measurement from the oscilloscope and acquires the fall or rise time is used. Finally, the measurement is compared to the expected value with the compare-to-expected program in figure 30.

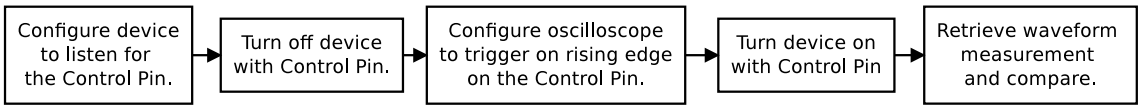
5.3 Verifying ON and OFF delays

When a PMBus device is turned on or off a delay can be introduced with the commands TON_DELAY_{0x60} and $TOFF\_DELAY_{0x64}$. These commands can be used to synchronize several devices and for protection. In this section two test programs are presented. The test programs are CHECK_ONDELAY and CHECK_OFFDELAY and they are called by the following script commands:

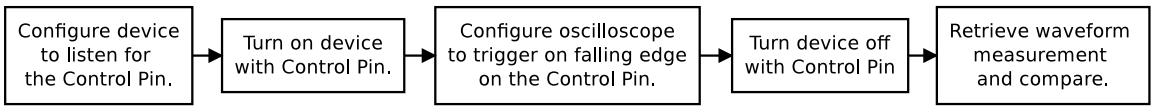
$$CHECK\_ONDELAY \ \ adr \ \ ch1 \ \ ch2 \ \ exp \ \ marg \tag{10}$$

$$CHECK\_OFFDELAY \ \ adr \ \ ch1 \ \ ch2 \ \ exp \ \ marg \tag{11}$$

Where *adr* is the PMBus device's address, *ch1* is the oscilloscope channel that probes the device's output, *ch2* is the oscilloscope channel that probes the PMBus control pin, *exp* is the expected delay and *marg* is the acceptable margin. Why all these parameters are needed is explained in this section. These test programs are divided into five steps illustrated in figure **??**.



(a) CHECK_ONDELAY



(b) CHECK_OFFDELAY

Figure 33: *Block diagrams of how the CHECK_OFFDELAY and CHECK_ONDELAY are implemented.*

The first step, configures the device with the address *adr* to be turned on and off with the control pin and ignore any $OPERATION_{0x01}$ commands. This is because the oscilloscope can not catch the delay by trigger on the output. The oscilloscope trigger when a slew or edge is detected and the delay is then missed. Instead, the oscilloscope could trigger on an edge on the control pin and the oscilloscope will capture the delay. This is why the device is first configured with the $ON\_OFF\_CONFIG_{0x02}$ command.

In the second step, the device is turned on or off depending on if it is the TON_DELAY_{0x60} or the $TOFF\_DELAY_{0x64}$ that should be tested. If the TON_DELAY_{0x60} is tested, the device is first turned off so that than when it can be turned on again. The $TOFF\_DELAY_{0x64}$ is tested in the opposite way, first the device is turned on so that it can be turned off.

In the third step the oscilloscope is configured to trigger on an edge to capture the delay. The edge trigger configured as either positive or negative. CHECK_ONDELAY configures a positive edge trigger and CHECK_OFFDELAY configures a negative edge trigger.

In the fourth step the device and the oscilloscope is ready, so the transition is started. This is done by flip the control pin so that the device turns off or on. The oscilloscope captures the waveform and the delay is presented in the waveform. However, the oscilloscope does not provide any measurement function that calculates a delay. Therefore, a LabView program that retrieves the waveform from the oscilloscope and calculates a delay is implemented and its block diagram is given in figure 34. This program requires that the oscilloscope already has captured the waveform. The oscilloscope driver provides the "lcwpxxx Fetch Waveform"-function that fetches the waveform captured on the oscilloscope as an array and that array is given as input to a while-loop. The while-loop then scans all elements in the array for a

value that is higher than a certain threshold. The first element that is above this threshold is the element that defines the beginning of a slope. The threshold value is of course different between CHECK_OFFDELAY and CHECK_ONDELAY. When the waveform has passed the threshold level the slope has already begin rising or falling, therefore the element pointer is adjusted by subtract 2. The "lcwpxx Fetch Waveform" function also outputs a value that defines the waveform x-axis increment in seconds per element. The total delay is then calculated by multiplying the element pointer that defines the beginning of the slope and the x-axis increment value.

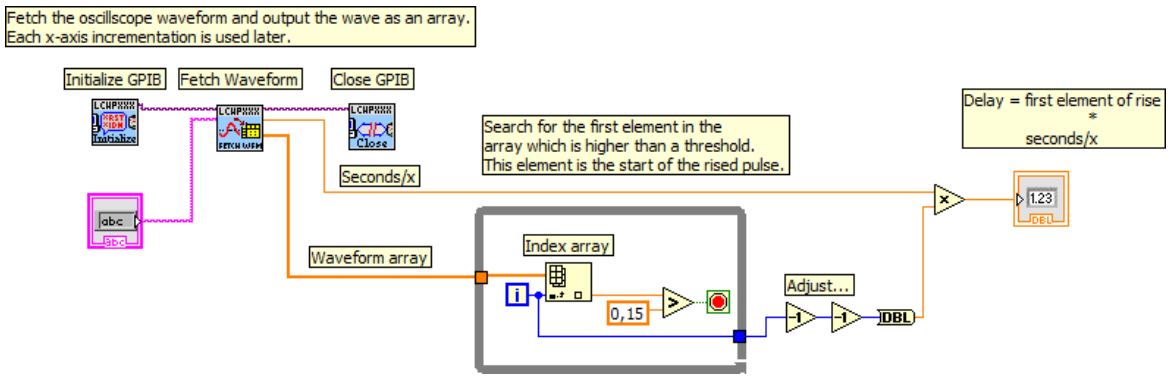


Figure 34: *A LabView program that finds a delay from a waveform.*

Finally, the delay is retrieved and given as input to the compare-to-expected program given in figure 30. This is the fifth and final step by checking the $TOFF\_DELAY_{0x65}$ and TON_DELAY_{0x60} . The compare-to-expected program outputs a text string that either indicate "TEST PASSED" or "TEST FAILED" to the main program that writes this string to the log file.

5.4 Verifying transition rates

When a PMBus device receives the $VOUT\_COMMAND_{0x21}$ command the device's output level is changed. The rate in $mV\mu/s$ which the device's output should change voltage is configured with the $VOUT\_TRANSITION\_RATE_{0x27}$ command. The configured transition rate applies to both negative and positive rates. In this section two test programs that verify the newly configured transition rate is presented. These programs are called by the following script commands:

$$CHECK\_TRANSITION\_POS\ \ adr\ \ high1\ \ low1\ \ high2\ \ low2\ \ exp\ \ marg\ \ ch\ \ tlo\ \ thi \tag{12}$$

$$CHECK\_TRANSITION\_NEG\ \ adr\ \ high1\ \ low1\ \ high2\ \ low2\ \ exp\ \ marg\ \ ch\ \ tlo\ \ thi \tag{13}$$

Where *adr* is the PMBus address for the test device, *high1* and *low1* defines the first voltage level in LINEAR data format, *high2* and *low2* defines the second voltage level in LINEAR data format, *exp* is the expected transition rate, *marg* is the acceptable margin, *ch* is the

oscilloscope channel that probes the device's output and *tlo* and *thi* defines the lower and upper trigger levels. The reason why all these parameters are needed is discussed in this section.

Both of the test programs are implemented as described by the flow chart in figure 35. The only difference is the oscilloscope trigger is configured differently.

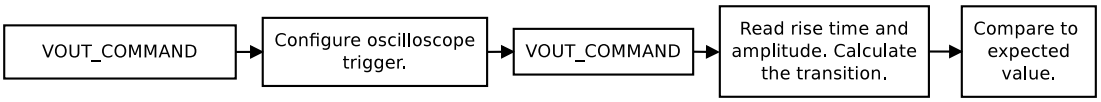


Figure 35: *Block diagram of how the CHECK_TRANSITION command is implemented.*

To test a transition rate the device has to switch from one voltage level to another. Therefore, two voltages levels are needed and defined when calling the test program. For a negative rate, the first voltage level is higher than the second and for a positive rate the first voltage level is lower than the second. The first step in the test programs is to make sure the device operates at the first voltage level by configures the level with the $VOUT\_COMMAND_{0x21}$ command. The second step configures the oscilloscope trigger to capture a slew. This step is what differs from the two test programs. The negative version configures a negative slew and the positive version configures a positive slew. This is the reason why two script commands are needed. When the trigger is ready and waits for a slew it is time to switch to the second voltage level. This is the third step and the device's output is once again changed with the $VOUT\_COMMAND_{0x21}$ command.

The fourth step retrieves the rise time and the amplitude of the captured waveform. The amplitude defines the ΔY and the rise time defines the ΔX. The transition rate is then calculated as

$$Transition\ rate = \frac{amplitude}{1000 \cdot rise\ time} \tag{14}$$

where the factor 1000 is applied to get the seconds in μs. The transition rate calculated in the final step is passed to the compare-to-expected program in figure 30.

5.5 Verifying outgoing currents

In this section two LabView programs that verify a PMBus device's output current are presented. Both of them can be called by writing the corresponding script command in the script file. The following script commands can be used to verify a PMBus device's outgoing current:

$$CHECK\_IOUT\ \ adr\ \ exp\ \ margin\ \ exponent \tag{15}$$

$$CHECK\_READIOUT\ \ adr\ \ exponent\ \ margin \tag{16}$$

Where *adr* is the device's PMBus address, *exp* is the expected current value, *margin* is the acceptable margin and *exponent* defines how many bits are used as exponent in the

LINEAR data format described in section 2.3. Both scripts verify the current but differently. CHECK_IOUT compares the expected current with the current that is retrieved from the PMBus command $READ\_IOUT_{0x8C}$. This script command assumes that the device is able to perfectly measure its own outgoing current without any help from external measurement instruments. The downside with this script is that if the device's current sensor is not perfect, the test is not fully valid. Therefore, the CHECK_READIOUT script command is provided as an alternative. This script command uses the electronic load to measure the outgoing current and compare that current with the result from the PMBus command $READ\_IOUT_{0x8C}$. The LabView program that implements this script command is presented in figure 36.

The program is implemented with a stacked-sequence block where each frame is executed in frame-by-frame from left to right. The current is retrieved from the electronic load by transmitting two GPIB commands. How the electronic load GPIB communication works is described in section 4.4.1. The current from the device's current sensor is retrieved by issue the PMBus command $READ\_IOUT_{0x8C}$. Both of values are now compared to each other to verify the device's current sensor. The comparison is made by apply the currents to the compare-to-expected program in figure 36.

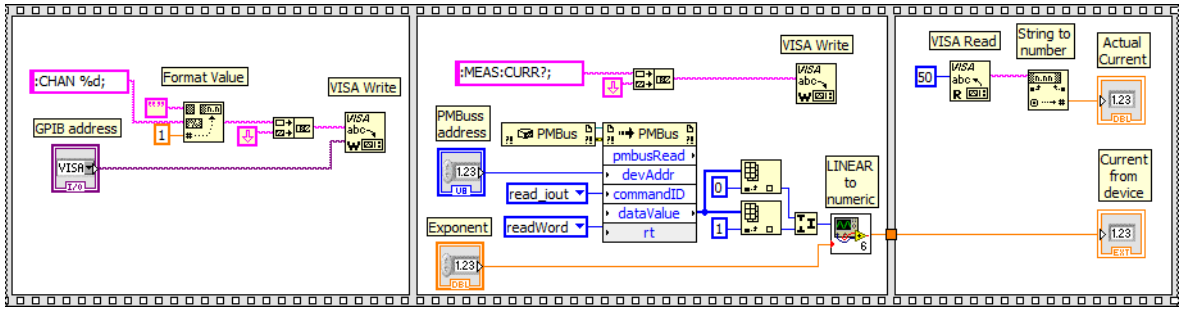


Figure 36: *This program retrieves the test device's current from the PMBus command* $READ\_IOUT_{0x8C}$ *and from the electronic load. These two values are then passed as inputs to the compare-to-expected program that verify that the values are equal within a given margin.*

5.6 Verifying status registers

When a PMBus device is tested for compliance it should be possible to verify the device's response to a fault since fault management is a major part of the PMBus specification. A fault should be possible to identify by reading the device's status registers. In this section eight LabView programs are presented. All of them are almost similar and verifies a bit in a specific status register. A bit is either set high or low and both cases can be verified. Each status register can be verified with the following script commands:

$$CHECK\_STATUS\ \ adr\ \ flag\ \ hilo \tag{17}$$

$$CHECK\_STATUS\_VOUT\ \ adr\ \ flag\ \ hilo \tag{18}$$

$$CHECK\_STATUS\_IOUT\ \ adr\ \ flag\ \ hilo \tag{19}$$

$$CHECK\_STATUS\_INPUT\ \ adr\ \ flag\ \ hilo \tag{20}$$

$$CHECK\_STATUS\_TEMPERATURE\ adr\ flag\ hilo \tag{21}$$

$$CHECK\_STATUS\_CML\ adr\ flag\ hilo \tag{22}$$

$$CHECK\_STATUS\_OTHER\ adr\ flag\ hilo \tag{23}$$

$$CHECK\_STATUS\_MFR\_SPECEFIC\ adr\ flag\ hilo \tag{24}$$

Where *adr* is the PMBus device address, *flag* is a string defining the status flag that should be verified and *hilo* is a string that defines if the flag should be verified as set ”high” or ”low”. If several flags in a status register should be verified, the script has to be executed multiple times with different defined flags. These script commands make it possible to generate a fault and verify that the correct status flag is set or not set. As an example, a over voltage fault can first be generated by issue a $VOUT\_COMMAND_{0x21}$ with a voltage level above the overvoltage threshold level defined by the $VOUT\_OV\_FAULT\_LIMIT_{0x40}$ command and then the overvoltage flag is verified with the CHECK_STATUS_VOUT command.

These script commands are implemented as in figure 37. From a separate script parser the parameters are retrieved. First, the status register is retrieved by transmitting the corresponding PMBus command. In figure 37 the $STATUS\_VOUT_{0x7A}$ command is transmitted to retrieve the voltage status register. The status register is collected as a Boolean array. Depending on which bit is supposed to be verified a case-structure is used to select the corresponding element from the array. The selected array element is the bit from the status register that is supposed to be verified. A comparison block then compares the element with either FALSE or TRUE depending which is expected and defined by the *hilo* flag in the script command. The result from the comparison block is then the decision factor that defines if the test has passed or failed and the result is passed to the log program that writes the result to the log file.

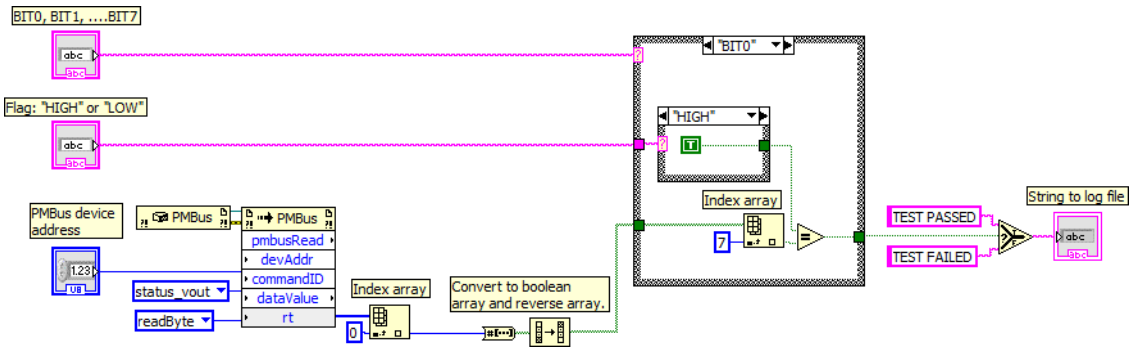


Figure 37: *This program retrieves a status register. The bit that is given as input is compared to the corresponding bit in the status register. The comparison part of this program compare if the register bit set as high or low depending on the input.*

5.7 Summary of compliance test programs

In this chapter a set of LabView program was presented. Each of the programs has a major role in implementing a complete PMBus compliance tester. These programs verify the commanded properties of the device being tested. They are called from a script file with corresponding script commands. Typically, a script first configures a device and then verifies the newly configured property with one of these verification script commands. For example, a device can be programmed to output 10 volts and it can then be verified with the CHECK_DC script command. Other properties that can be verified are the fall and rise time, fall and rise delay, the transition rate and the flags of the status registers.

6 Results

A fully automatically PMBus compliance tester has been implemented. The tester makes it possible for the test engineer to attack a PMBus device with a set of commands and monitor the compliancy of those commands with special measurement programs.
Each PMBus command is implemented as separate LabView sub-programs that transmit the PMBus command to the test device. A number of LabView sub-programs that verify the behavior and the output of the test device are implemented. Both types of programs are called from a main program that also passes parameters to the sub-programs.

Test instruments are used to perform measurements on the test device's output such as an oscilloscope, an electronic load and a power supply. These instruments are remotely controlled through a GPIB interface. The GPIB communication interface implementation is not an open standard so the test instruments cannot be substituted without having to redo the implementation.
The tester is modifiable based on a script language where each sub-program is called by a script file. The script file defines what types of PMBus commands should be transmitted to the test device and what kind of verification tests should be issued. The test engineer can select the script file to run from a GUI.

A PMBus device can also be configured by just writing PMBus commands without any verification test commands in the script file and execute it. This makes the tester also useable for other occasions where a device needs to be configured.

After a script file has been executed by the tester, the test result is logged in a separate log file where the result can be analyzed.

7 Discussion

A fully automatic PMBus compliance tester has been implemented and the tester executes on script files. This means that a test engineer can easily verify a PMBus device without having to understand how the tester works but instead just press the start button in the GUI. Also, the test engineer can change the test parameters by writing a new script file. This is therefore a very useful tool that could be used for different creative test scenarios.
However, the tester can only test the device's responses to PMBus commands and output characteristics with assistance from the test instruments. The test device's compliance with PMBus bit-level transmission issues cannot be tested. This is because of the USB Interface Adapter. This adapter implements the PMBus and makes it easy to send and receive messages on the bus. But with this adapter one has no control of what is transmitted through the bus and the adapter always transmit correct bus messages. There exist test cases where one wants to test the device response to a corrupted bus message. To enable these kind of bit-level tests one has to implement a complete custom microcontroller adapter solution. With a microcontroller based adapter one has full control of the PMBus and different kinds of bit-level tests can be performed. During this thesis there was not time enough to implement a custom adapter.

LabView was used as the main software development tool since the major advantage of available instrument drivers. There existed drivers for two of the three instruments. For instruments that do not feature LabView drivers it is still easy to implement custom made drivers since LabView provides functions for reading and writing to a GPIB bus. For the electronic load used in this report a custom made driver was implemented and is presented in section 4.4.1. If the tester is used in another lab environment where these instruments are not available, one can still find LabView drivers for the new instruments. If an instrument is substituted only a few LabView programs needs to be changed.

A restriction was made early in this report that only subset of all PMBus commands was supposed to be implemented. This means that the tester cannot test a device's PMBus compliancy for hundred percent. Only thirty-nine of all PMBus commands are implemented. But these commands are the ones that are the most used commands at Ericsson Power Solutions.

8 Conclusion

The aim of this report was to implement a PMBus tester. This tester should be able to verify that a PMBus device is compliant with the PMBus standard. Any problems with the compliance should be identified as early as possible in a system project. The tester is automatic and reconfigurable so that test scenarios and test parameters can be changed.

A PMBus tester was implemented using a PC with a LabView program as system controller. Instruments that were used for the tester was remotely configured with a GPIB interface. An oscilloscope has been included for verification measurements on the test device's output. A power supply is used to power up the test device and an electronic load is used as a load. A test is defined and reconfigurable by a script file that lists several script commands. A script command can either be a PMBus command, a verification command or a instrument configuration command. A verification command performs tests and the test result is provided in a separate log file.
With this implemented tester a PMBus device's response to PMBus commands can be verified. However, not all PMBus commands can be issued since that was a restriction for this report. Also, only command responses can be verified and bit-level transmission tests cannot be verified.

For future work of this tester, two important additions can be implemented. First, the PMBus commands that were left out in this report can be added. Then, a PMBus device can be fully configured and tested with all PMBus commands.
The other addition would be to replace the USB Interface Adapter that acts as a PMBus dongle. The replacement could be a microcontroller that implements a PMBus interface between the PC and the test device. The microcontroller should have full control of the PMBus and all data transfers. The meaning for this is to be able to test how a test device response to corrupted data transmissions and bus timing issues.

References

[1] [Online]. Available: http://www.ericsson.com/thecompany

[2] H. Bao and H. Multani, "Energy-based life cycle assessment of industrial products," in *Electronics the Environment, Proceedings of the 2007 IEEE International Symposium on*, may 2007, pp. 123 –127.

[3] F. Taiariol, P. Fea, C. Papuzza, R. Casalino, E. Galbiati, and S. Zappa, "Life cycle assessment of an integrated circuit product," in *Electronics and the Environment, 2001. Proceedings of the 2001 IEEE International Symposium on*, 2001, pp. 128 –133.

[4] H. Li, H. chao Zhang, J. Carrell, and D. Tate, "Integrating energy-saving concept into general product design," in *Electronic Manufacturing Technology Symposium, 2007. IEMT '07. 32nd IEEE/CPMT International*, oct. 2007, pp. 335 –338.

[5] Ericsson. (2010, September) Power supplies go digital. [Online]. Available: http://www.ericsson.com/res/docs/whitepapers/power_%20supplies_go_digital_rev_c.pdf

[6] P. Gensinger, "Microporessors in power control, the time has come."

[7] R. V. White and D. Freeman, "Data communications issues for power system management," in *Applied Power Electronics Conference, APEC 2007 - Twenty Second Annual IEEE*, 25 2007-march 1 2007, pp. 1188 –1199.

[8] G. Baccolini and C. Offeli, "A bus-test program generator for microprocessor systems," in *Instrumentation and Measurement Technology Conference, 1988. IMTC-88. Conference Record., 5th IEEE*, apr 1988, p. 138.

[9] Q. Chengqun, "A design of can-bus test system based on volkswagen passat b5," in *Digital Manufacturing and Automation (ICDMA), 2011 Second International Conference on*, aug. 2011, pp. 16 –19.

[10] M. Popa, V. Groza, and A. Botas, "Lin bus testing software," in *Electrical and Computer Engineering, 2006. CCECE '06. Canadian Conference on*, may 2006, pp. 1287 –1290.

[11] M. Wanliang, G. Deyuan, and M. Chengyu, "The design of pcm tester using fpga," in *ASIC, 1996., 2nd International Conference on*, oct 1996, pp. 202 –205.

[12] M. Nguyen, M. Thalmaier, M. Wedler, D. Stoffel, W. Kunz, and J. Bormann, "A reuse methodology for formal soc protocol compliance verification," in *Specification Design Languages, 2009. FDL 2009. Forum on*, sept. 2009, pp. 1 –6.

[13] M. Soeken, U. Kuhne, M. Freibothe, G. Fe, and R. Drechsler, "Automatic property generation for the formal verification of bus bridges," in *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2011 IEEE 14th International Symposium on*, april 2011, pp. 417 –422.

[14] K. Oumalou, A. Habibi, and S. Tahar, "Design for verification of a pci bus in systemc," in *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*, nov. 2004, pp. 201 – 204.

[15] R. Cram, "Gpib compliance testing in a large test and measurement company," in *AUTOTESTCON '94. IEEE Systems Readiness Technology Conference. 'Cost Effective Support Into the Next Century', Conference Proceedings.*, sep 1994, pp. 669 –671.

[16] J. Floyd and M. Perry, "Real-time on-board bus testing," in *VLSI Test Symposium, 1995. Proceedings., 13th IEEE*, apr-3 may 1995, pp. 140 –145.

[17] V. Raju, K. Balu, G. Collier, and O. Duran, "Automated testing of satellite data configuration modules using labview," in *Space Technology (ICST), 2011 2nd International Conference on*, sept. 2011, pp. 1 –4.

[18] J. Zhao, "Automating measurement bench of dc-dc converter in labview," in *Industrial Electronics and Applications (ICIEA), 2011 6th IEEE Conference on*, june 2011, pp. 897 –900.

[19] M. Neskovic and, J. SpasicÌĄ and, V. CÌĬŇ andelebicÌĄ and, and I. Salom, "An automated hardware testing using pxi hardware and labview software," in *Circuits and Systems for Communications (ECCSC), 2010 5th European Conference on*, nov. 2010, pp. 232 –235.

[20] P. Turley and M. Wright, "Developing engine test software in labview," in *AUTOTESTCON, 97. 1997 IEEE Autotestcon Proceedings*, sep 1997, pp. 575 –579.

[21] Pmbus specification part i rev. 1.1 final. [Online]. Available: http://pmbus.org/docs/PMBus_Specification_Part_I_Rev_1-1_20070205.pdf

[22] Pmbus specification part ii rev. 1.1 final. [Online]. Available: http://pmbus.org/docs/PMBus_Specification_Part_II_Rev_1-1_20070205.pdf

[23] Ericsson. (2010, September) Digital dc/dc pmbus connect. [Online]. Available: http://fplreflib.findlay.co.uk/articles/25099%5Cdigital-dcdc-pmbus-connect.pdf

[24] P. Thanigai., "Pmbus implementation using the msp430 usci," January 2008.

[25] Ericsson. Microcontroller for programming for 3e digital products. [Online]. Available: http://archive.ericsson.net/service/internet/picov/get?DocNo=34/28701-FGC1011486&Lang=EN&HighestFree=Y

[26] R. White and D. Durant, "Understanding and using pmbus trade; data formats," in *Applied Power Electronics Conference and Exposition, 2006. APEC '06. Twenty-First Annual IEEE*, march 2006, p. 7 pp.

[27] Usb interface adapter manual. [Online]. Available: http://cds.linear.com/docs/Demo%20Board%20Manual/DC1613Af.pdf

[28] Ericsson. (2011, December) Bmr 450 digital pol regulators. [Online]. Available: http://archive.ericsson.net/service/internet/picov/get?DocNo=28701-EN/LZT146395&Lang=EN&HighestFree=Y

[29] ——. (2011, December) Bmr 450 digital pol regulators. [Online]. Available: http://archive.ericsson.net/service/internet/picov/get?DocNo=28701-EN/LZT146400&Lang=EN&HighestFree=Y

[30] Xantrex. (2002, July) Operating manual for internal gpib interface for xkw

1000 watt series programmable dc power supply. [Online]. Available: http://fplreflib.findlay.co.uk/articles/25099%5Cdigital-dcdc-pmbus-connect.pdf

[31] Lecroy. (2002, January) Wavepro dso operator manual. [Online]. Available: http://cdn.lecroy.com/files/manuals/wp_om_revc.pdf

[32] G. Coelho da Silva Stanisce Correa, A. da Cunha, L. Vieira Dias, and O. Saotome, "A comparison between automated generated code tools using model based development," in *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*, oct. 2011, pp. 7E4–1 –7E4–9.

[33] T. Fountain, "Software advances in measurement and instrumentation," in *Software Instrumentation - Software Components, IEE Colloquium on*, feb 1993, pp. 4/1 –445.

[34] [Online]. Available: http://www.mathworks.se/products/instrument/hardware/gpib.html

[35] L. Hong and J. Cai, "The application guide of mixed programming between matlab and other programming languages," in *Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on*, vol. 3, feb. 2010, pp. 185 –189.

[36] "Ieee standard serial interface for programmable instrumentation," *IEEE Std 1174-2000*, pp. i –30, 2001.

[37] G. Drenkow, "Future test system architectures," in *AUTOTESTCON 2004. Proceedings*, sept. 2004, pp. 441 – 447.

[38] N. Instruments. Labview visa tutorial. [Online]. Available: www.ni.com/support/visa/vintro.pdf

[39] Chroma. Programmable dc electronic load 6310 series operation and programming manual. [Online]. Available: http://128.238.9.201/~kurt/manuals/manuals/Other/CHROMA%206310%20Series,%206312,%206314%20Mainframes%206310X,%20Operation%20&%20Programming.pdf

[40] N. Instruments. Labview driver for labview. [Online]. Available: http://sine.ni.com/apps/utf8/niid_web_display.download_page?p_id_guid=E3B19B3E9511659CE034080020E74861

[41] ——. Power supply driver for labview. [Online]. Available: http://sine.ni.com/apps/utf8/niid_web_display.download_page?p_id_guid=E3B19B3E93DF659CE034080020E74861

[42] Xantrex. (2003, Mars) Xkw 1kw user guide 10op 01xn. [Online]. Available: http://www.powerten.com/products/XKW/downloads/XKW_1kW_User_Guide_10OP-01XN.pdf