Anssi Niemi

Control and data collection software for automatic integrated circuit evaluation

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of
Science in Technology.
Espoo 15.2.2012

Thesis supervisor:

Prof. Kari Koskinen

Thesis instructor:

M.Sc. (Tech.) Kimmo Hauskaviita

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Anssi Niemi

Title: Control and data collection software for automatic integrated circuit evaluation

Date: 15.2.2012 Language: English Number of pages: 7+61

Department of Automation and Systems Technology

Professorship: Information and computer systems in automation Code: AS-116

Supervisor: Prof. Kari Koskinen

Instructor: M.Sc. (Tech.) Kimmo Hauskaviita

Automatic evaluation of integrated circuits can provide significant benefits and savings for a company compared to doing the evaluation manually. This paper describes a LabVIEW application called ATAC that was developed to automate evaluation of ASIC circuits. The same software can also be used to test integrated circuits in general in different hardware environments. A hardware setup for automatic ASIC evaluation is presented and it is used as basis for ATAC design. Presented software application ATAC can be used to control all hardware components in the evaluation setup but it can also be used without the hardware components. Hardware abstraction in ATAC makes it possible to use the same software solution in different environments with little modifications to the software code. The process of developing ATAC is presented as well as final application. Screenshots of GUI are presented as well as the underlying code using state chart presentation. The software was reviewed and assessed by end users who performed several ASIC evaluation tests using ATAC during the software development. An example case of an ASIC evaluation test is presented and the measurement results gathered with ATAC are discussed.

Keywords: ASIC, IC, ATAC, evaluation, characterization, automatic, automated, software, application, LabVIEW, object-oriented programming

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Anssi Niemi

Työn nimi: Ohjaus- ja tiedonkeruuohjelma automaattiseen integroitujen piirien evaluointiin

Päivämäärä: 15.2.2012 Kieli: Englanti Sivumäärä: 7+61

Automaatio- ja systeemitekniikan koulutusohjelma

Professuuri: Automaation tietotekniikka Koodi: AS-116

Valvoja: Prof. Kari Koskinen

Ohjaaja: DI Kimmo Hauskaviita

Integroitujen piirien automaattinen evaluointi voi tuottaa huomattavia etuja ja säästöjä yhtiölle verrattuna evaluoinnin manuaaliseen suorittamiseen. Tässä opinnäytetyössä esitellään LabVIEW-sovellus nimeltään ATAC, joka kehitettiin automatisoimaan ASIC-piirien evaluointi. Samaa sovellusta voi käyttää myös muiden integroitujen piirien testaukseen erilaisissa laitteistoympäristöissä. Laitteisto ASIC-piirien automaattista evaluointia varten esitellään ja tämän laitteiston käyttöön ATAC lähtökohtaisesti suunniteltiin. Esitettyä ohjelmaa voi käyttää laitteiston kaikkien osien kontrollointiin, mutta ohjelmaa voi käyttää myös ilman laitteistokomponentteja. Laitteiston abstrahointi ATAC:ssa mahdollistaa ATAC:n käytön eri ympäristöissä ilman suuria muutoksia ohjelmakoodiin. ATAC:n kehitysprosessi sekä lopullinen ohjelma esitellään tässä opinnäytetyössä. Graafinen käyttöliittymä esitellään kuvakaappausten avulla ja koodi kuvaillaan käyttäen tilakaavioita. Ohjelman käytettävyyttä ja ohjelmankehitysprojektin onnistuneisuutta arvioidaan loppukäyttäjien kokemusten sekä todellisten ASIC-piirien evaluointimittausten tulosten perusteella. Esimerkki ASIC-piirin evaluointitestistä esitetään ja ATAC:n keräämiä mittaustuloksia arvioidaan.

Avainsanat: ASIC, IC, ATAC, evaluointi, karakterisointi, automaattinen, automatisoitu, ohjelma, sovellus, LabVIEW, olio-ohjelmointi

Preface

I would like to thank D.Sc. (Tech.) Saska Lindfors and Prof. Kari Koskinen for guidance through writing this thesis. I would also like to thank M.Sc. (Tech.) Kimmo Hauskaviita for providing expertise and assistance with the evaluation system hardware.

Helsinki, 15.2.2012

Anssi Niemi

Contents

Symbols and Abbreviations

Symbols

I	Current
P	Power
R	Resistance
U	Voltage

Abbreviations

ACE	Automated circuit evaluation
ASIC	Application specific integrated circuit
ASL	Application separation layer
ATAC	Automated test and characterization. Name of the software developed in this thesis
ATE	Automated test equipment
API	Application programming interface
CPU	Central processing unit
DLL	Dynamic-link library
DSSP	Device-specific software plug-in
DUT	Device under test
EMI	Electromagnetic interference
EVM	Evaluation module
FPGA	Field-programmable gate array
GPIB	General Purpose Interface Bus
GPIO	General Purpose Input/Output
GUI	Graphical user interface
I^2C	Two-wire serial bus invented by Philips
IC	Integrated circuit
IDE	Integrated development environment
MCU	Microcontroller
NRE	Non-recurring engineering
OOP	Object-oriented programming
PC	Personal computer
PCB	Printed circuit board
SPI	Serial Peripheral Interface Bus
TDMS	File format created by National Instruments
UART	Universal Asynchronous Receiver/Transmitter
UML	Unified modeling language
VI	Virtual instrument. Basic block of LabVIEW programming.
VISA	Virtual instrument software architecture

1 Introduction

The use of integrated circuits (IC) in electronic devices has increased rapidly since the invention of IC technology at late 1950's. Today a modern silicon chip can hold over hundred million transistors and the complexity of a chip has become very high [5]. At the same time the cost of a single chip has decreased systematically [3]. The large number of components and the complexity of a modern IC set challenges for design and evaluation when new circuits are being developed.

Today many different IC devices are produced with different purposes and complexities. The software presented in this thesis was developed to be used to evaluate application specific integrated circuits (ASIC) but can also be used to evaluate and debug various other type of circuits. Evaluation also known as characterization is a process where the operation of a device is tested and compared against requirements documentation and simulations. Evaluation of ASICs involves many different kinds of tests and measurements that are performed before the device can be released for market. This thesis focuses on the so called bench evaluation tests that are performed on a lab setup that closely resembles the target application of the ASIC. In bench evaluation external components are connected to the ASIC pins to match the target application. The goal of this thesis was to develop a software application to automate the bench evaluation testing. The design of the software is presented as well as implementation and test results. The end users of the developed software had existing software application that was able to execute measurement scripts somewhat automatically and to control instruments remotely. The goal of this thesis was to create a better software that would be easier to use and would be able to control whole evaluation system instead of just instruments.

The thesis is constructed such that after the introduction chapter the reader is given a general view of ASIC circuits and circuit evaluation. This provides basis for understanding the requirements for the evaluation environment and for the software. Chapter 3 describes the evaluation system environment that includes hardware components such as PCBs (Printed circuit board), instruments and software framework. Understanding the whole evaluation system is essential for creating reasonable requirements for the software. The chapter 3 is divided into several sub chapters describing all the different parts of the system. Good understanding of different components of the system hardware is needed because the software will communicate with all the components and make them work together. In chapter 4 software development techniques and LabVIEW are introduced to provide the reader a basis for understanding the reasons behind the decisions made during software design. The software that was designed and implemented in this thesis is presented in chapter 5. The software, named as ATAC (Automated Test And Characterization), is first presented in a general level describing the features and the development process. After that the reader is given a more detailed description over different features of the software and descriptions how the software can be used. Understanding how the ATAC is used provides the reader a better view over the software and the descriptions regarding structure and architecture of the software are easier to understand in later chapters. After describing how the ATAC is used the structure of the software is illustrated and explained. At the end of chapter 5 the methods used for testing the software are presented. Successfulness of the software project and the usability and flexibility of ATAC is assessed in chapter 6. The quality of the software is assessed by meeting the specification, correctness of produced measurement data and by the general usability and flexibility. An example of a real evaluation test performed to an ASIC using the presented evaluation system and ATAC is also presented in chapter 6. The reader is given a general view of how the evaluation of an ASIC is done with ATAC and what results and data the ATAC produces. Possible future improvements are presented in chapter 7.

2 Integrated circuit characterization

In this chapter application specific integrated circuits are described in general level and their properties are compared to other IC technologies. The evaluation of ASIC devices is also described to provide basic understanding of the measurements and tests needed to characterize a new ASIC after it has been manufactured. The benefits of automatic evaluation are presented as well as reasons for developing the control and data collection software that is presented in this thesis.

2.1 Application specific integrated circuits

Application specific integrated circuits are widely used devices in high volume products. Because of their good performance combined with small size and small energy consumption they are often better choice than general purpose devices such as FPGAs (Field-Programmable Gate Array) [2], [9] and MCUs (Microcontroller) [22]. Because ASIC circuits are custom made for specific applications it is vital that the performance of the ASIC is tested before it is sent to a customer.

The design of an ASIC is often done at transistor level but also at higher hardware layers. The difference to general purpose microcontrollers and FPGAs is that general purpose devices have generic hardware and they can be programmed to suite different applications. Microcontrollers have fixed hardware but the MCUs' software can be programmed to meet different applications. FPGAs have a generic hardware that can be programmed for different applications at hardware level. Thus FPGAs can be seen as compromise between hardware solution (ASIC) and software solution (MCU). Because user can program the hardware of an FPGA the performance is usually better than a pure software solution like using MCU. [22]

ASIC circuits have hardware that is designed for a specific application and the hardware cannot be altered by programming. ASIC devices usually have some parameters that can be adjusted by programming but the main functionality of the device is fixed at hardware level. This means that ASICs usually have very good performance, low power consumption and small size compared to general purpose devices [2]. On the other hand the initial cost of ASIC is greater than with off-the-shelf devices. The high initial cost consists of designing and manufacturing of the ASIC. Besides being expensive the design and manufacturing of ASIC device is also time-consuming. The development of an ASIC may be several years long project whereas off-the-shelf devices can be programmed within much less time. However if the volume of the application where ASICs are used is high, then the high initial cost may not be a problem. This is because once the ASIC device has been designed and manufactured the cost of a single device is low. Choosing between different device solutions (ASIC, FPGA, MCU) is an optimization problem that depends on the initial NRE (Nonrecurring engineering) cost, continuous cost and the volume of the devices [22]. Figure 1 illustrates how ASIC cost is competitive with different technologies when the volume of devices is high.

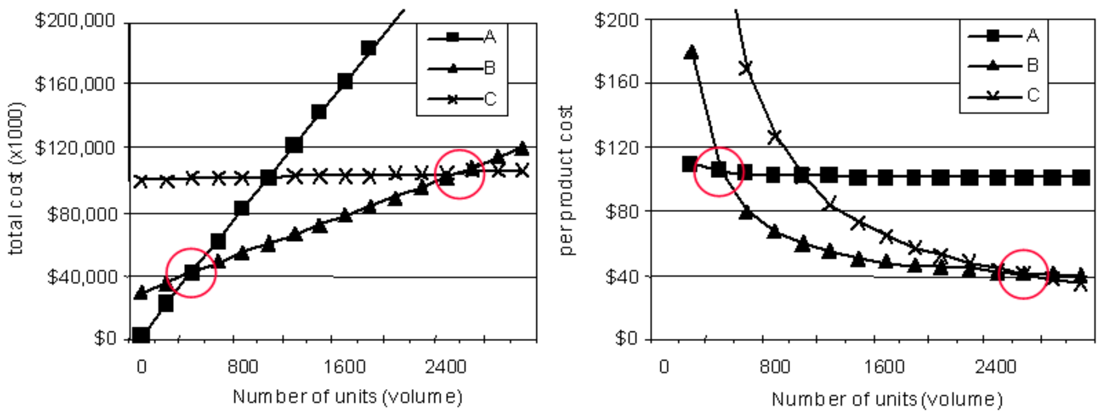


Figure 1. Cost of different IC technologies in general. A = MCU, B = FPGA, C = ASIC. [25].

2.2 Trimming and evaluation

Because an application specific integrated circuit is designed at hardware layer it is vital to perform evaluation once the device has been manufactured. In evaluation the device performance and functionality is tested and compared against simulations and specification. Usual evaluation tests that are performed include functional tests, open-loop tests and closed-loop tests. Functional tests are used to verify the general functionality of the device. Different values are written to registers of DUT (Device Under Test) and the state of the device should change accordingly.

Open-loop tests are tests where no external components are connected to the pins of DUT. These tests include sweeping and adjusting internal parameters and the effects can be measured from specific output pins. Open-loop tests can be performed using highly automated systems called ATE (Automated Test Equipment). With ATE thousands of ASIC devices can be tested on silicon wafers right after the manufacturing. The problem with these test systems is that they are expensive to use and they cannot evaluate all the parameters of the device. For example measurements that need external components cannot be tested on ATE systems.

Closed-loop tests are performed by connecting external components to ASIC pins to create a setup that closely matches the intended target application. In many cases ASIC has an internal switching regulator or some other component that needs external components for performing so to test these internal components different external components are required. These external components allow flexibility for the applications that use the ASIC. External components such as resistors, capacitors, and coils can be used to adjust specific characteristics of the ASIC. For example output voltages of ASIC output pins could be designed to be externally adjustable by adding external components to specific pins of the device. In some cases it is not even possible to integrate all components inside the ASIC. For example large capacitors and coils for switching regulators are such components. Closed-loop tests are much slower to perform than open-loop tests that are done with ATE. That is why not all devices can be tested in closed-loop environment. Closed-loop tests are usually done only to few devices where as open-loop tests are performed to all new ASICs.

Before characterization of an ASIC is started the internal reference quantities such as reference voltage, reference current and reference frequency must be trimmed. Trimming means adjusting the internal reference parameters so that the reference quantities correspond to predefined values. Trimming is necessary because process variation in manufacturing results in imprecise reference quantities. All voltages and currents in an ASIC are derived from the few reference quantities, so if the references have offset all the other quantities have offset also [26]. ASICs are usually designed to have trimming registers which can be programmed through communication interface to fine adjust reference signals. The trim registers as well as other registers of an ASIC are located inside the digital core of the device [24]. In addition to the digital core there usually exists number of analog circuits integrated around the digital core. The digital core controls the state flow of the device and provides communication interface. In addition to programming trim values to device memory communication interface is used to set output channels of the device to a desired level and to control the state of the device.

2.3 Automated evaluation

Time-to-market is a critical concern in the business of selling ASICs. If the design, manufacturing or evaluation is delayed for some reason the impact on profit can be significant. If these phases can be speeded up it will provide great edge in the market for the company. Time needed for evaluation depends on the complexity of the device and the length of specification. It can take several weeks or months to measure all the parameters and cases specified in the device specification. If all the tests are performed using for example three different supply voltages and three different temperatures the amount of data produced is very large and acquiring and processing that data takes time. Even using only one temperature and one supply voltage level a single test can require tens to hundreds of measurements. Usually closed-loop bench evaluation consists of tens of tests which each are performed for few different devices using different temperatures and supply voltages. If evaluation is done by a lab engineer who manually connects instruments (multimeters, source meters, power supplies, etc.) to DUT for different tests and manually writes commands to device registers it is easy to see that evaluation will take lots of time. Also it might not be motivating work for engineer to perform several tests that are almost identical to each other. Automating data collection will provide savings for a company in time and money and it will increase the work motivation of test engineers.

Many tests can be performed automatically for large number of devices in small amount of time using ATE right after the devices have been manufactured. Although these test systems can test functionality of large number of devices quickly, they are expensive to use and they can only perform open-loop measurements. Closed-loop measurements must be performed in different environment using PCBs (Printed Circuit Board) that provide external components for the DUT. Different commercial [29], [17] and non-commercial systems and software exist to automate the characterization at these closed-loop tests. The automation level may vary from very low automation level where user has to control the testing all the time to high automation level software that can perform tests quite independently. The benefit from creating new software for automated evaluation is that the software can be designed to meet specific needs of target users. Commercial software solutions tend to be generic in nature to be compatible with many different hardware environments and use cases. This has a drawback that the user needs to spend more time configuring the software and creating the tests. The ATAC software developed and described in this thesis was designed in close co-operation with target users to provide easy interface for creating and executing tests for ASIC circuits and yet maintaining certain level of generic compatibility.

3 System environment

The environment where ATAC was designed to be used is presented in this chapter. The hardware components such as instruments, PCBs (Printed circuit board) and adapters are essential parts of any IC evaluation system. Understanding the environment is basis for understanding the functionality and use of ATAC software. The presented environment is only one possible setup that can be used to automate ASIC evaluation. The ATAC does have some features specifically designed for the presented hardware setup but the software can be used in other hardware environments also.

3.1 Overview

The hardware of an evaluation environment sets boundary conditions and requirements for the control software. Block diagram of the whole evaluation system is illustrated in Figure 2. The software is executed on PC (Personal computer) that connects to different components of the system. The instruments illustrated in the figure can be power sources, multimeters, source meters, etc. that are connected to bench EVM (Evaluation module) PCB through ACE (Automated circuit evaluation) board. The ACE board is used to route instruments to different pins on bench EVM. An ASIC to be evaluated is mounted on bench EVM board and thus the instruments that are routed to bench EVM pins are also connected to the ASIC pins. Different components of the system have different communication interfaces and thus adapters are needed to translate communication between PC and other components of the system. In the following chapters individual components of the system are described in more detail.

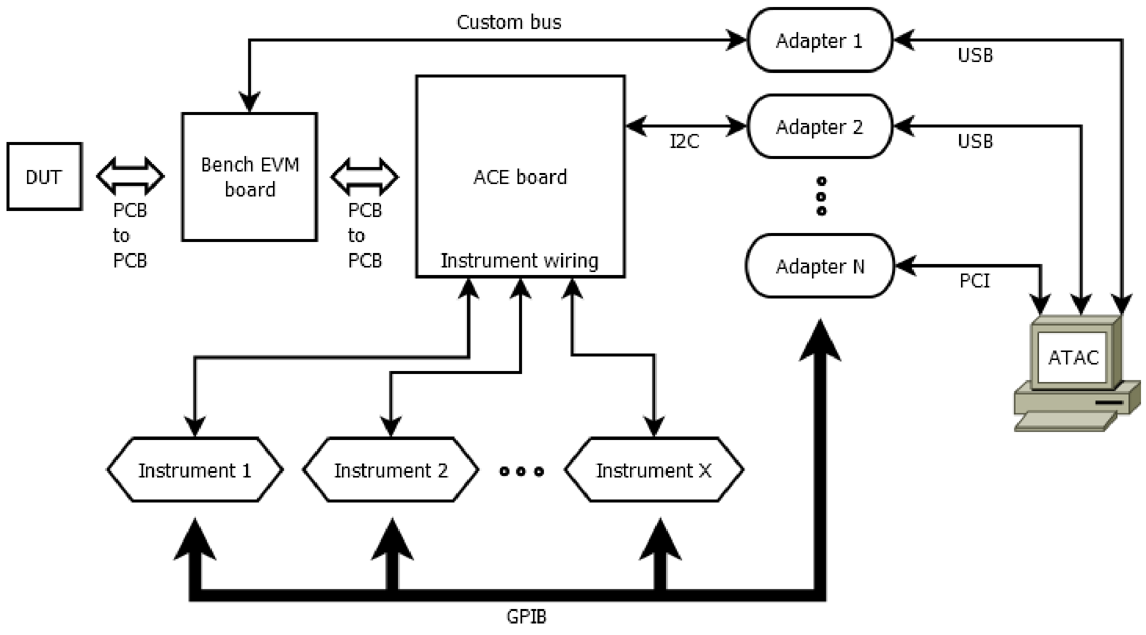


Figure 2. Overview of the automated evaluation system.

3.2 ACE board

3.2.1 Overview

The ACE board is used to automatically route instruments to different pins of bench EVM and DUT and to provide adjustable external resistance load. ACE board can also be used to generate line and load transients to evaluate transient characteristics of the ASIC under test. ACE board PCB layout is presented in Figure 3. The ACE board is controlled through I^{2}C bus.

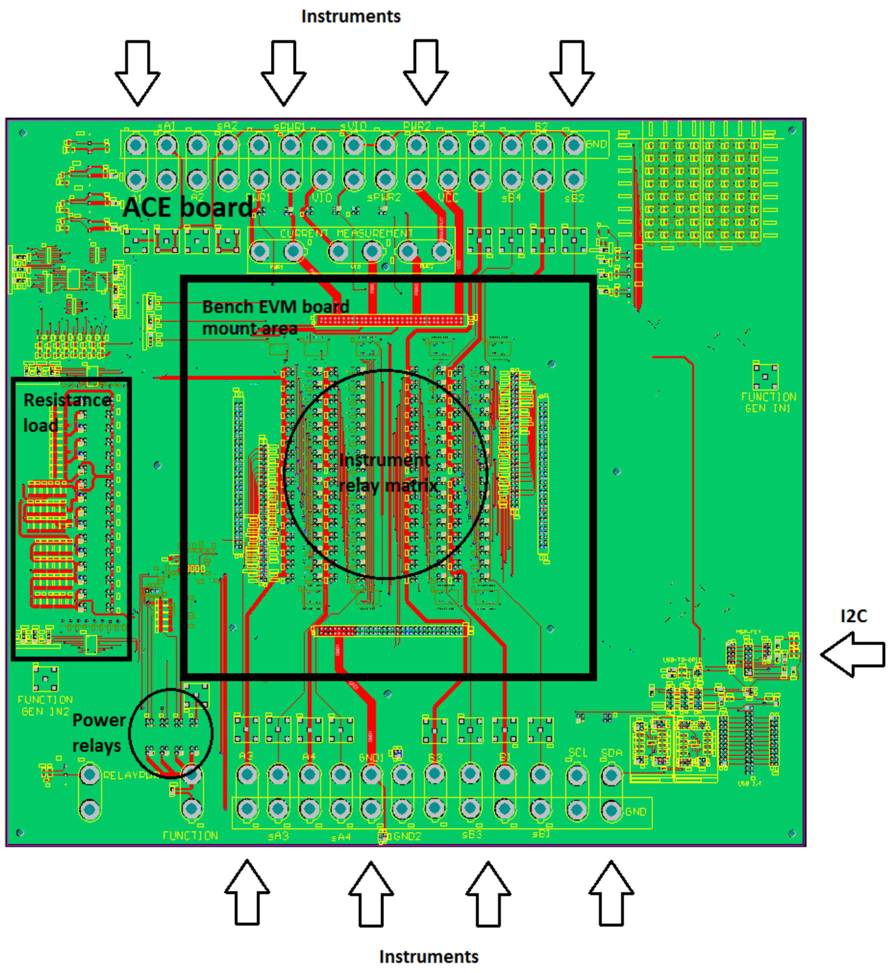


Figure 3. ACE board PCB layout.

3.2.2 Instrument relay matrix

The main feature of the ACE board is to provide a remotely controllable relay matrix for connecting instruments to different pins of bench EVM board. The relay matrix works as a multiplexer between instruments and bench EVM board so that specific instrument can be connected to any pin on the bench EVM without manually routing wires between instrument and target circuit pins. The relay matrix is controlled with I^{2}C communication bus. The relay matrix has 64 relays in total and can connect up to eight instruments to fourteen different bench EVM pins. Every instrument has a signal route and a sense route to the bench EVM board. Sense route can be used by the instrument to measure actual voltage (or other quantity) on the DUT pin because sense line does not have current flowing. Without current flow the voltage drop between two ends of the sense line is zero. Thus the instrument sees the actual voltage on the ASIC pin. In signal lines current flows between instrument and the DUT and causes voltage drop over signal line. Using only signal line the instrument would not see the actual voltage on DUT pin because of the voltage drop over signal line.

3.2.3 Power relays

Power relays provide a way to control special functions of ACE board such as line and load transients and resistance load. With power relays user can choose what function is selected and routed to bench EVM board. Power relays are controlled by I^2C bus in the same manner as instrument relay matrix.

3.2.4 Resistance load

ACE board provides adjustable resistance load that can be routed to bench EVM board through power relays. The resistance value can be adjusted by remote I^2C commands. Supported resistance values are 2^n Ω, where n = 0, 1, 2, … , 15. The resistance load is designed for adjusting small (< 100 mA) currents. Using the resistance load to sink large (0,1 – 1 A) currents is possible as long as keeping the limited resolution of the resistance load in mind as well as the power rating of the resistors.

An example of the effect of the limited resolution: Let’s say the external resistance load is connected to ASIC pin that is at 5V potential. Changing resistance value from 32768 Ω (2^{15} Ω) to 16384 Ω (2^{14} Ω) results in current changing from $I_1 = \frac{U}{R_1} = \frac{5V}{32768\,\Omega} = 152{,}6\,uA$ to $I_2 = \frac{U}{R_2} = \frac{5V}{16384\,\Omega} = 305{,}2\,uA$. The current changes only 152,6 uA allowing accurate adjustment of small currents using the resistance load. Changing resistance value for example from 16 Ω (2^4 Ω) to 8 Ω (2^3 Ω) results in current changing from $I_1 = \frac{U}{R_1} = \frac{5V}{16\Omega} = 0{,}3125\,A$ to $I_2 = \frac{U}{R_2} = \frac{5V}{8\Omega} = 0{,}625\,A$. The current changes 312,5 mA which is a large step in current. Accurate adjustment of a large current is thus impossible with the ACE board resistance load.

3.3 Bench EVM board

3.3.1 Overview

Bench EVM is a PCB that provides all the necessary components for the ASIC under test to operate as it would in target application. The ASIC is mounted on top of the bench EVM and the pins of the ASIC are routed to bench EVM connectors that lead to ACE board. The bench EVM board also provides functionality to drive the DUT into different modes so that evaluation of all necessary features can be performed.

The bench EVM board needs to be specifically designed and manufactured for every new ASIC. However the interface between the bench EVM and ACE board remains same throughout different projects. This enables the use of ACE board in different projects and only the bench EVM board needs to be redesigned for each project. Communication to bench EVM board can be routed through the ACE board if the bench EVM and DUT use I^2C communication. Otherwise a separate adapter between bench EVM and PC needs to be used to enable communication with the bench EVM and the DUT.

3.4 Adapters

3.4.1 Overview

Adapters work as a link between PC and other hardware. An adapter provides API (application programming interface) for the PC software and hides low level hardware specific communication. ACE board uses I^2C communication protocol so adapter is needed to communicate between PC and ACE board. Bench EVM board is different for each ASIC so the communication bus to bench EVM and DUT can differ between projects. An adapter is needed to communicate with bench EVM as well as DUT mounted on bench EVM. The DUT can use custom communication protocols so adapter needs to be flexible enough to support many different communication means. Often many different adapters are used at the same time to communicate with different parts of hardware. Depending on what communication cards the PC has an adapter between PC and instrument bus may also be needed.

3.4.2 Adapter interfaces

Adapter handles the low level hardware communication towards target device and provides a high level API for main software. All the APIs for different adapters that were used with the ATAC software were available as dynamic-link library (DLL) or as LabVIEW VI’s. The LabVIEW VI’s hid the DLL calls and provided easy API to use in LabVIEW environment. The DLL works in between the main software and adapter firmware providing methods to main software and converting the data to the format adapter firmware supports.

3.5 Instruments

3.5.1 Overview

Essential part of the evaluation environment is instruments that provide voltage and current to DUT and other hardware. The instruments are also used to read specific quantities such as voltage, current and frequency from the pins of DUT. Before automated evaluation system instruments were used in manual manner meaning instruments were needed to manually connect to specific pins and reading of measurements was done by visually reading the instrument display. In the automated evaluation system presented in this paper the instruments are connected to different pins of DUT automatically and the measurements are also read automatically by the ATAC software. Different instrument types the presented evaluation system has and the ATAC supports are

- Multimeter

- Source meter
- Power supply
- Battery simulator.

Other instruments that were used with manual evaluation but not implemented in the automated system were

- Oscilloscope
- Function generator.

These instruments would require more initial configuration than multimeters, source meters, power supplies and battery simulators making them more difficult to handle in the software. Support for oscilloscopes and function generators was considered to be unnecessary in ATAC because great number of evaluation tests can be done without them. However ATAC was designed so that support for new instruments could easily be added in the future.

3.5.2 Instrument interfaces

Instruments were connected to IEEE-488 bus which is more commonly known as GPIB (General Purpose Interface Bus). GPIB is a common industrial bus used to communicate with instruments remotely. The bus supports different topologies such as star, chain and hybrids of these two. PC communicates to GPIB bus through GPIB card installed in PC’s PCI bus or through adapter that connects to PC’s USB port. The low level commands sent to GPIB bus were hidden by the LabVIEW drivers that provided high level access to the instrument. LabVIEW drivers for instruments are a common way of communication between PC and instruments because they provide fast and easy access to instruments without having to learn low level command syntaxes. Large number of LabVIEW drivers for different instruments are available at National Instruments website [16] for free use which makes the remote controlling of instruments very easy.

3.6 Device under test

3.6.1 Overview

In the evaluation system the device under test is an ASIC circuit that needs to be evaluated. The ASIC is mounted on the bench EVM board that is designed specifically for that ASIC device. The bench EVM board provides all the external components needed for the ASIC to operate and to be able to perform all the evaluation measurements.

The ASIC can be soldered directly to bench EVM board or it can be mounted on a specially made socket that is soldered or screwed to the bench EVM board. Coupon boards are also used to mount ASIC to bench EVM board. Coupon board is like a socket for the ASIC but it provides also essential capacitors and inductors that need to be as close to the ASIC as possible. The coupon board has connectors which connect to corresponding connectors on bench EVM board.

3.6.2 Device interfaces

Making ASIC circuits as small as possible has benefits such as saving target PCB space and making the ASIC applicable to small hand held devices. Silicon area of an ASIC is used sparingly and the number of pins is kept low. Single pin usually has many different operating modes that can be chosen by writing to device registers. Implementation of communication protocol uses silicon area and affects the total size of the ASIC. Communication interface also needs some number of pins which again increases the final size of the device. [8] This is why special communication methods that use as few as one pin and as little space on silicon as possible are tempting choices in ASICs. However these communication methods may not be standardized and they can be quite difficult to operate. Adapter that works as a link between PC and the ASIC needs to know how to handle such communication protocols. The communication interface that is used to control the ASIC provides means to write and read the registers of the ASIC and thus control the operation of the ASIC. Although no communication interface may not be needed when using an ASIC in a target application, it is necessary to have an interface for testing and trimming the ASIC device after it has been manufactured.

3.7 PC requirements

The ATAC software is a LabVIEW application and thus needs LabVIEW run-time engine (2011 SP1) to be installed on the PC in order to be executable. The LabVIEW run-time engine is free for everyone and can be downloaded at National Instruments website. The LabVIEW run-time engine requires Windows 7/Vista/XP/Server operating system. [12]

A GPIB interface card or a GPIB adapter is needed for communication between PC and instruments. Other communication buses can be used instead of GPIB depending on what buses instruments support. ATAC software has a hardware abstraction layer which enables the use of any communication bus between PC and other components of the system including instruments and adapters.

To communicate with different PCBs in the system adapter or multiple adapters is needed. All the adapters and instruments require drivers to be installed on PC before use. In addition all instruments and adapters must have an implemented ATAC plug-in in order for the device to be accessible from ATAC. A plug-in separates hardware specific functionality of instruments and adapters from the main software and makes it possible to easily add support for different instruments and adapters in ATAC. More detailed description of the hardware abstraction layer (HAL) of ATAC is presented in chapter 5.

4 Software development techniques

This chapter describes software development techniques such as object oriented programming and software life cycle management. The ATAC was developed using object oriented programming and spiral type of life cycle management so understanding the basics of these concepts gives the reader a deeper view on the ATAC structure and development process. LabVIEW programming is also described in general level to give the reader an understanding how LabVIEW programming is practiced and how it differs from programming with other languages.

4.1 Object oriented programming

Object oriented programming (OOP) is a programming style that has become widely used when programming large software applications. OOP provides a way to efficiently manage software and maintain good scalability and modularity. Many programmers can work on the same project without conflicting each other's work because the different parts of software communicate through certain interface and the inner functionality of one part of software can be handled as a black box by another part. [6]

In OOP classes are used to encapsulate features that belong to certain area. For example there could be a class called “Car” and that class would hold all information and functionality that is related to different cars. For example the class “Car” could hold information such as “car name”, “model”, “number of tires”, etc. This information is encapsulated inside of the class and it is not visible from outside of the class. To access the information from outside the class the class provides methods for that. Methods are an interface to a class. For example the class “Car” could have methods such as “get car name”, “set number of tires”, “accelerate”, etc.

Class is a static structure that defines features and operations for a certain entity. Object is an instance of a class. When a program wants to create a new object it makes a call that creates a new instance of a class. One class can be used to create multiple objects that all have different parameters. For example the class “Car” could be used to create objects that represent different cars that have individual names and model types. They would all have the methods and private fields “name”, “model”, etc. that were defined by the class “Car”.

The use of classes makes it possible to develop different parts of large software separately and make the different parts work together relatively easily. Because the communication between different objects is performed through methods, one object only needs to know what the methods of the object it is using are. It does not have to know the inner functionality of the other object and it cannot accidentally change the inner data of the other object.

OOP includes concepts called inheritance and polymorphism. Inheritance means that a class can inherit another class and become a child class for the inherited class. A parent class and its private data and methods that are inherited usually contain common information that applies to all child classes. For example a class “Car” could be used to define information and operations that apply to all cars and sub classes. For example classes such as “Toyota”, “BMW”, and “Skoda” could be used to define information specific to a certain car manufacturer. If these sub classes are defined to inherit the class “Car”, then the child classes also have the data and operations of the parent “Car” class.

Use of inheritance makes the code efficient because same information does not have to be defined to different classes separately if they all inherit the information from a single parent class. Inheritance also makes the maintainability of the software more manageable. If one would like to change a certain feature of the software one could make the change to a parent class and the same change would then apply to all child classes, instead of modifying the code in several different places.

Polymorphism means that the data type of an object is not only defined by the class of the object but also by the parent class of that object. For example an object called “BMW 7 Hatchback” is a type of “BMW” but also a type of “Car”. Polymorphism enables a way to create generic interfaces that accept multiple different data types. [6], [23]

Object oriented programming was used to design and program ATAC software. This provided variety of benefits which will be discussed in chapter 5.

4.2 LabVIEW programming

4.2.1 LabVIEW in general

LabVIEW is a graphical programming language and IDE (Integrated Development Environment) developed by National Instruments [16]. It is widely used in electrical and test engineering because it provides a fast and easy way to interact with hardware [27]. A large database for support and drivers for different hardware exists at National Instruments website [16] and user rarely needs to program low level functions for interacting with hardware. LabVIEW also provides ready to use elements such as windows, textboxes, buttons, etc. for creating graphical interfaces which makes it possible to build applications with very little effort [27].

LabVIEW programs are called virtual instruments (VI) because they usually imitate physical instruments. A VI resembles a function or a method but with a difference that a VI can be run individually or it can be called from other VI’s. LabVIEW divides the programming of a VI into three sections: Front panel, block diagram and connector pane. Front panel is a graphical user interface of the VI containing access to controls and indicators of the VI for the user. Every VI has a front panel even though only the top level VI’s front panel is visible to the user. Block diagram of a VI is the actual code that executes when the VI is run. When creating a block diagram code LabVIEW compiles the code in real-time as user writes it. This makes the development of block diagram code easy because user sees instantly if the code contains errors that do not compile. LabVIEW provides also debugging tools that help to check the correctness of the code effortlessly without writing separate test programs. The debugging tools of LabVIEW were used systematically when validating ATAC software components. Third section of a VI is called connector pane. Connector pane is the interface that other VI’s see when calling another VI. Connector pane defines the interface to a VI and allows other VI’s to access controls and indicators of the VI. When a VI calls other VI in its block diagram the called VI is usually referred to as sub VI. [13]

LabVIEW has the same common programming structures such as if-else, do-while, event handlers, etc. as many other programming languages. The main difference is that instead of writing code as text commands the user draws blocks and wires them together with wires. With text based languages like Java and C++ the order of the instruction execution is clear. The instructions are executed in the order they are written. With LabVIEW the order of the instruction execution is not so straightforward. If user for example constructs two parallel loops that do not depend on each other the order of execution between the loops is not defined similarly as with text based programming languages. This is because the two loops/blocks can be drawn where-ever on the sheet and their execution order does not depend on their position on the sheet. This is illustrated in Figure 4 where two while-loops are executed in parallel. The order of the program execution is defined by data flow [7]. When an element has all of its inputs defined it is executed. If more than one element has its inputs defined at the same time their execution order is random. [13] In the Figure 4 neither of the loops have any inputs so they are both ready to execute at the same time. The order of execution in this example goes as follows:

1. Program execution randomly selects Loop 1 as first element and steps into it.
2. Execution steps into the Loop 2 because it was waiting its turn from previous step.
3. Register “Channel 0” as input for “Read from I/O” VI in Loop 1
4. Register “Channel 2” as input for “Read from I/O” VI in the Loop 2
5. Inputs for VI “Read from I/O” in the Loop 1 are defined so the block is executed.
6. Inputs for VI “Read from I/O” in the Loop 2 are defined so the block is executed.
7. VI “Read from I/O” in Loop 1 has produced output which is input for “BandPass Filter” so “BandPass Filter” VI is executed.
8. Etc.

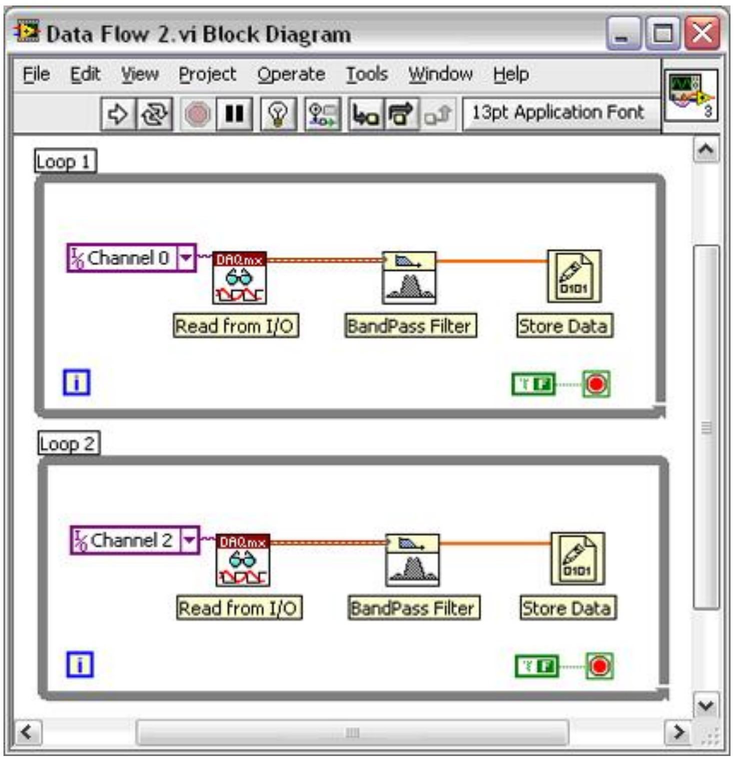


Figure 4. Parallel execution.

The example above illustrated how program execution order is defined by data flow instead of the positions between different instructions. This is why LabVIEW is called data flow programming language. Using structures that are executed in parallel provides a way to execute different tasks simultaneously similarly as use of threads makes it possible to execute different parts of code simultaneously in text based languages. The analogy with parallel structures in LabVIEW and threads in text based languages is not very accurate because LabVIEW also uses threads on a lower level. Even though a LabVIEW program would not have any parallel structures it might be divided into several threads by LabVIEW IDE when the program is compiled and executed. This is usually hidden from the programmer. It is possible to force the number of threads and control the division of tasks between multiple processors but in most cases programmer can ignore this and let the IDE handle low level threading. [20]

4.2.2 Controlling hardware with LabVIEW

LabVIEW is designed to be easy to use programming environment that can be used to create data acquisition applications with very little effort. Many different instruments have LabVIEW drivers available at National Instruments website and on the website of the instrument manufacturer. LabVIEW drivers for instruments are VIs that have inputs and outputs and they allow fast way to use instruments remotely from the software. LabVIEW has built-in abstraction layer called VISA (Virtual Instrument Software Architecture) which provides an easy way to handle and control hardware connected to different communication buses. [18]

4.2.3 OOP in LabVIEW

LabVIEW has long been a way to build small applications that interact with hardware with little effort needed at programming. Object oriented programming would be overkill in these small applications where the structure of the software does not play vital role. LabVIEW does however provide structures and ways to program using object oriented style [11]. The support for object oriented programming can be seen as inevitable development because of the popularity of object oriented programming today. LabVIEW has also become a language that is used to program larger applications that need modular and maintainable structure. [15]

The difference between LabVIEW classes and classes in for example Java and C++ is that LabVIEW classes do not have constructors. An instance of a class is created by drawing a wire from a block diagram control that represents the class. The wire is the created object and it can be routed throughout the software. Like other OOP languages LabVIEW also supports inheritance and polymorphism which enabled the construction of a hardware abstraction layer into the ATAC software.

4.3 Software life cycle management

Planning of a software life cycle plays important role when developing new software. Traditional waterfall type of flow where each development phase is performed once and then proceeded to a next phase might not be efficient way of managing the software development in many cases. In fact many software development projects use more agile strategies where different phases are executed simultaneously in some degree and the shift from one phase to another is not a one way transition. This enables earlier prototypes of the software and bad features can be modified at early stage of the development process. [4] Different software life cycle models have been studied and advantages and disadvantages of different models have been presented [1]. Especially with the ATAC software described in this paper the traditional waterfall model was not reasonable approach because the specification and requirements for the software were likely to change throughout the development process. This is why the software development flow for ATAC was managed using the principles of spiral flow [4] shown in the Figure 5. A similar agile model called Twin peaks model was presented by Nuseibeh [21] and was also studied for life cycle managing of ATAC development. With ATAC development there was not much time spent in making requirements at the beginning of the project because it was not known exactly what the requirements would be and it would be likely that the requirements will change and become more precise when users can get a feel of early stage prototypes of the software. Using a risk driven spiral type of software development model the risks related to the user interface and other parts of the software were handled in a reasonable fashion. The Figure 5 which illustrates the spiral flow of ATAC development can be interpreted in the following way: The development is started at near the center of the spiral. After designing and implementing some aspects of the software new round of spiral is started. At each iteration the software becomes better and the focus changes from developing software towards maintaining the software. Software components that hold greatest risks are implemented first to make sure the essential parts of the software can be implemented and that they work as designed. This avoids situation where large part of software needs to be rewritten if essential parts of the software need to be changed.

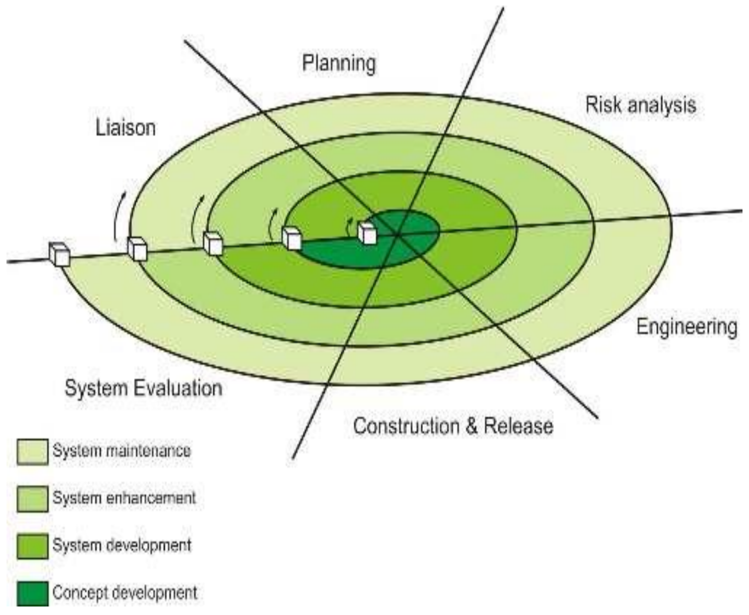


Figure 5. Spiral model of the software development project. [30].

5 ATAC – Automated Test And Characterization software

In this chapter the control and data collection software named ATAC that was developed in this thesis is presented. First the requirements for the software are presented to give the reader an overview of the features and capabilities that were implemented in ATAC. After that the reader is given an example of how test scripts are written and loaded into ATAC. The use of ATAC to run scripts and to control hardware is described and associated GUI windows are presented. This gives an overview of how the ATAC is used and the inner structure and functionality of the software is easier to understand in later subchapters. The structure of the software is described in class level and the block diagram code is described using state charts. Methods used for testing the software are described at the end of this chapter.

5.1 Application overview

Control and data collection software, ATAC, described in this paper is designed to provide high level of automation for ASIC evaluation. The goal is that user can write test sequences that the software executes independently. The software must be generic enough so that the same software can be used with different PCB hardware, different instruments and different ASICs. At the same time it must be easy to use, intuitive and quickly adoptable so that user can maintain focus on debugging and testing the DUT and not struggling with GUI that has too many adjustable parameters and features.

ATAC enables user to execute tests automatically or by instruction at a time displaying real-time information about test and measurements to the user. Debugging of DUT is important aspect when trying to characterize a device so ATAC also provides ways of sending commands to different components of the system in the middle of test execution using manual communication windows. ATAC also abstracts the ACE board control and provides GUI for controlling ACE board relays and other functionality.

5.2 Specification and planning

The development of ATAC was performed using principles of Spiral model [4] which is a risk driven style of managing software development. Greatest risks in the ATAC development were related to communication with hardware and the style how tests are created by user. The communication with hardware was perhaps the most essential part of the software so it was implemented first. When working VI’s for communication with hardware PCBs and instruments were successfully implemented other parts of the software were designed. User interface was a big risk as well as general usability of the software so GUI prototype was sketched at an early stage of development. Sketching GUI is easy with LabVIEW because every VI has a front panel where readily available buttons, graphs and other components can easily be added. The GUI defines the style how software would be used and sets requirements how the program structure should be implemented.

Other significant risk was related to implementing parallel style of test execution. The parallel test execution means that test sequences should be able to run at the same time as other features of the ATAC are used. The parallel nature of execution defines the top level structure of the software and implementing the top level code at an early development stage avoids large changes in the code later.

By using Spiral model approach the development of ATAC did not follow either bottom-up nor top-down style of software development. Both, bottom level hardware communication and top level software structure were implemented in early stages of development to minimize risks and possibilities of having to rewrite software.

Although requirements for the software were specified throughout ATAC development some initial requirements were defined at the beginning of the development project. The requirements were defined in co-operation with target users. The initial requirements were

- Automatic execution of test files
  - User must be able to create test scripts and execute them in automatic fashion using the software. Queuing test scripts should also be possible to allow the user to divide large tests in different files.

- Pausing and single stepping test commands
  - Test scripts created by user can contain errors that cause the test to fail or to produce unreasonable results. The device under test can also behave unpredictably. If whole test script is executed automatically it is impossible for the user to see what instruction caused the failure. This is why the software has to have capabilities to execute test scripts line by line and to pause the test execution if continuous execution mode is used.

- Recovering from error states automatically
  - The software should not stop test execution in every error situation. Errors that can be handled without human interaction should be handled automatically to minimize the execution time of tests.

- Manual control interfaces for adapters
  - User should be able to send commands through adapters whenever a test sequence is not running. This enables the software to be used as a generic debugger and communication interface towards different devices. In the middle of a test manual commands should be able to be sent to DUT to read state of the device.

- Hardware abstraction layer for instruments and adapters
  - Support for new instruments and adapters should be able to be added to the software with minimal modifications to the existing code to keep the software clean and manageable. The main software should also be able to execute without knowing how devices are handled at lower levels of the software.

- Parallel execution between test execution and other features
  - User must be able to execute test sequences in background while performing other tasks with the same software. For example if a graphical creation of test scripts is added to the software in the future, user must be able to create tests with the software at the same time other tests are being executed.

- Control of ACE board power up/down with changeable instruments and adapters
  - The software should be able to control ACE board power because manually powering the ACE board is difficult. Different instruments could be used to power the ACE board so the software should be flexible enough to handle any instrument as ACE board power.

- Use of multiple adapters simultaneously
  - Different components of the evaluation system hardware use different communication protocols and different adapters need to be used at the same time. The software must support simultaneous use of multiple adapters regardless of the adapter type.

- Real-time graphical examination of measurement results
  - User needs to be able to track all measurements at real-time while a test is being executed. The measurements should be displayed in a graph.

- If-else structure for tests
  - User has to be able to construct consecutive and overlapping conditional statements and structures in test scripts.

- Loop structure for tests
  - User has to be able to construct consecutive and overlapping loop structures in test scripts.

- Storing of measurement results to an Excel compatible format
  - Measurements should be automatically saved to file system in a format that can be opened with Excel. The measurement data should be in easily viewable and usable matrix.

The control and data collection software described in this paper can be regarded as a fairly large LabVIEW application and to keep the software development manageable a good programming style had to be practiced. The software has to be easily modifiable, scalable, generic and modular. It is likely that new functionality will be added throughout the application’s life cycle. The same software must also work with different instruments, adapters and PCB hardware. The structure of the software must be designed so that these requirements are fulfilled.

Before presenting the implemented ATAC architecture the reader is given a description of how the ATAC is used. This provides the reader a better understanding of the ATAC features and makes it easier to understand the software architecture beneath.

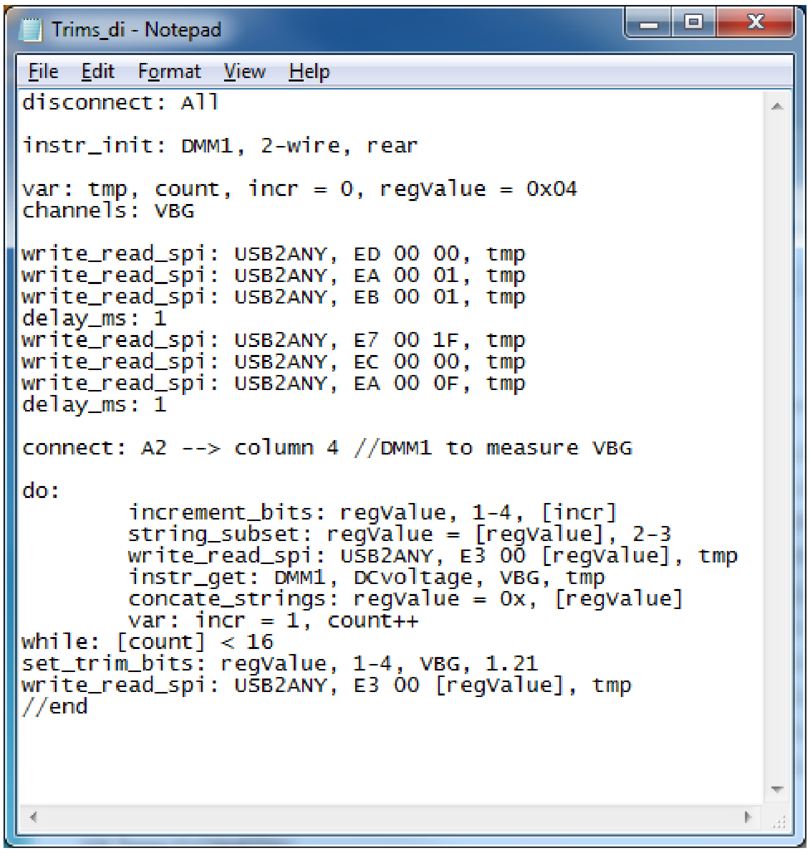
5.3 Using ATAC

5.3.1 Creating tests

Tests can be written using any text editor. During the development of ATAC Notepad was most commonly used to write test sequences. ATAC uses the following syntax for commands:

<command>: <parameter 1>, <parameter 2>, ... , <parameter n>

The double dot (:) separates the command from parameters. The number of parameters as well as syntax of parameters depends on the command. Every command requires unique parameters that can be mandatory or optional. A dash (,) is used to separate commands from each others. White spaces can be freely used around double dot (:) and dash (,) as well as in the beginning and the end of the line. Text can be commented by adding // before the text. Everything that comes after the comment mark at that line is ignored by the instruction handler in ATAC. An example of a test script is presented in Figure 6. A description for the example script is given below the figure.



```
disconnect: All

instr_init: DMM1, 2-wire, rear

var: tmp, count, incr = 0, regValue = 0x04
channels: VBG

write_read_spi: USB2ANY, ED 00 00, tmp
write_read_spi: USB2ANY, EA 00 01, tmp
write_read_spi: USB2ANY, EB 00 01, tmp
delay_ms: 1
write_read_spi: USB2ANY, E7 00 1F, tmp
write_read_spi: USB2ANY, EC 00 00, tmp
write_read_spi: USB2ANY, EA 00 0F, tmp
delay_ms: 1

connect: A2 --> column 4 //DMM1 to measure VBG

do:
        increment_bits: regValue, 1-4, [incr]
        string_subset: regValue = [regValue], 2-3
        write_read_spi: USB2ANY, E3 00 [regValue], tmp
        instr_get: DMM1, DCvoltage, VBG, tmp
        concate_strings: regValue = 0x, [regValue]
        var: incr = 1, count++
while: [count] < 16
set_trim_bits: regValue, 1-4, VBG, 1.21
write_read_spi: USB2ANY, E3 00 [regValue], tmp
//end
```

Figure 6. Example of test script.

The script presented in Figure 6 trims (adjusts) bandgap [26] voltage reference of an ASIC device to a desired value. The used commands are described next:

disconnect: All

Disconnects all instruments from bench EVM board by opening instrument relays on ACE board. Opens also all power relay connections on ACE board.

instr_init: DMM1, 2-wire, rear

Initializes multimeter called DMM1 that is used to measure bandgap voltage from DUT. Parameter “2-wire” tells the DMM1 to use 2-wire measurement mode and parameter “rear” instructs the DMM1 to use rear terminals.

var: tmp, count, incr = 0, regValue = 0x04

Initializes variables and assigns values to them. No values for variables “tmp” and “count” are given so they are both initialized to 0. Variable “tmp” is used as a general purpose variable to temporarily store various different information. Variable “count” is used as loop iteration counter. Variable “incr” is used to increment a bandgap register value of an ASIC at every loop iteration. Variable “regValue” represents a value of an 8-bit register.

channels: VBG

Defines a channel called VBG where bandgap measurements are stored. Channel data is written to a file when test execution is finished. While executing test the VBG data can be examined on front panel graph in real-time.

```
write_read_spi: USB2ANY, ED 00 00, tmp
write_read_spi: USB2ANY, EA 00 01, tmp
write_read_spi: USB2ANY, EB 00 01, tmp
delay_ms: 1
write_read_spi: USB2ANY, E7 00 1F, tmp
write_read_spi: USB2ANY, EC 00 00, tmp
write_read_spi: USB2ANY, EA 00 0F, tmp
delay_ms: 1
```

Sequence of SPI (Serial Peripheral Interface Bus) instructions are send to adapter called USB2ANY. These set the DUT to appropriate state for bandgap voltage measurement. Data that is sent to SPI bus is given in groups of 8 bits. SPI configuration has already been done at a separate window before starting the test execution. For example the first command sent to SPI bus is ED 00 00 which in binary format is 11101101 00000000 00000000. Variable “tmp” is used to temporarily store data that is read from SPI.

connect: A2 --> column 4 //DMM1 to measure VBG

Connects ACE board terminal A2 to bench EVM column 4 by closing appropriate relay on ACE board. DMM1 connected to ACE board terminal A2 now has connection to bench EVM column 4 and can read bandgap voltage from the DUT.

do:

Starts loop structure.

increment_bits: regValue, 1-4, [incr]

Gets bits from regValue variable at indexes 1,2,3 and 4 starting from least significant bit and increments this value by the amount defined by variable “incr”.

Value of “incr” is accessed by adding “[“ and “]” characters around variable name. Bandgap voltage reference is trimmed by bits 1,2,3 and 4 and other bits of the same register are left unchanged. The new register value is stored back to variable “regValue” after the increment.

string_subset: regValue = [regValue], 2-3

Prefix “0x” is removed from “regValue” variable and the stripped value is stored back to “regValue” variable. At first loop iteration the value of “regValue” before this command was 0x04 because at the first iteration of loop the value is incremented by amount of 0. After execution of this command the value of “regValue” was 04 because the command was instructed to get characters at indexes 2 and 3 starting from the left.

write_read_spi: USB2ANY, E3 00 [regValue], tmp

The incremented register value is sent to SPI bus using adapter called USB2ANY. This sets the bandgap register value of the DUT.

instr_get: DMM1, DCvoltage, VBG, tmp

Multimeter called DMM1 that was connected to measure the device bandgap voltage is instructed to perform DC voltage measurement. The read measurement is stored in channel “VBG” and in variable “tmp”.

concate_strings: regValue = 0x, [regValue]

A prefix “0x” is added to the value of variable “regValue”. The prefix is required by commands “increment_bits” and “set_trim_bits”.

var: incr = 1, count++

Sets variable “incr” value to 1. Variable “incr” is used to define increment amount of bandgap register value at each loop iteration. Loop iteration count is stored in variable “count”.

while: [count] < 16

This command decides whether to jump to previous “do” command or to exit the loop. If value of variable “count” is under 16, then the execution jumps to line where previous “do” command was given. If value of “count” is 16 or over the loop is exited and execution continues from next line. Bandgap is trimmed with four bits so to go through all the register values for bandgap the loop must be executed 16 times.

set_trim_bits: regValue, 1-4, VBG, 1.21

When this command is executed the previous loop has been iterated 16 times. All bandgap register values have been sweeped through and bandgap voltages for each register value have been measured and measurements stored to channel

"VBG". This command goes through the "VBG" channel data and finds index of measurement that is a closest match to specified 1,21 target voltage. This index is the same value that needs to be written to bandgap register of the device to produce the 1,21 voltage. The found index is written to the variable "regValue" that holds bandgap register values.

write_read_spi: USB2ANY, E3 00 [regValue], tmp

The solved register value that produces the desired bandgap voltage is written to the DUT register using adapter USB2ANY and SPI bus.

//end

Comment that is ignored by ATAC. Helps the user to know this is the end of test.

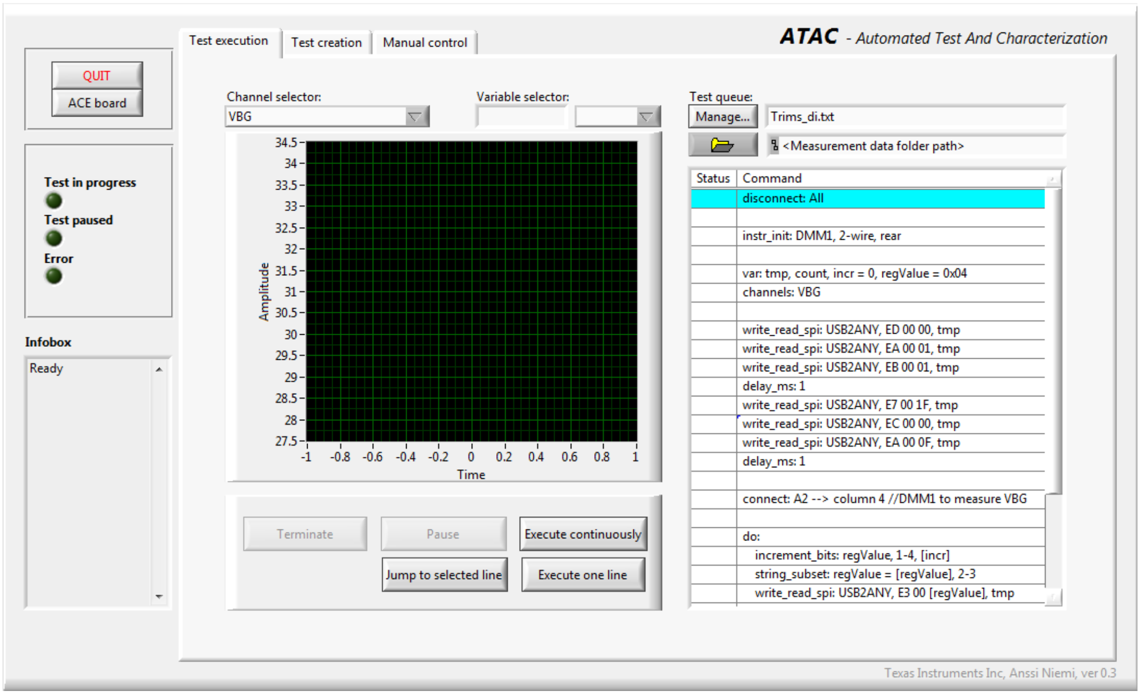
Test file is loaded into the ATAC software directly in the same text format it is written. Test is not compiled before execution and the execution of the test is done line by line for the text instructions. This way user can follow the execution of the test in "Test execution" window which is the main window of ATAC.

ATAC supports use of variables, arrays and channels for passing information through a test. Variables can be written and read and the values of variables can be viewed at main screen of ATAC in real-time. Arrays can be read but not written. Arrays can be used for example to define a series of values that define supply voltage of DUT through each iteration of a measurement loop. Channels can be written but not read. Channels are the storage for measurement data gathered throughout the test. All channel data is visible to user in a graph format on main window of ATAC. ATAC supports do-while loop structure and if-else conditional structures. In addition around 40 different commands are supported. Full list of supported commands can be found at appendix A.

Individual test scripts can be queued in ATAC which means different tests can be executed automatically one after another. This keeps the number of commands in a single test small and the tests remain manageable to the user. Automatically executing multiple test files sets responsibilities for the user to make sure the end of a test is compatible with a start of next test. Usual way of doing this is by disconnecting and closing all instruments at the end of a single test and initializing everything at the beginning of next test. This ensures that no voltages or short circuits exist in the hardware when starting a new test. In addition to executing one test in one file the queuing provides easy way of managing the file system hierarchy where the measurement data is stored. Adding a test file that changes the measurement data folder to queue automatically divides measurement data into manageable file system hierarchy.

5.3.2 Executing tests

The "Test execution" window which is the main window of ATAC is described next. Screenshot of the window is presented in Figure 7. This is the window where user can load tests and execute them. The graph in the middle shows measurements in real-time so user can easily monitor the progress of the test. Multiple tests can be queued and measurements are automatically stored in measurement files.



Status	Command
	disconnect: All
	instr_init: DMM1, 2-wire, rear
	var: tmp, count, incr = 0, regValue = 0x04
	channels: VBG
	write_read_spi: USB2ANY, ED 00 00, tmp
	write_read_spi: USB2ANY, EA 00 01, tmp
	write_read_spi: USB2ANY, EB 00 01, tmp
	delay_ms: 1
	write_read_spi: USB2ANY, E7 00 1F, tmp
	write_read_spi: USB2ANY, EC 00 00, tmp
	write_read_spi: USB2ANY, EA 00 0F, tmp
	delay_ms: 1
	connect: A2 --> column 4 //DMM1 to measure VBG
	do:
	increment_bits: regValue, 1-4, [incr]
	string_subset: regValue = [regValue], 2-3
	write_read_spi: USB2ANY, E3 00 [regValue], tmp

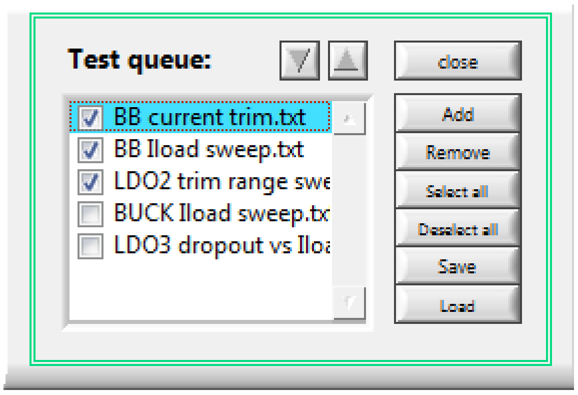
Figure 7. ATAC Test execution window.

When tests are loaded and a measurement file path is defined the test execution can be initiated. There are options for different execution modes such as "Execute continuously", "Execute one line" or "Jump to selected line". In case of error occurs the user is given possibilities to continue the execution from any line or terminate the test. In noisy environment errors can easily occur in communication lines so instead of waiting for user input in every error event the program first tries to solve the error by itself and continue execution. This will save time in cases where tests are left to run for long periods of time without human monitoring.

The main window shows values of variables and channels that are defined in the test script in real-time. This helps the user to follow test progress and to see if the measurements are in reasonable range. Measurement channels are where the measurements are stored. Selecting channel from pull-down list shows all measurements for that channel on the graph.

The software is designed so that test execution runs independently in its own loop and other tasks in another loop. This enables the user to use other features of ATAC at the same time as tests are being executed in the main window. One of the requirements for the software was to be able to pause a test at any state and to be able to send manual commands to the DUT and instruments and then continue the test execution. Pausing a test can be done using pause button or by adding a pause command to the test file at desired line.

ATAC supports also queuing of the test files meaning that multiple different tests can be executed automatically one after another. The loading of tests for execution is done by pressing button "Manage…" under "Test queue:" label on front panel (Figure 7). This opens a queue management window that is shown in Figure 8. User can save and load queues and select which tests of the queue are loaded into ATAC. The order of the tests is also changeable. By closing the window the selected tests names and paths on file system are loaded into ATAC memory. The first test is loaded into ATAC front panel for execution. After the first test has executed the second test in the queue is loaded and executed.



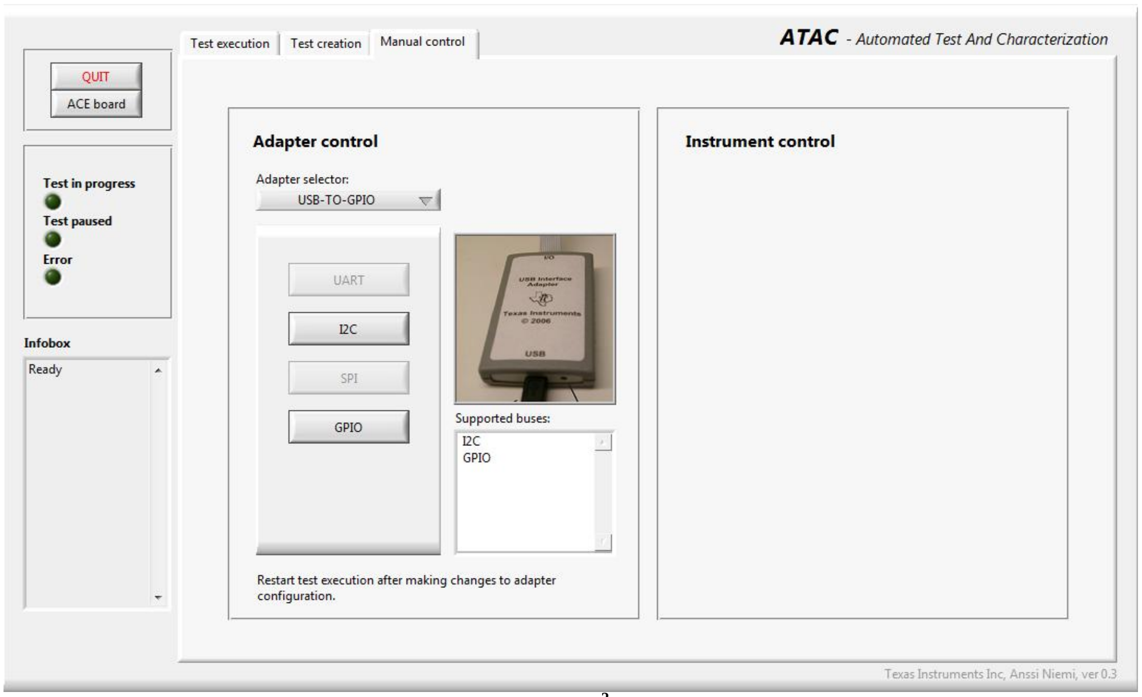
☑ BB current trim.txt
☑ BB Iload sweep.txt
☑ LDO2 trim range swe
☐ BUCK Iload sweep.tx
☐ LDO3 dropout vs Iloa

Figure 8. Loading of test scripts into ATAC.

5.3.3 Adapter control

ATAC provides manual control over adapters and ACE board. Manual control window is used to set the adapters into desired state before starting to execute tests. This reduces the number of initialization commands needed in the test sequence and makes the initialization easy through buttons and pull-down menus.

Adapter control window provides a graphical interface to communicate with adapters. Adapter features can be set and specific commands can be sent to adapters. Many adapters provide I^{2}C, SPI, UART or GPIO communication towards DUT so parameters and configuration of these protocols can be set using this window. When debugging the DUT it is also convenient to have a manual interface for communication. Screenshot of manual control window is shown in Figure 9. The view changes when selecting a different adapter that supports different communication protocols. This is shown in Figure 10. Pressing one of the buttons (“UART”, “I2C”, “SPI”, “GPIO”) opens a pop-up window for that communication protocol.



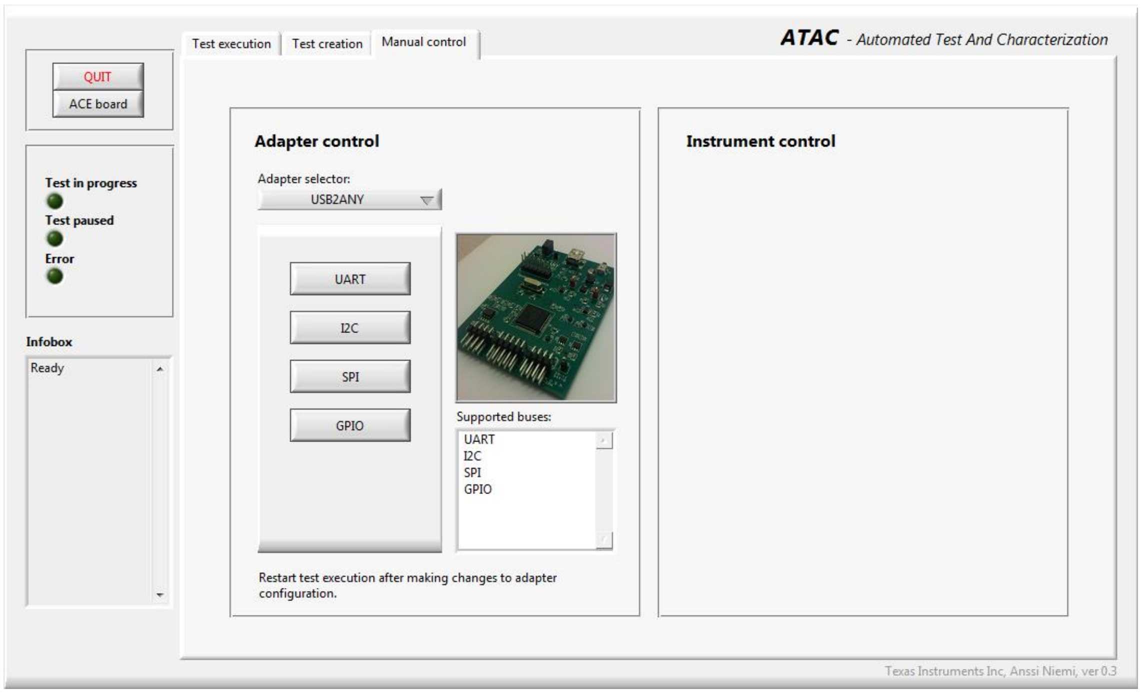
Adapter selector: USB-TO-GPIO

Infobox: Ready

Supported buses:
I2C
GPIO

Restart test execution after making changes to adapter configuration.

Figure 9. Manual control window. Adapter supports only I^2C and GPIO.



Adapter selector: USB2ANY

Infobox: Ready

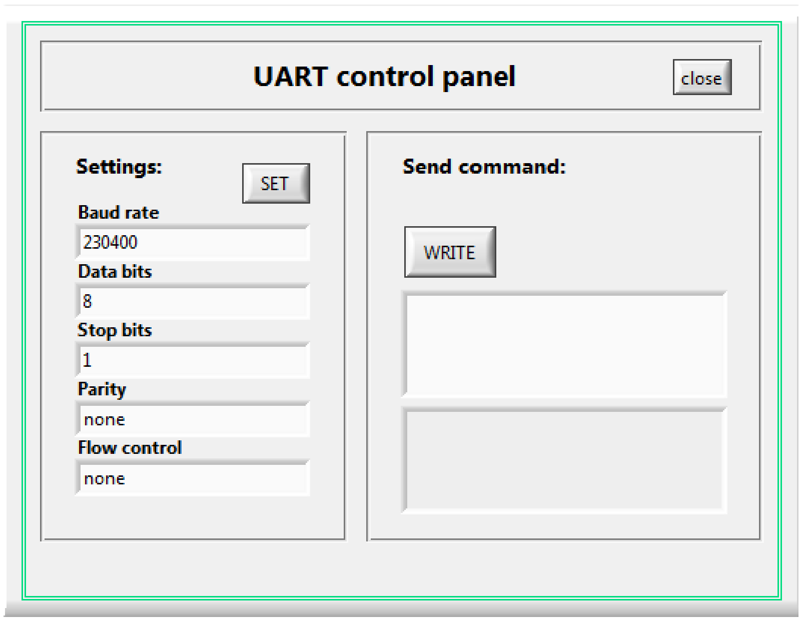
Supported buses:
UART
I2C
SPI
GPIO

Restart test execution after making changes to adapter configuration.

Figure 10. Manual control window. Adapter supports all communication protocols.

An adapter object tells the main software what buses it supports and the main software can show supported buses for the user. Pressing a button corresponding to a specific bus user gets to configure the bus parameters and send/read commands. Each communication protocol has its own pop-up window for setting the communication parameters and sending commands and they are presented below. Manual instrument control and “Test creation” tab seen in Figure 9 are place holders for possible features that might be added to the software in the future.

UART (Universal Asynchronous Receiver/Transmitter) is an asynchronous communication protocol and very common in various different devices. One of the reasons for UART's popularity is its simple operation. As a drawback only one master and one slave device can be connected to the bus [10]. If an adapter supports UART the "UART" button is visible in the adapter control window. Pressing the button a pop-up window for configuring UART settings and sending commands is opened. The pop-up window is presented in Figure 11. Through this window user can send UART commands to a slave and read back data the slave sends.


UART control panel

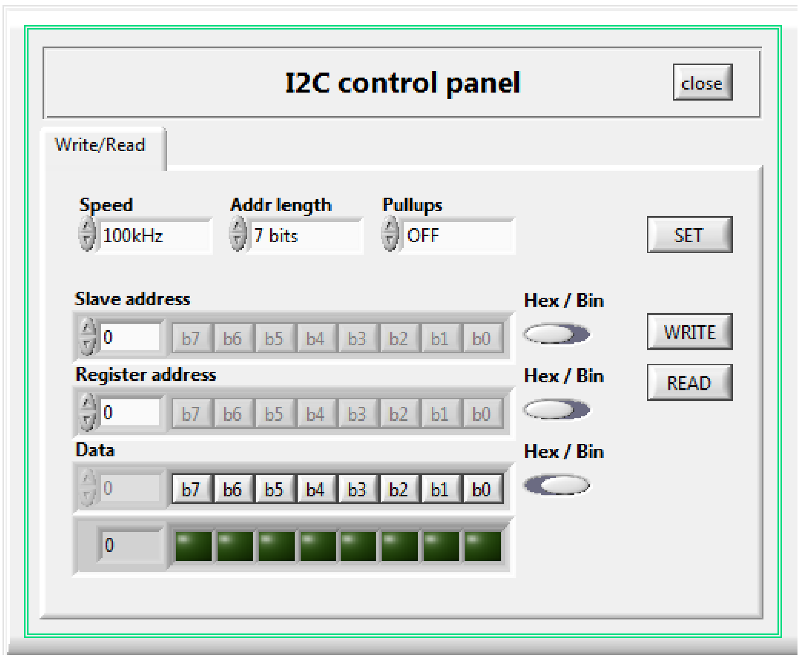
Settings:

Baud rate: 230400
Data bits: 8
Stop bits: 1
Parity: none
Flow control: none

Send command:

Figure 11. UART communication window.

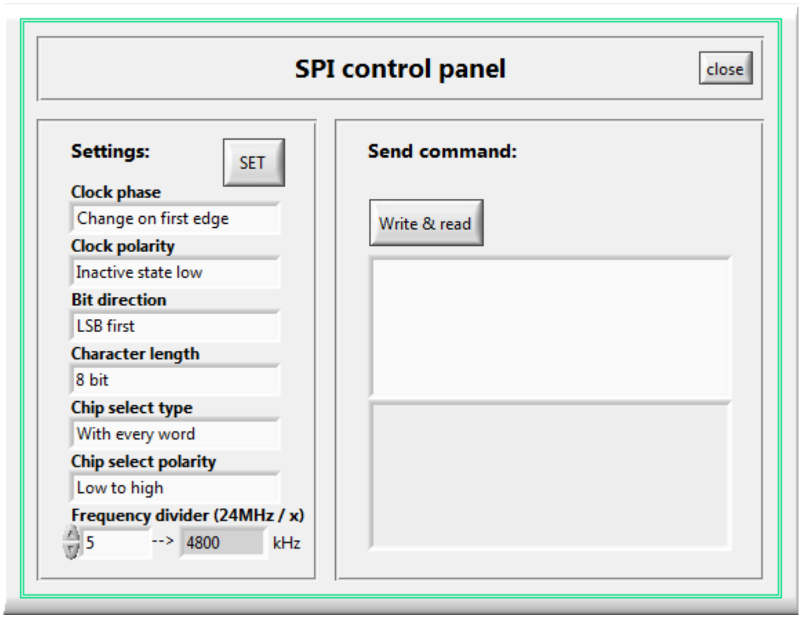
I^{2}C is another commonly used communication protocol in embedded systems and ASICs. I^{2}C uses only two lines for communication which makes it possible to minimize the number of pins when designing an ASIC. In addition multiple devices can be connected to a same I^{2}C bus and every device can act as a master or as a slave. Drawback of I^{2}C is its limited speed [10]. The ACE board presented in this paper also uses I^{2}C for communication. I^{2}C commands can be sent as well as setting I^{2}C configuration parameters in a window presented in Figure 12. Data can be written and read in binary or hexadecimal format.



Speed: 100kHz
Addr length: 7 bits
Pullups: OFF
Slave address: 0
Register address: 0
Data: 0
0

Figure 12. I^2C communication window.

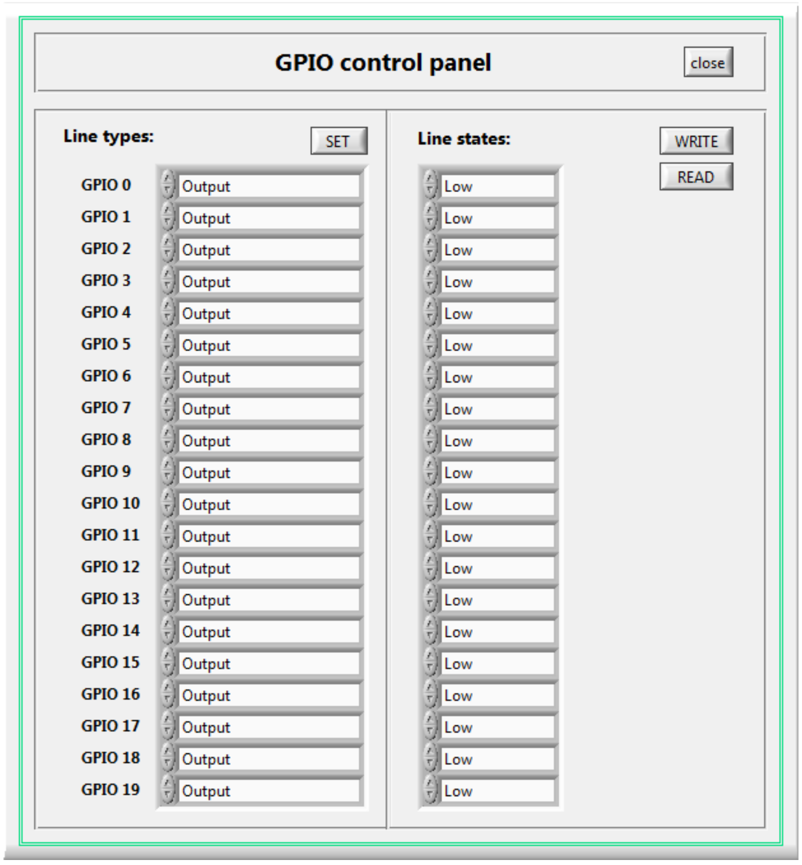
SPI (Serial Peripheral Interface Bus) is a commonly used communication protocol that is also supported by the ATAC. SPI communication uses 3 + n wires, where n is the number of slaves connected to the bus. Large number of wires limits the use of SPI if more than one device is connected to the bus. On the other hand the SPI can reach higher baud rates than I^2C and UART. [10] Communication window for SPI is presented if Figure 13. As with UART and I^2C user can similarly configure SPI settings and send commands. In SPI communication the master (ATAC) and a slave change information at the same time. There is no separate write and read commands but the information to both directions is transmitted in a same communication cycle.



Settings:
Clock phase: Change on first edge
Clock polarity: Inactive state low
Bit direction: LSB first
Character length: 8 bit
Chip select type: With every word
Chip select polarity: Low to high
Frequency divider (24MHz / x): 5 --> 4800 kHz

Figure 13. SPI communication window.

GPIO (General Purpose Input/Output) is not a communication protocol like the UART, I^{2}C and SPI. With GPIO user can set lines to be digital inputs or digital outputs and select impedance of the line. User can also select the state of the line to be high or low. The GPIO does not define any protocol how the lines are used for communication. That is left for the user to define. Usually GPIO lines are used as a general purpose digital I/O to control and read states of individual nets. GPIO control window is presented in Figure 14. ATAC supports twenty different GPIO lines that can be operated as inputs or outputs.



	Line types:	Line states:
GPIO 0	Output	Low
GPIO 1	Output	Low
GPIO 2	Output	Low
GPIO 3	Output	Low
GPIO 4	Output	Low
GPIO 5	Output	Low
GPIO 6	Output	Low
GPIO 7	Output	Low
GPIO 8	Output	Low
GPIO 9	Output	Low
GPIO 10	Output	Low
GPIO 11	Output	Low
GPIO 12	Output	Low
GPIO 13	Output	Low
GPIO 14	Output	Low
GPIO 15	Output	Low
GPIO 16	Output	Low
GPIO 17	Output	Low
GPIO 18	Output	Low
GPIO 19	Output	Low

Figure 14. GPIO control window.

5.3.4 ACE board control

The GUI shown in Figure 7 and Figure 9 is not tied to any hardware and enable the ATAC to be used as a standalone test sequencer and debugger. The functionality related to ACE board is behind “ACE board” button. The ACE board functionality is integrated into the software but in such fashion that it is easily modified for other future ACE boards and different hardware. In practice the ACE board class would have to be replaced by new class representing new board but no other parts of software would have to be altered. The ACE board user interface is shown in Figure 15.

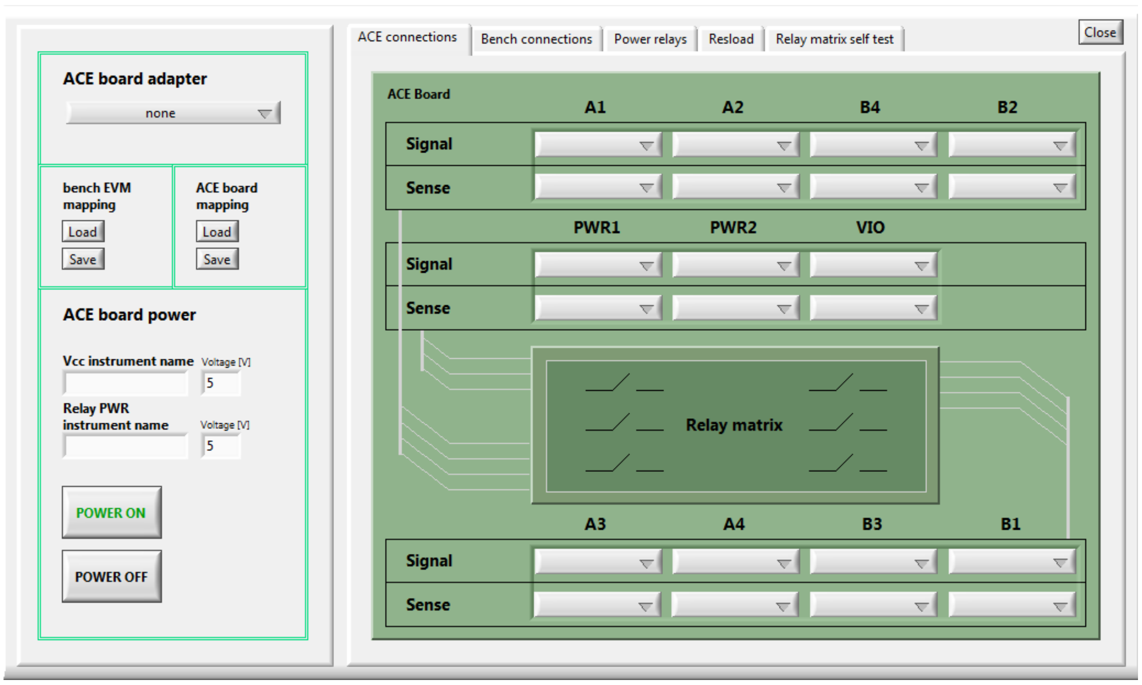
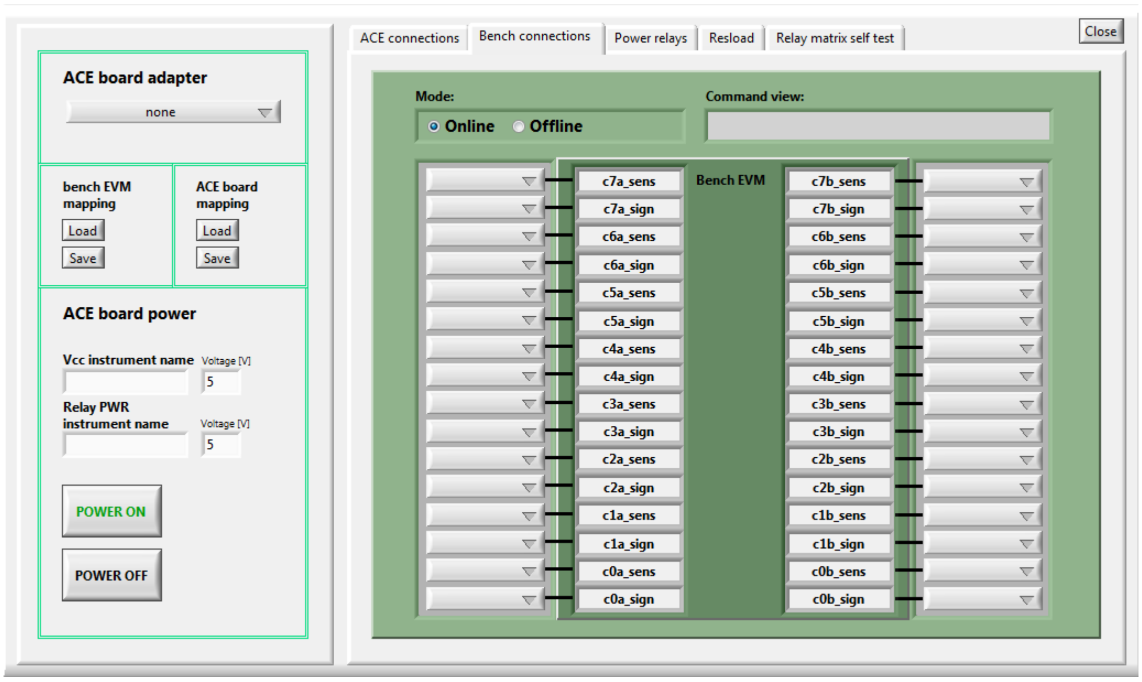


Figure 15. ACE board GUI window main tab.

On the left user can control the ACE board supply voltage and load/save instrument mappings of ACE board and pin name mappings of Bench EVM board. Using mappings the actual terminal and pin ID's present on the PCBs can be hidden and instead user can use self defined terminal and pin names. This enables the same GUI to be used with different hardware setups that have different pin and terminal configurations.

The ACE connections tab can be used to define instrument to ACE board terminal mapping. User can select from pull-down lists which instrument is connected to which terminal and after that this mapping can be used elsewhere in the program. This way the user does not have to refer to terminals A1, A2, etc. but can refer to actual instrument names when routing instruments from ACE board to bench EVM board.

Bench connections window (Figure 16) can be used to configure instrument routing from ACE board to bench EVM board. Bench EVM pin names can be changed and saved to match specific bench EVM board which makes it easy for user to define instrument connections. From pull-down lists user can select which of the available instruments connect to what pin on bench EVM board. Depending on the selected mode, instrument connection commands are sent to ACE board hardware at real-time or if the mode is offline, the connections only apply at the software but no hardware connections are made. Offline mode can be used if user wants to create a connection command and use it later in a test sequence. Other tabs are defined for future development but not yet implemented.



ACE board adapter: none

Vcc instrument name — Voltage [V]: 5

Relay PWR instrument name — Voltage [V]: 5

Mode: [✓ Online] Offline

Figure 16. ACE board GUI window bench connections tab.

5.4 Application structure

5.4.1 Classes

In addition to meeting the requirements specified for the software the software must be modular, scalable and easily modifiable. Therefore the structure of the software must be designed carefully. The software has to be easily modifiable so that new functionality can be implemented throughout the software life cycle. Object oriented programming has proven itself as a good programming style for large applications so the ATAC was also designed by the principles of OOP.

The functionality of the software is divided into different classes. For example class named “Adapter” represents an adapter that is connected to PC and has all the methods/VIs that can be used to communicate with an adapter. Different adapter classes that represent some specific adapter inherit the generic “Adapter” class and thus have all the same methods/VIs as the “Adapter” class. This also enables hardware abstraction [19] because the main software can use Vis of the class “Adapter” to communicate with variety of different adapters that are implemented separately from the main application. Same principle is applied to instruments. The main application has classes such as “Multimeter”, “Source Meter”, “Power Supply” etc. and it does not have to know which specific instrument is connected to the system. This makes it possible to use the same application with different instruments that are connected to PC through different communication buses.

The execution of test sequences can be run parallel to other tasks such as viewing adapter configurations. Synchronization between different parallel tasks is needed because some tasks can otherwise conflict with each other or cause unwanted behavior. For example sending manual commands to an adapter should be forbidden when a test is running. Otherwise the manually sent command might cause the DUT to go into unwanted mode or cause other unpredictable conflicts. A class named “Mode

Controller" is a synchronization class that controls other classes and test execution. It holds information about the states of parallel tasks and makes sure no conflicting tasks are executed simultaneously.

GUI events such as user pressing a button are registered with event handlers that transmit the information to the "Mode Controller" object. The mode controller then decides whether the instruction can be executed and issues commands to other objects and tasks. Figure 17 illustrates the structure of the main software using UML (Unified Modeling Language).

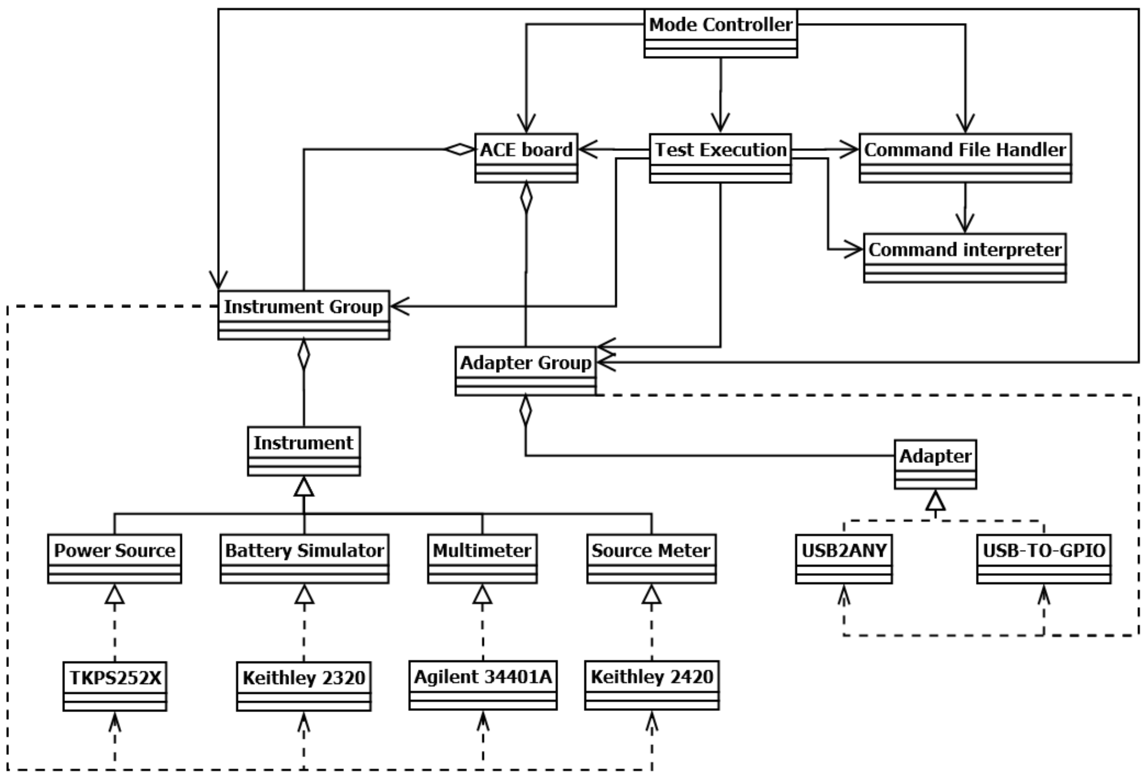


Figure 17. UML Class diagram of the software.

"Mode Controller" holds and uses the following objects:

- "Instrument Group". Holds all instrument objects.
- "Adapter Group". Holds all adapter objects.
- "Test Execution". Handles test script execution. "Test Execution" holds its own instances of instruments, adapters, "ACE board", "Command Interpreter" and "Command File Handler".
- "Command File Handler". Handles write/read operations of command files.
- "ACE board". Represents the PCB board that routes instruments to bench EVM board.

New instruments and adapters are added as external software plug-ins that override methods/VIs defined by generic "Power Source", "Battery Simulator", "Multimeter", "Source Meter" and "Adapter" classes. This enables hardware abstraction that is discussed next.

5.4.2 Hardware abstraction layer

Hardware abstraction layer (HAL) enables the ATAC software to be used in different environments and with different hardware without having to modify the main software for each environment separately. HAL was designed by using factory design pattern [19]. Instruments and adapters are behind the HAL. These are both components that have certain common features that are generic and independent of the manufacturer or model. A common interface is defined for instruments and adapters and the main software uses generic VI's defined by the interface. The generic VI's are overloaded by actual instrument/adapter plug-in software at run-time. This is possible because the main software does not care how the interface method is implemented. Every hardware object has its own software plug-in written that implements the defined interface. The separate plug-in is linked to the main software by a configuration file that tells the main software where the implementation can be found at file system.

Instruments are divided into following sub categories:

- Multimeter
- Power source
- Source meter
- Battery simulator.

These all have unique interface that the instrument plug-in must implement. All multimeters support roughly the same functionality and an interface that satisfies all different multimeters is easy to design. The same applies to all the other instruments. Adapters on the other hand are a bit trickier to handle. All adapters must use the same predefined interface although a variation between different adapters can be quite large. This problem is solved by making the overloading of interface methods optional. Only some VI's are required to be overloaded by the adapter plug-in. Such VI's are "initialize", "close" and "get supported buses". With "get supported buses" VI the adapter plug-in tells the main software what buses and communication methods the adapter supports towards hardware components. This way the main software can display buttons and functionality that applies to the connected adapter.

When ATAC is started all the available instruments and adapters are displayed to the user. The user selects which of the instruments and adapters are loaded and initialized at the beginning of the program. This way same ATAC software can be used in different workstations regardless of what adapters, instruments and drivers that particular workstation has. Only the selected adapters and instruments are loaded into PC memory and initialized. This run-time loading of drivers was achieved by using factory design pattern that enabled construction of the whole HAL.

Software plug-ins for instruments and adapters can be created separately from the main ATAC code. Only linkage between a plug-in and the main code is that a plug-in class must inherit a generic instrument or adapter class defined in the main ATAC software. This way the main software remains untouched when adding support for new hardware.

The structure of HAL implemented in ATAC is shown in Figure 18. Not all methods and devices are depicted in the figure because the main idea is to present the structure of the HAL and not full documentation of implemented methods/VIs. The top level code is implemented in general level without specifying how individual tasks are implemented. The ASL (Application separation layer) is a software layer for abstracting adapter and instrument specific implementation of commands such as “Initialize Multimeter” and “Configure I2C” and providing generic interface for the top level. DSSP (Device-specific software plug-in) level classes handle the device specific implementations of commands such as “Initialize Multimeter”, “Set Current”, etc. that the ASL defines. When adding a new instrument or adapter to the system a new DSSP class has to be created for that device. The new DSSP has to implement some or all of the methods defined by ASL depending on what functionality the new device supports. The top level software does not see what the actual driver for the device is or what communication bus is used to communicate with the device.

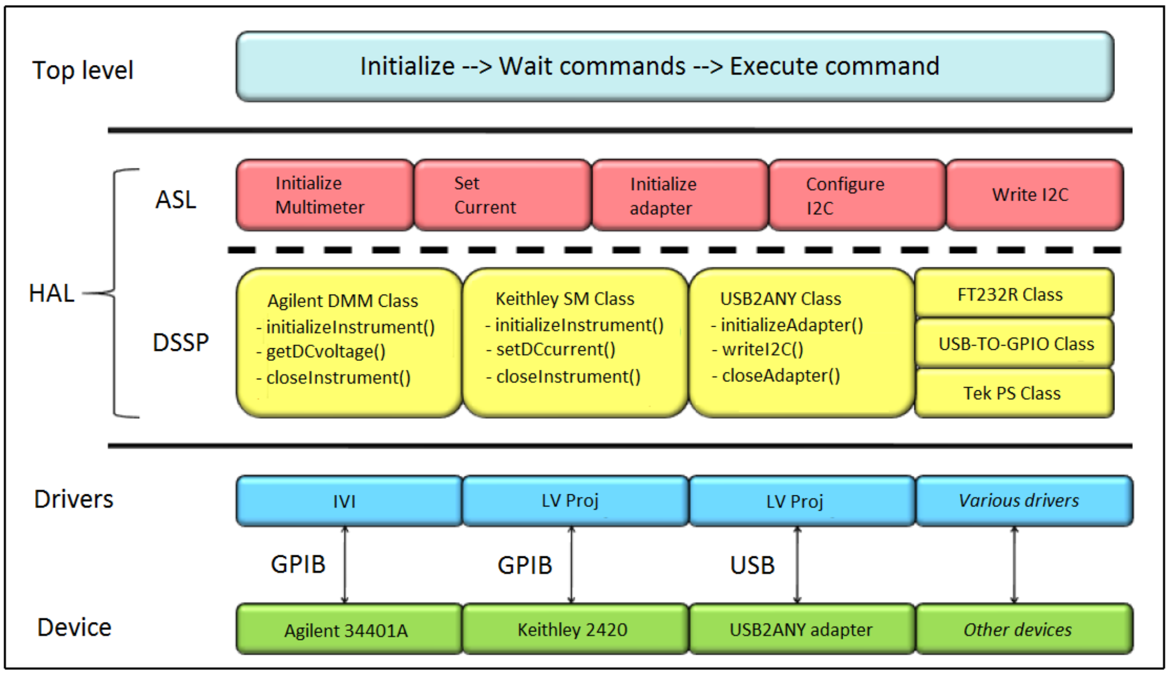

Top level
Initialize --> Wait commands --> Execute command
HAL
ASL
Initialize Multimeter
Set Current
Initialize adapter
Configure I2C
Write I2C
DSSP
Agilent DMM Class - initializeInstrument() - getDCvoltage() - closeInstrument()
Keithley SM Class - initializeInstrument() - setDCcurrent() - closeInstrument()
USB2ANY Class - initializeAdapter() - writeI2C() - closeAdapter()
FT232R Class
USB-TO-GPIO Class
Tek PS Class
Drivers
IVI
LV Proj
LV Proj
Various drivers
GPIB
GPIB
USB
Device
Agilent 34401A
Keithley 2420
USB2ANY adapter
Other devices

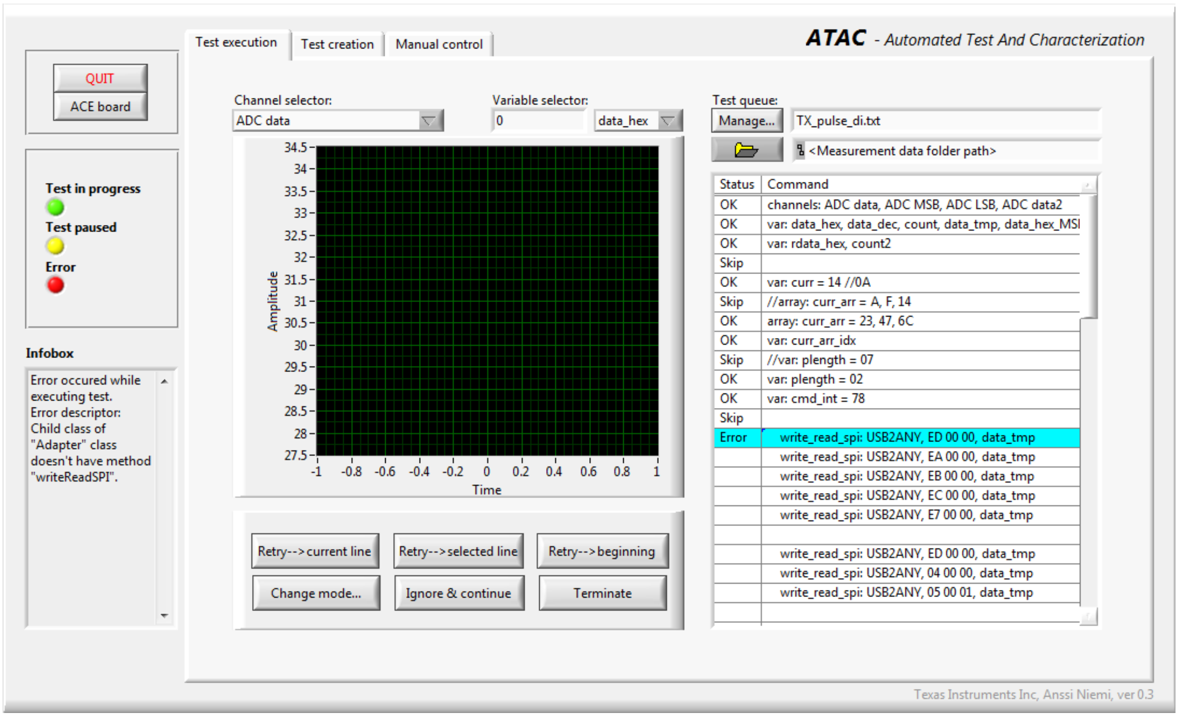
Figure 18, Hardware abstraction layer structure in ATAC.

5.4.3 Error handling

In LabVIEW exceptions are routed through the software with an error wire. Exceptions are generated when something goes wrong in the program execution. For example if the program tries to open a non-existent file. In ATAC exceptions are also created when something undesirable happens, for example if initialization of an object was unsuccessful or a device does not answer when spoken to. In most cases the exceptions are routed to top level with error wire where they are handled. In case of error the error indicator on main GUI is highlighted and the error description is displayed in the “Infobox”. The user can evaluate the nature and significance of the error and decide whether to continue or quit the program. If user decides to continue the error is cleared as well as the error indicator and “Infobox”.

If error is encountered while executing a test sequence the application enters a test execution error handling state presented in Figure 19. Here the user has different options as how to continue the execution or whether to terminate the test. In lab environment that has lots of different electrical machines EMI (electromagnetic interference) can cause signals to corrupt. While developing the ATAC software and running test scripts the I^2C communication failed occasionally without apparent reason.

A test sequence could take hours or days to execute so it is not convenient if the execution stops completely when encountering error. In many cases a test is left running alone while the user focuses on other tasks so it is important that the software can solve error states by itself without user interaction. This problem was solved by making the ATAC to retry a command three times before stopping and displaying the error to the user. This solved the most frequent error where I^{2}C communication failed at random intervals. Most likely the signal was occasionally corrupted by external interference and when retrying the message went through uncorrupted.



Error occured while executing test. Error descriptor: Child class of "Adapter" class doesn't have method "writeReadSPI".

Status	Command
OK	channels: ADC data, ADC MSB, ADC LSB, ADC data2
OK	var: data_hex, data_dec, count, data_tmp, data_hex_MSI
OK	var: rdata_hex, count2
Skip	
OK	var: curr = 14 //0A
Skip	//array: curr_arr = A, F, 14
OK	array: curr_arr = 23, 47, 6C
OK	var: curr_arr_idx
Skip	//var: plength = 07
OK	var: plength = 02
OK	var: cmd_int = 78
Skip	
Error	write_read_spi: USB2ANY, ED 00 00, data_tmp
	write_read_spi: USB2ANY, EA 00 00, data_tmp
	write_read_spi: USB2ANY, EB 00 00, data_tmp
	write_read_spi: USB2ANY, EC 00 00, data_tmp
	write_read_spi: USB2ANY, E7 00 00, data_tmp
	write_read_spi: USB2ANY, ED 00 00, data_tmp
	write_read_spi: USB2ANY, 04 00 00, data_tmp
	write_read_spi: USB2ANY, 05 00 01, data_tmp

Figure 19. Error handling when encountering error at test execution.

5.4.4 Flow of execution

The flow of program execution on the top level is described next using a flowchart and three state charts. In Figure 20 the general execution flow of main VI is presented. First the front panel properties of the main VI are initialized as well as blocks used for synchronization between parallel top level loops. After that the queues and software events are initialized in parallel to allow communication between the loops.

The communication between loops is designed and implemented by the principles of producer/consumer design pattern [14]. The top loop acts as a producer loop for the middle loop and sends commands to the middle loop through a queue. The middle loop acts as a consumer loop for the top loop and as a producer loop for the bottom loop and sends commands to the bottom loop through another queue. Bottom loop can send information with events to the top loop. The master/slave type of communication of producer/consumer design pattern is thus turned into a communication chain where every loop can send information to next loop. The top loop acts as an event handler for user interface events and software generated events. The middle loop is the main loop of the software and holds instance of a “Mode Controller” object that coordinates execution between tasks. The bottom loop handles test script execution.

After initializing the communication queues and events the loops are synchronized and started in parallel at the same time. The design of each loop is illustrated in Figure 21, Figure 22 and Figure 23 using state charts. When each loop has terminated (happens when user quits application) the communication queues and events are closed and application is shut down.

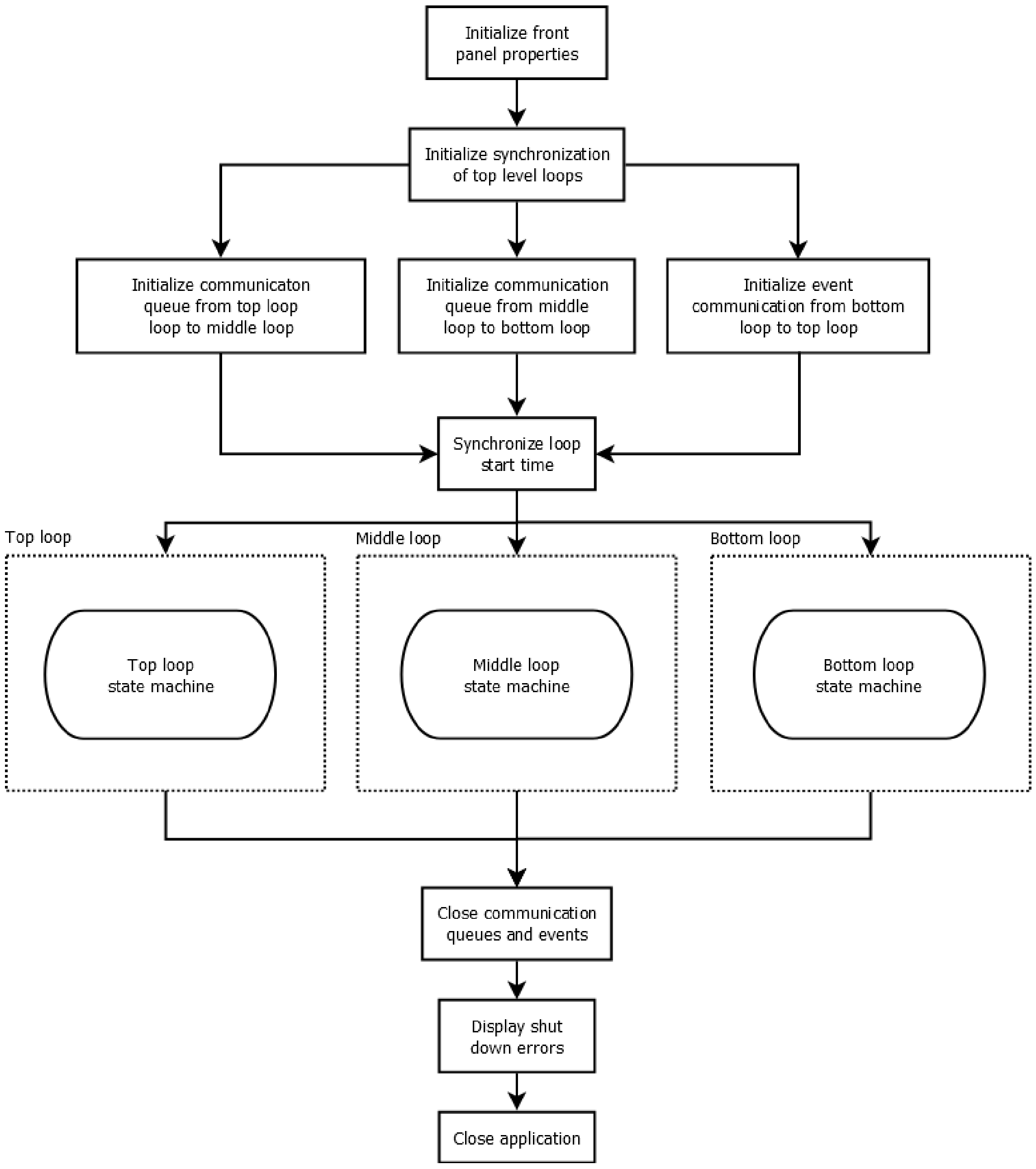

Initialize front panel properties
Initialize synchronization of top level loops
Initialize communicaton queue from top loop loop to middle loop
Initialize communication queue from middle loop to bottom loop
Initialize event communication from bottom loop to top loop
Synchronize loop start time
Top loop
Middle loop
Bottom loop
Top loop state machine
Middle loop state machine
Bottom loop state machine
Close communication queues and events
Display shut down errors
Close application

Figure 20. Top level program execution flowchart.

Figure 21 illustrates the top loop (event handler loop) execution in general level using state chart representation. The program execution waits for events and does not consume processor resources if no events are available. Most events are generated when user presses buttons or changes values of other controls on ATAC front panel. Every event has its own event handler which sends appropriate command to main loop for desired action. In most cases the top loop does not do anything else but to send instruction for middle loop when event is registered. This way the top loop stays always ready for registering events and the GUI does not freeze up. In Figure 21 all events other than quit button activity is considered to be included in the “Event registered” event. In practice every event causes a different command to be sent to the middle loop (main loop) but for the sake of simplicity the figure is drawn with a single “Send command to middle loop” state. After sending command possible errors encountered are displayed and cleared. The front panel GUI shows error description until user presses a button but for the program execution to continue appropriately the error is cleared on the block diagram. If quit button is pressed a quit command is sent to the middle loop and the top loop pauses to wait until other loops are ready to close. It is necessary to stop execution of all loops in a controlled manner before closing communication channels and hardware references to avoid errors and race conditions.

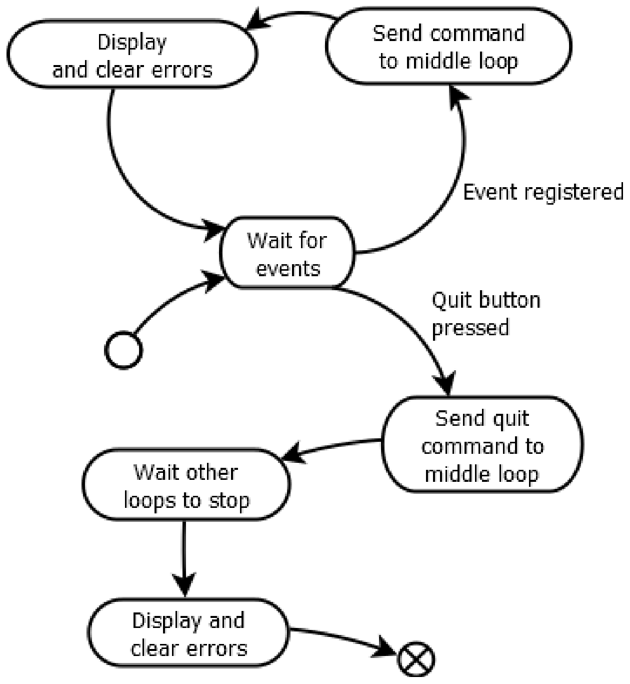


Figure 21. Top loop state machine. The loop handles all the events created either by user pressing buttons or other parts of software.

The execution of middle loop (main loop) is illustrated in Figure 22. The middle loop initializes the GUI and hardware when it is started. After every action the errors are checked and displayed if there are any. The main loop checks queue for commands from the top loop every 200ms. If no commands are received the mode controller status is updated to hold newest information about test execution state. The test execution state information is held at GUI indicators which are accessed by both middle loop and bottom loop. 200ms is a long time in respect to processor clock frequency so updating the mode controller status does not take much processor resources. Every command that does not involve executing a test file is handled in the middle loop. When test execution command is received the command is routed to the bottom loop if test execution can be initiated or continued. In case of receiving quit command the middle loop sends the quit command forward to the bottom loop and waits other loops to stop executing before exiting the loop.

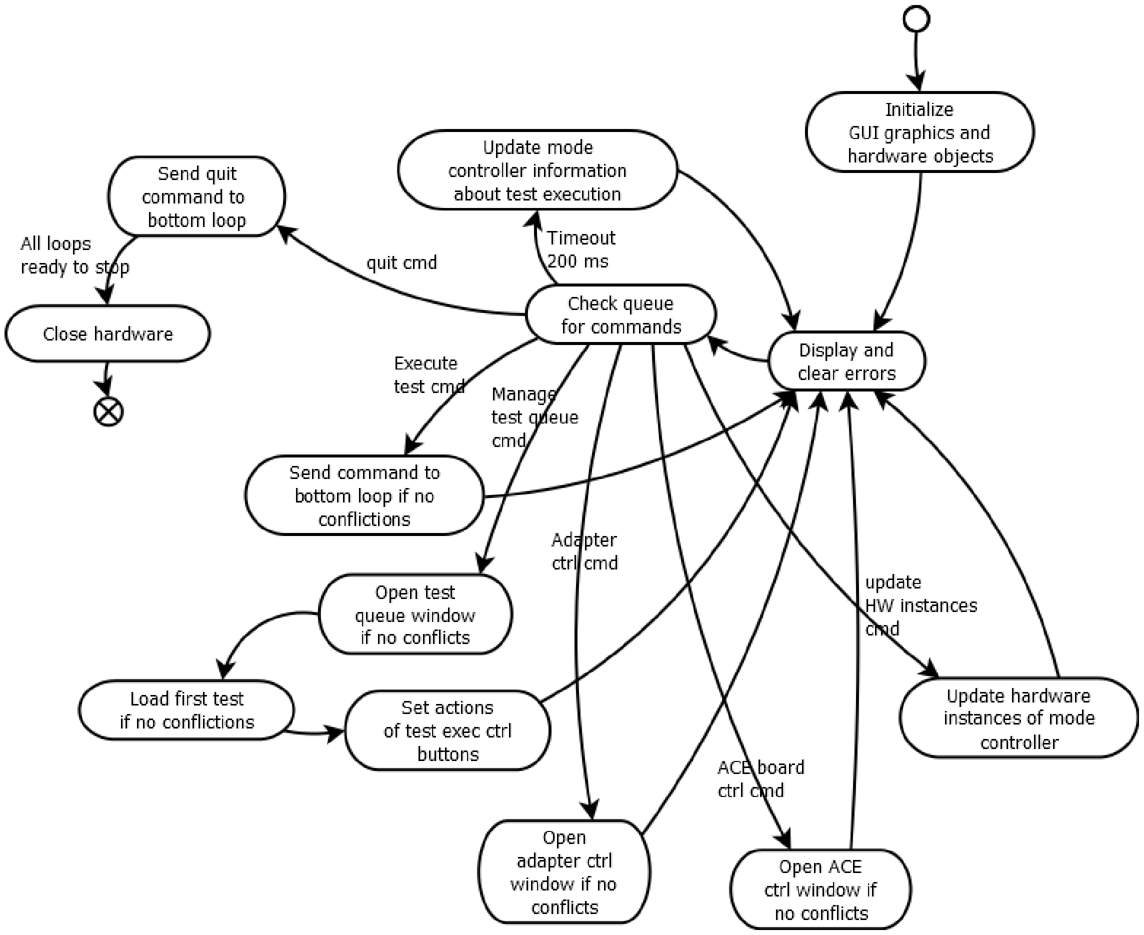

Initialize GUI graphics and hardware objects
Update mode controller information about test execution
Send quit command to bottom loop
All loops ready to stop
Close hardware
quit cmd
Timeout 200 ms
Check queue for commands
Display and clear errors
Execute test cmd
Manage test queue cmd
Send command to bottom loop if no conflictions
Adapter ctrl cmd
Open test queue window if no conflicts
update HW instances cmd
Load first test if no conflictions
Set actions of test exec ctrl buttons
Update hardware instances of mode controller
ACE board ctrl cmd
Open adapter ctrl window if no conflicts
Open ACE ctrl window if no conflicts

Figure 22. Middle loop state machine. This is the main loop of the software that handles everything else but test execution.

The bottom loop (test execution loop) illustrated in Figure 23 using state chart is the loop that handles test script execution. Initially the loop waits for commands from the middle loop and does not take processor resources. When an instruction is received an action for that instruction is performed. If execution mode is set to be continuous, meaning the test is executed automatically from start to finish, the program will check for incoming instructions only for 1 ms period until continuing to execute next line. The smaller the period for waiting commands is the faster the test execution is which is critical in large time consuming tests. After executing a line all hardware object instances are sent to top loop using an event to provide up-to-date information about the hardware instances. Top loop routes hardware instances to the middle loop that holds “Mode controller” object instance. Thus after every executed line the “Mode controller” object has newest information about the hardware objects in the system. Sending of hardware objects to top loop is considered to be included in “Execute next line of test” state in the Figure 23. After executing a line possible errors are handled. Error handling in the bottom loop differs from the top and middle loop error handling. When errors are encountered in test execution user is given different choices to handle the error and continue as presented earlier in this paper.

If quit command is received the graphics related to test execution are cleared and the program pauses to wait that every loop has stopped. When all three loops are stopped and waiting for other loops the execution can continue.

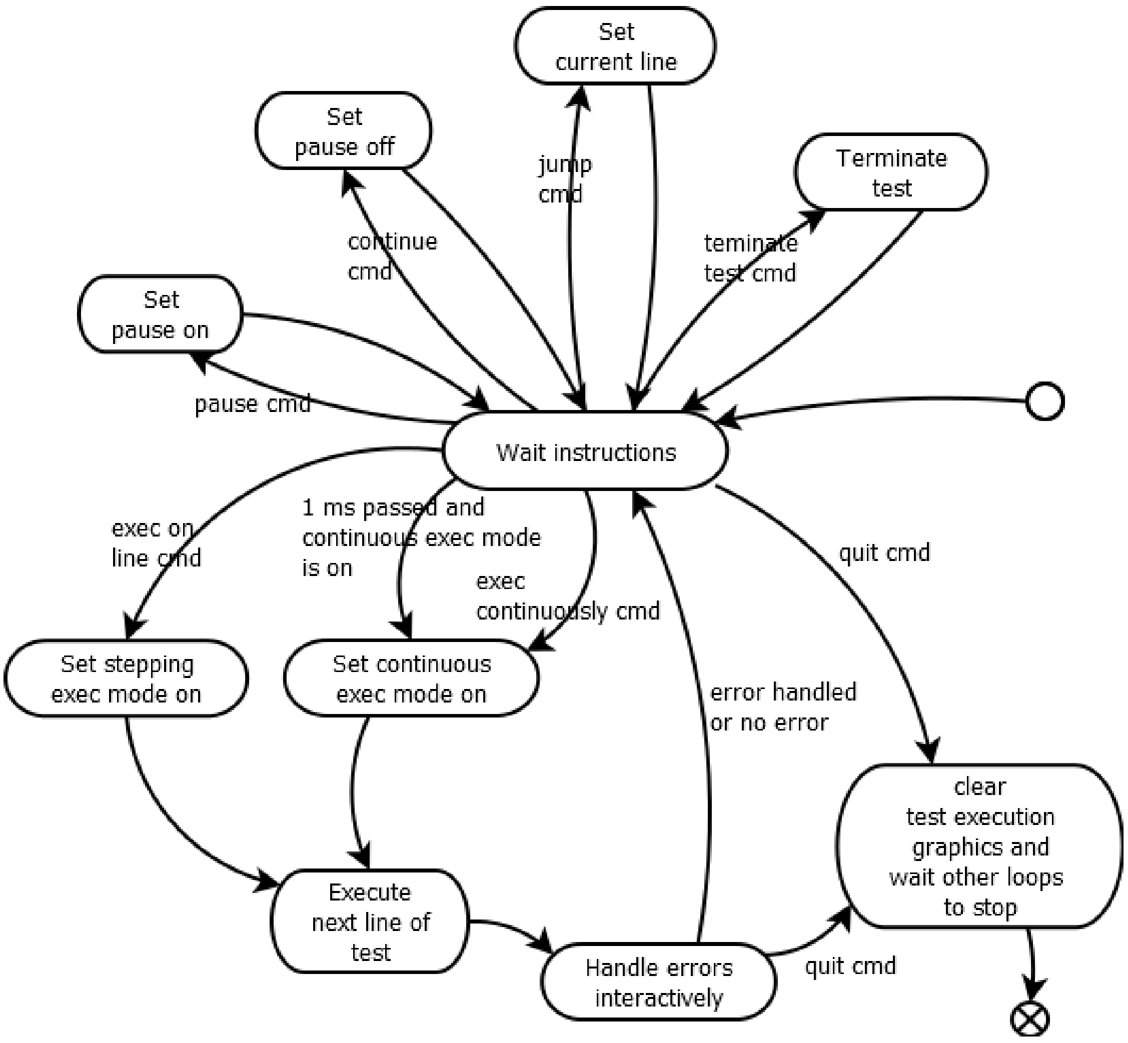

Set current line
Set pause off
jump cmd
Terminate test
continue cmd
teminate test cmd
Set pause on
pause cmd
Wait instructions
exec on line cmd
1 ms passed and continuous exec mode is on
exec continuously cmd
quit cmd
Set stepping exec mode on
Set continuous exec mode on
error handled or no error
clear test execution graphics and wait other loops to stop
Execute next line of test
Handle errors interactively
quit cmd

Figure 23. Bottom loop state machine. This loop handles the test execution.

Long tests might take hours or days to complete so it’s efficient if user can use the software for different tasks while a test sequence is being executed. To achieve this the execution of tests is run in parallel to other tasks. In future the software could be enhanced by adding test creation window or measurement post processing capabilities. User could create tests or process measurement data with the same application and at the same time test sequences are executed. Parallel execution of tests and other tasks sets certain challenges that need to be assessed carefully when designing the software. For example controlling of adapters and instruments has to be blocked from the user when a test sequence is being executed. If test script uses instruments and adapters as is commonly the case the user interaction with an adapter or an instrument could cause the script to fail or to produce non-reasonable results.

Parallel execution of tests is implemented by a parallel top level loop. The top level structure follows producer/consumer design pattern [14] but adds extra loop to handle test execution. The master/slave type of communication of producer/consumer design pattern is enhanced and the resulting pattern has a chain type of communication flow. The event handler loop (top loop in Figure 24) functions as a producer for the main loop (middle loop in Figure 24). The middle loop acts as consumer for the event handler loop and producer for the test execution loop (bottom loop in Figure 24). The test execution loop handles the execution of tests and updates “Test execution” window on ATAC front panel accordingly. The test execution loop sends events to the event handler thus acting as a producer for the event handler loop.

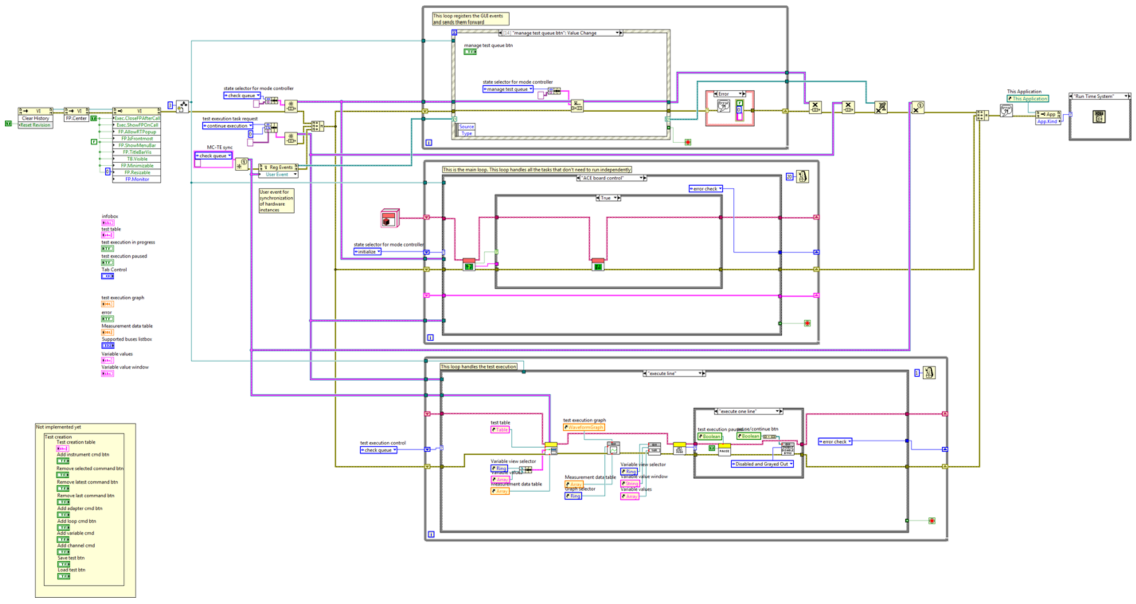


Figure 24. Block diagram of top level VI.

5.5 Software testing

5.5.1 Unit testing

To minimize errors in the code every VI was tested individually with different parameters before integrating the VI into other VI's. This type of unit testing makes debugging easy because only small part of code is tested at a time. When connecting individual VI's together the complexity of the code increases and finding errors becomes more difficult.

LabVIEW makes unit testing very easy because no external test program is needed to run a VI with different input parameters. The VI front panel controls can be set by user to execute the VI with different input parameters. Unit testing works well in most cases but if a VI depends on complex objects it can be very time consuming to manually set the input parameters from the front panel of the VI. In these cases the VI was integrated into the main software without pre-testing and the main software was used to test the new VI by running the main program to certain states with different parameters.

Viewing the state and values of different variables and wires is easy in LabVIEW because of the debugging tools LabVIEW provides. User can place probes on wires to view all the data that flows through the code. In addition user can place breakpoints to specific parts of the code and make the program execution to pause at that point until instructed to continue. The program can be run in normal execution mode or by stepping instructions one by one. These debugging capabilities made the debugging fast and efficient and errors in the code were easily discovered. Not all the possible states and input combinations of a VI could be tested because the number of possible combinations of input parameters is very large at times. In these cases the VI was tested with input values that had most chances to generate incorrect behavior.

Below is a unit testing example of a VI called "buildInstrumentObjects" that creates instrument objects based on instrument names that are given to the VI as input parameters. This example shows the principle how VIs are tested in general before integrating them into other VIs. The front panel of the VI is shown in Figure 25. Input parameters for the VI are placed on left side of the front panel and their meaning is the following:

- Instrument Group in
  - Object representing all of the instruments. Provides methods/VIs for easy picking of specific instruments from the whole group.

- XML File Name
  - A string that tells the VI what is the name of the configuration file that holds information about all instrument names, types and paths of classes that implement instrument functionality.

- Instruments to build
  - Array containing names of instruments that are built and loaded into memory. The VI goes through the specified XML file and loads all instrument plug-in classes that match to specified instrument names.

- error in (no error)
  - Error information to the VI. The VI is not executed if error status is true.

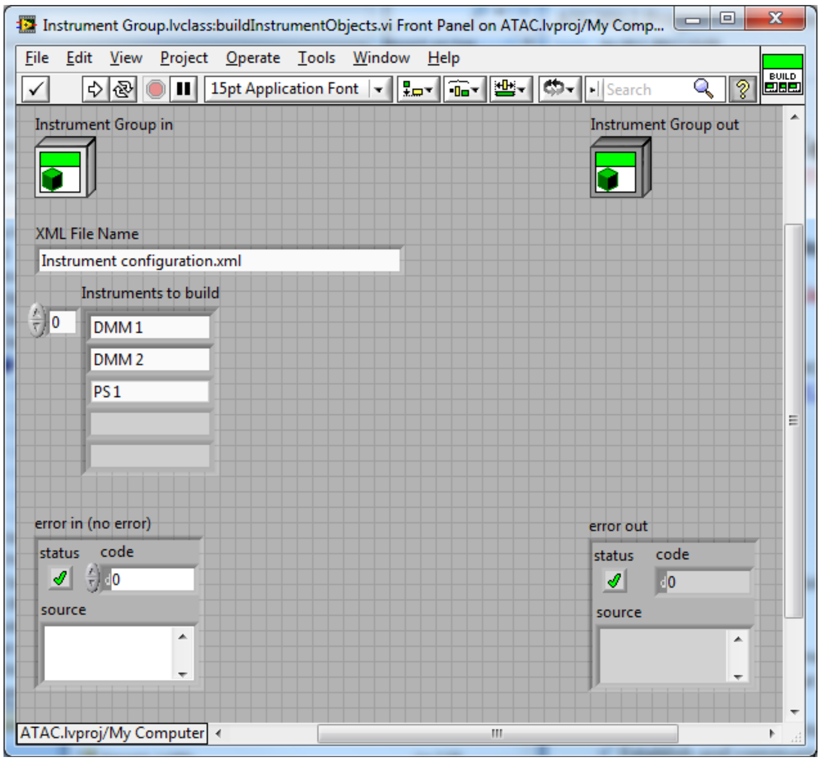


Figure 25. Front panel of a VI called "buildInstrumentObjects".

Outputs of the VI are updated instrument group object and error information. The front panel of the VI can be used to examine whether the VI produces error signals to "error out" indicator but not much more can be seen by examining the front panel. This is why the functionality of the VI was tested by examining the block diagram execution with different input values. Block diagram of the "buildInstrumentObjects" VI is shown in Figure 26. The yellow arrows are added to point to debug tools LabVIEW provides. Two probes are placed on wires to show user what information flows through the wires when the VI is executed. The one break point (red circle) is added to pause the program execution after a single instrument plug-in has been loaded into memory. When the execution is paused by the break point user can view the information of the probe "3" that holds information about the newly loaded instrument plug-in class. This way user can track the execution of the program and tell whether the VI is working as it should. After all instrument plug-in classes and instrument drivers have been loaded into PC memory the program execution reaches probe "1". This probe displays an array that holds all the newly created instrument objects. The use of probes and break points makes the debugging fast and efficient because large part of code can be tested simultaneously by adding more probes to the block diagram.

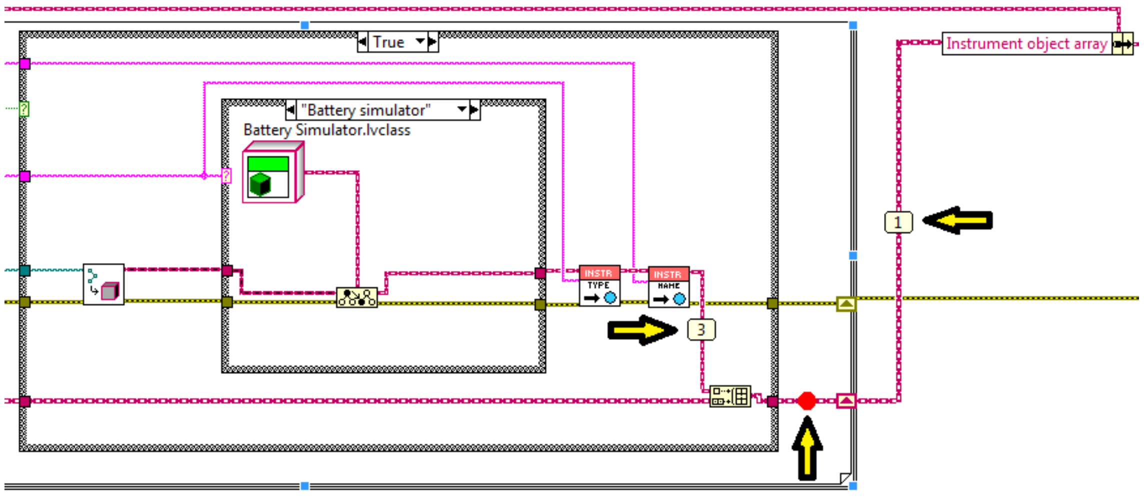


Figure 26. Part of a VI block diagram that illustrates the use of probes and break points.

5.5.2 System testing

When connecting individual VIs together the communication and compatibility of different VIs needs to be tested for errors. Proper unit testing reduces the possibility of encountering errors when connecting different VIs together but not all errors can be found at unit testing. For example different VIs might work properly on their own but when connecting an output of a VI to be input to another VI the interfaces (connector panes) of the two VIs might not match.

System testing is used to test whether different VIs work well together. The ATAC software has deep hierarchy of VIs so system testing was performed at many different levels. Final system testing was done at top level where the software was used like it would be used in target environment. In addition many different unusual cases were tested to find all possible errors in the program. For example user could press buttons while program is processing events for previous user action and the program must work reliably also in these situations. Other possible cases for unexpected errors are for example if user tries to jump to a line that does not exist or instructs the program to do something else that was not meant to be done. The system testing was performed in close co-operation with end users that used the software for two months and gave feedback about the features and bugs they had encountered.

6 Results and discussion

The overall successfulness of the ATAC development project is assessed in this chapter. An example case of an evaluation test performed to a real ASIC circuit is presented and the results are discussed. The reader is given a general view how a single evaluation test is performed using ATAC and what kind of results the ATAC produces. The data produced by ATAC is compared to manually gathered measurement data and the use of ATAC and automated evaluation environment is assessed based on the results. The usability of ATAC is also discussed based on feedback gathered from the end users.

6.1 Meeting the specification

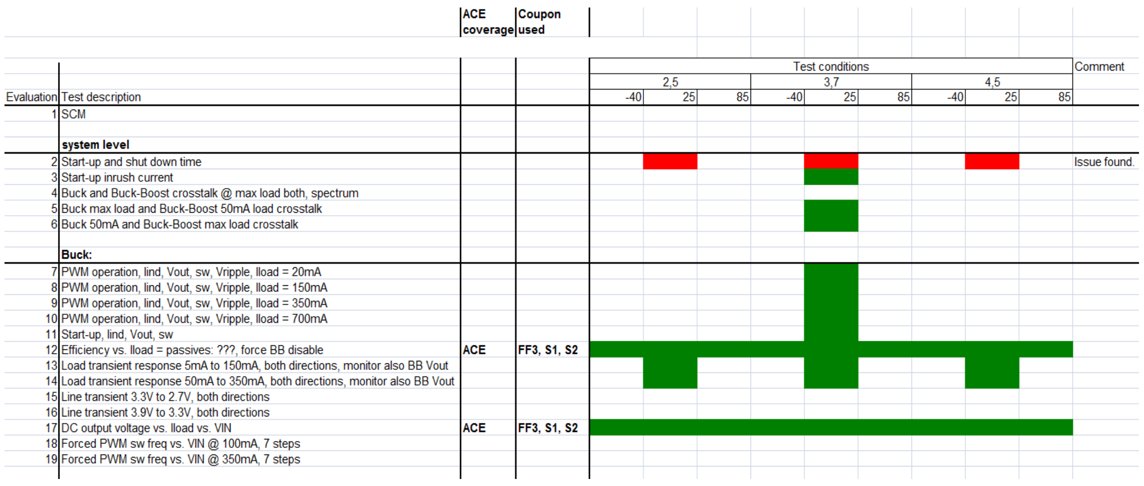
One way of determining successfulness of a software project is by comparing the final software to the specification and requirements. The requirements for ATAC software defined in chapter 5.2 were implemented entirely and some extra features such as ACE board instrument mapping were added in addition. The specification for the software evolved through the software life cycle and features were gradually added to specification.

6.2 ASIC evaluation measurement results

The main objective of ATAC software was to be able to control the ASIC under test and to retrieve measurements from instruments and store them to measurement files automatically. To validate correctness of the measurement data the retrieved measurements were compared against simulation data and measurement data that was gathered by hand using separate measurement setup. If the DUT works as expected the measurement results are similar to simulation results. The general operation of the automated characterization system is thus easily validated. Still there is a chance that the characterization system hardware adds resistance, inductance, capacitance and other non-idealities to the measurement results and the results might thus be worse than what the DUT really is. To test the quality of the automated evaluation system hardware a comparison to manually made measurements was performed with different tests.

If the DUT does not work as simulated the validation of measurement results by comparing to simulation is not so straightforward. In these cases doing the measurements manually with different measurement setup gives good reference whether the bad measurements were due to bad hardware setup or malfunctioning DUT.

The application was successfully used to evaluate three different ASICs. The results had good quality compared to manually made measurements (no difference) and in line with simulations. The automatic characterization system proved to be efficient way to get large amount of measurement data from the DUT. Before automated system the measurements were done by hand. User had to manually connect instruments to different pins of DUT and use unorthodox software applications for reading the measurements to PC. The measurement results were visible to user only after the whole test was executed because the measurements were stored in the instruments’ memory through the execution of a test. With ATAC the time spent creating tests reduced due to easier command syntax. The debugging of tests was also more efficient because user could see the measurements in real-time at the PC screen and could pause and jump in the test execution. The automation made possible to use large number of measurement points and to sweep over many different parameters such as supply voltages, load currents and temperature. The resulting measurement data set provided essential information about the DUT in multiple different operation points which would have been impossible to achieve by manually doing measurements. By executing different test script files consecutively in automatic fashion the resulting data set could easily have over thousand data points. Measuring this number of data by hand would have not been possible in reasonable time and effort. An example of evaluation measurements performed for a real ASIC is presented below starting from the initial evaluation plan and ending in measured results and graphs. The characterization is started by defining evaluation plan which is presented in Figure 27. Initially the right side of the plan is empty and is filled with green and red boxes when that specific test has been performed.

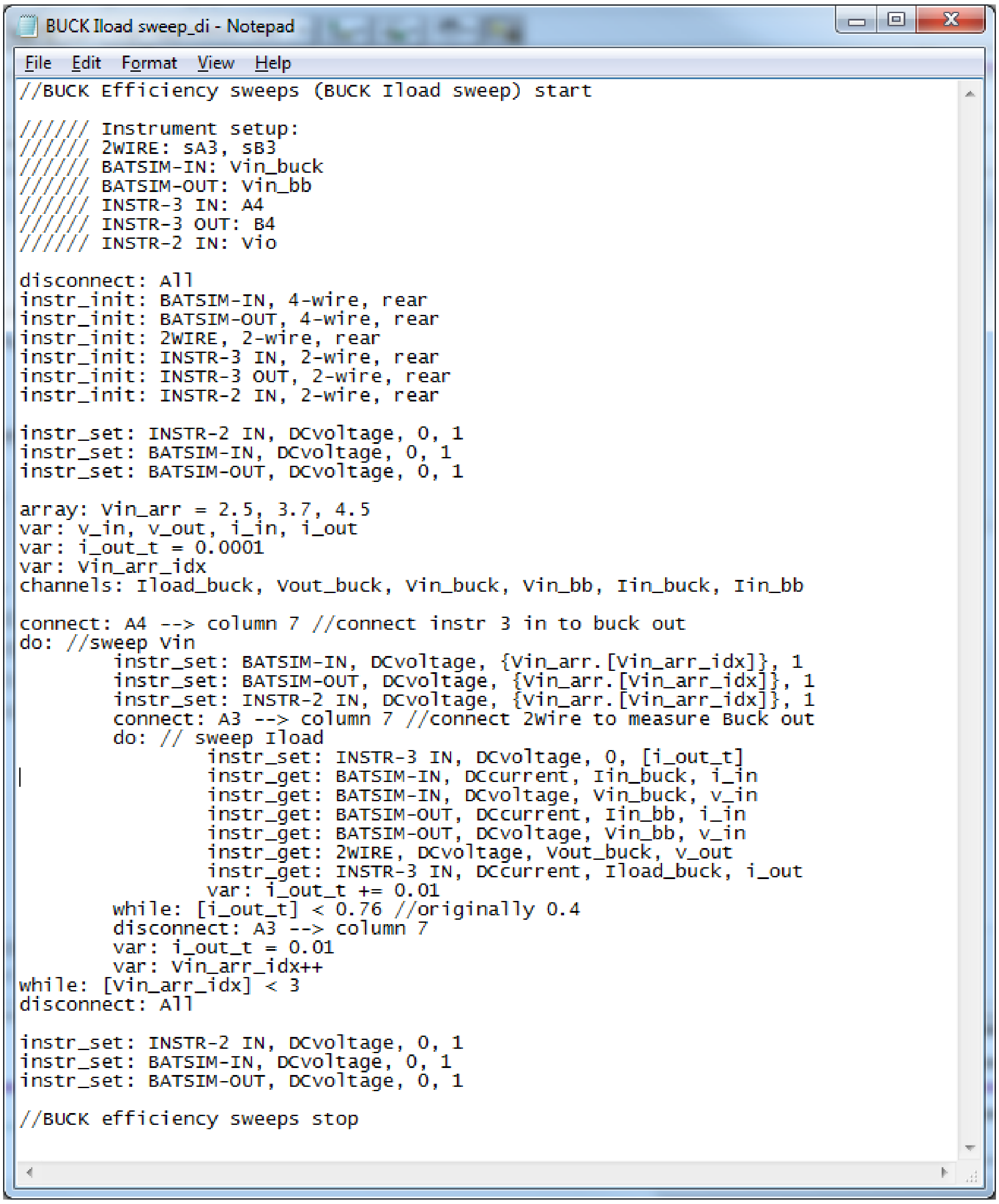


		ACE coverage	Coupon used	Test conditions									Comment
				2,5			3,7			4,5			
Evaluation	Test description			-40	25	85	-40	25	85	-40	25	85	
1	SCM												
	system level												
2	Start-up and shut down time				🟥			🟥			🟥		Issue found.
3	Start-up inrush current							🟩					
4	Buck and Buck-Boost crosstalk @ max load both, spectrum												
5	Buck max load and Buck-Boost 50mA load crosstalk							🟩					
6	Buck 50mA and Buck-Boost max load crosstalk							🟩					
	Buck:												
7	PWM operation, Iind, Vout, sw, Vripple, Iload = 20mA							🟩					
8	PWM operation, Iind, Vout, sw, Vripple, Iload = 150mA							🟩					
9	PWM operation, Iind, Vout, sw, Vripple, Iload = 350mA							🟩					
10	PWM operation, Iind, Vout, sw, Vripple, Iload = 700mA							🟩					
11	Start-up, Iind, Vout, sw							🟩					
12	Efficiency vs. Iload = passives: ???, force BB disable	**ACE**	**FF3, S1, S2**	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
13	Load transient response 5mA to 150mA, both directions, monitor also BB Vout				🟩			🟩			🟩		
14	Load transient response 50mA to 350mA, both directions, monitor also BB Vout				🟩			🟩			🟩		
15	Line transient 3.3V to 2.7V, both directions												
16	Line transient 3.9V to 3.3V, both directions												
17	DC output voltage vs. Iload vs. VIN	**ACE**	**FF3, S1, S2**	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
18	Forced PWM sw freq vs. VIN @ 100mA, 7 steps												
19	Forced PWM sw freq vs. VIN @ 350mA, 7 steps												

Figure 27. Evaluation plan for a specific ASIC.

As can be seen from Figure 27 evaluation plan all the tests need to be performed with different supply voltages and in different temperatures. A single test already involves sweeping over some parameters, so in addition sweeping over supply voltage and temperature results in a very large measurement data set. The ACE coverage column in Figure 27 tells whether the specific test is to be done with the automated evaluation setup or manually. The tests that require oscilloscope are done manually and others with automatic setup using ATAC. When a test script is written for ATAC it is very easy to sweep over different parameters such as supply voltage and load current because these add only additional loops to the test script that is otherwise functional. As can be seen from Figure 27 evaluation plan, manual measurements were done only with one supply voltage and in one temperature at the time the evaluation plan image was added in this paper. This indicates that it is very time consuming to perform tests manually by hand. The tests that were done with ATAC were performed for every supply voltage and every temperature. Although for now ATAC does not support temperature control it is very fast to manually set the temperature of an environmental chamber where DUT is placed to different temperatures between a set of tests.

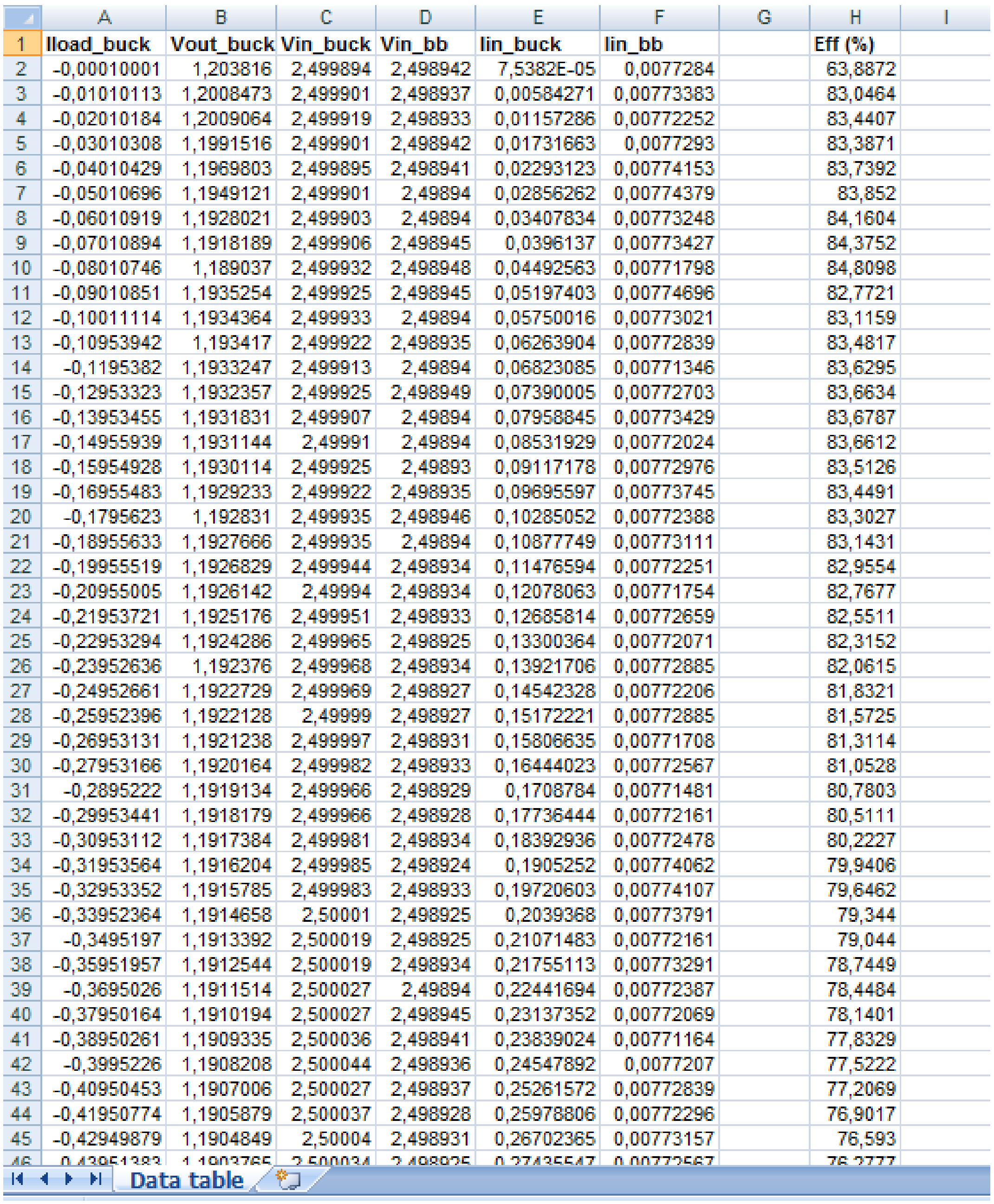
The next step after creating evaluation plan is to create test scripts for ATAC. Below in Figure 28 is presented a script that performed a test “DC output voltage vs. Iload vs. VIN” that was defined in the evaluation plan.



```
//BUCK Efficiency sweeps (BUCK Iload sweep) start

////// Instrument setup:
////// 2WIRE: SA3, SB3
////// BATSIM-IN: Vin_buck
////// BATSIM-OUT: Vin_bb
////// INSTR-3 IN: A4
////// INSTR-3 OUT: B4
////// INSTR-2 IN: Vio

disconnect: All
instr_init: BATSIM-IN, 4-wire, rear
instr_init: BATSIM-OUT, 4-wire, rear
instr_init: 2WIRE, 2-wire, rear
instr_init: INSTR-3 IN, 2-wire, rear
instr_init: INSTR-3 OUT, 2-wire, rear
instr_init: INSTR-2 IN, 2-wire, rear

instr_set: INSTR-2 IN, DCvoltage, 0, 1
instr_set: BATSIM-IN, DCvoltage, 0, 1
instr_set: BATSIM-OUT, DCvoltage, 0, 1

array: Vin_arr = 2.5, 3.7, 4.5
var: v_in, v_out, i_in, i_out
var: i_out_t = 0.0001
var: Vin_arr_idx
channels: Iload_buck, Vout_buck, Vin_buck, Vin_bb, Iin_buck, Iin_bb

connect: A4 --> column 7 //connect instr 3 in to buck out
do: //sweep Vin
        instr_set: BATSIM-IN, DCvoltage, {Vin_arr.[Vin_arr_idx]}, 1
        instr_set: BATSIM-OUT, DCvoltage, {Vin_arr.[Vin_arr_idx]}, 1
        instr_set: INSTR-2 IN, DCvoltage, {Vin_arr.[Vin_arr_idx]}, 1
        connect: A3 --> column 7 //connect 2Wire to measure Buck out
        do: // sweep Iload
                instr_set: INSTR-3 IN, DCvoltage, 0, [i_out_t]
                instr_get: BATSIM-IN, DCcurrent, Iin_buck, i_in
                instr_get: BATSIM-IN, DCvoltage, Vin_buck, v_in
                instr_get: BATSIM-OUT, DCcurrent, Iin_bb, i_in
                instr_get: BATSIM-OUT, DCvoltage, Vin_bb, v_in
                instr_get: 2WIRE, DCvoltage, Vout_buck, v_out
                instr_get: INSTR-3 IN, DCcurrent, Iload_buck, i_out
                var: i_out_t += 0.01
        while: [i_out_t] < 0.76 //originally 0.4
        disconnect: A3 --> column 7
        var: i_out_t = 0.01
        var: Vin_arr_idx++
while: [Vin_arr_idx] < 3
disconnect: All

instr_set: INSTR-2 IN, DCvoltage, 0, 1
instr_set: BATSIM-IN, DCvoltage, 0, 1
instr_set: BATSIM-OUT, DCvoltage, 0, 1

//BUCK efficiency sweeps stop
```

Figure 28. Test script for DC output voltage vs. Iload vs. VIN.

The test script is loaded into ATAC and executed. The measurement results are automatically stored to a user specified TDMS file. TDMS is a file format developed by National Instruments and is designed for storing measurement data. The TDMS file can be read into Excel for post processing using a free TDM Excel Add-in [28]. The measurement data seen in Excel is shown in Figure 29. There are five times more measurement data in the table that is seen in the Figure 29 but the picture gives a basic view of what the output data of the ATAC looks like. In this case the "Eff (%)" column is added afterwards in Excel but it could be automatically generated by ATAC also if the test script was modified to calculate efficiency at real-time when the test is being executed. Other columns in Figure 29 are automatically generated by ATAC.



	A	B	C	D	E	F	G	H	I
1	Iload_buck	Vout_buck	Vin_buck	Vin_bb	Iin_buck	Iin_bb		Eff (%)	
2	-0,00010001	1,203816	2,499894	2,498942	7,5382E-05	0,0077284		63,8872	
3	-0,01010113	1,2008473	2,499901	2,498937	0,00584271	0,00773383		83,0464	
4	-0,02010184	1,2009064	2,499919	2,498933	0,01157286	0,00772252		83,4407	
5	-0,03010308	1,1991516	2,499901	2,498942	0,01731663	0,0077293		83,3871	
6	-0,04010429	1,1969803	2,499895	2,498941	0,02293123	0,00774153		83,7392	
7	-0,05010696	1,1949121	2,499901	2,49894	0,02856262	0,00774379		83,852	
8	-0,06010919	1,1928021	2,499903	2,49894	0,03407834	0,00773248		84,1604	
9	-0,07010894	1,1918189	2,499906	2,498945	0,0396137	0,00773427		84,3752	
10	-0,08010746	1,189037	2,499932	2,498948	0,04492563	0,00771798		84,8098	
11	-0,09010851	1,1935254	2,499925	2,498945	0,05197403	0,00774696		82,7721	
12	-0,10011114	1,1934364	2,499933	2,49894	0,05750016	0,00773021		83,1159	
13	-0,10953942	1,193417	2,499922	2,498935	0,06263904	0,00772839		83,4817	
14	-0,1195382	1,1933247	2,499913	2,49894	0,06823085	0,00771346		83,6295	
15	-0,12953323	1,1932357	2,499925	2,498949	0,07390005	0,00772703		83,6634	
16	-0,13953455	1,1931831	2,499907	2,49894	0,07958845	0,00773429		83,6787	
17	-0,14955939	1,1931144	2,49991	2,49894	0,08531929	0,00772024		83,6612	
18	-0,15954928	1,1930114	2,499925	2,49893	0,09117178	0,00772976		83,5126	
19	-0,16955483	1,1929233	2,499922	2,498935	0,09695597	0,00773745		83,4491	
20	-0,1795623	1,192831	2,499935	2,498946	0,10285052	0,00772388		83,3027	
21	-0,18955633	1,1927666	2,499935	2,49894	0,10877749	0,00773111		83,1431	
22	-0,19955519	1,1926829	2,499944	2,498934	0,11476594	0,00772251		82,9554	
23	-0,20955005	1,1926142	2,49994	2,498934	0,12078063	0,00771754		82,7677	
24	-0,21953721	1,1925176	2,499951	2,498933	0,12685814	0,00772659		82,5511	
25	-0,22953294	1,1924286	2,499965	2,498925	0,13300364	0,00772071		82,3152	
26	-0,23952636	1,192376	2,499968	2,498934	0,13921706	0,00772885		82,0615	
27	-0,24952661	1,1922729	2,499969	2,498927	0,14542328	0,00772206		81,8321	
28	-0,25952396	1,1922128	2,49999	2,498927	0,15172221	0,00772885		81,5725	
29	-0,26953131	1,1921238	2,499997	2,498931	0,15806635	0,00771708		81,3114	
30	-0,27953166	1,1920164	2,499982	2,498933	0,16444023	0,00772567		81,0528	
31	-0,2895222	1,1919134	2,499966	2,498929	0,1708784	0,00771481		80,7803	
32	-0,29953441	1,1918179	2,499966	2,498928	0,17736444	0,00772161		80,5111	
33	-0,30953112	1,1917384	2,499981	2,498934	0,18392936	0,00772478		80,2227	
34	-0,31953564	1,1916204	2,499985	2,498924	0,1905252	0,00774062		79,9406	
35	-0,32953352	1,1915785	2,499983	2,498933	0,19720603	0,00774107		79,6462	
36	-0,33952364	1,1914658	2,50001	2,498925	0,2039368	0,00773791		79,344	
37	-0,3495197	1,1913392	2,500019	2,498925	0,21071483	0,00772161		79,044	
38	-0,35951957	1,1912544	2,500019	2,498934	0,21755113	0,00773291		78,7449	
39	-0,3695026	1,1911514	2,500027	2,49894	0,22441694	0,00772387		78,4484	
40	-0,37950164	1,1910194	2,500027	2,498945	0,23137352	0,00772069		78,1401	
41	-0,38950261	1,1909335	2,500036	2,498941	0,23839024	0,00771164		77,8329	
42	-0,3995226	1,1908208	2,500044	2,498936	0,24547892	0,0077207		77,5222	
43	-0,40950453	1,1907006	2,500027	2,498937	0,25261572	0,00772839		77,2069	
44	-0,41950774	1,1905879	2,500037	2,498928	0,25978806	0,00772296		76,9017	
45	-0,42949879	1,1904849	2,50004	2,498931	0,26702365	0,00773157		76,593	
46	-0,43951383	1,1903765	2,500034	2,498925	0,27435547	0,00772567		76,2777	

Figure 29. Measurement data generated by ATAC loaded into Excel.

The next step is to post process the measurement data. The data should be converted to a graph for easy examination and desired key parameters should be generated from the data. This post processing is done by hand using Excel or other similar tool. Figure 30 shows a graph that visualizes the measurement data seen in Figure 29. The graph was constructed by hand using tools of Excel. X-axis of the graph is the load current from

DUT to load. The values have minus sign because the current is towards the source meter that was used to measure the current. Y-axis represents the overall power efficiency of the DUT in percentage. The efficiency is calculated using formula:

$$Power\ efficiency\ (\%) = \frac{P_{out}}{P_{in}} \times 100\% = \frac{U_{out}\, I_{out}}{U_{in}\, I_{in}} \times 100\%. \qquad (1)$$

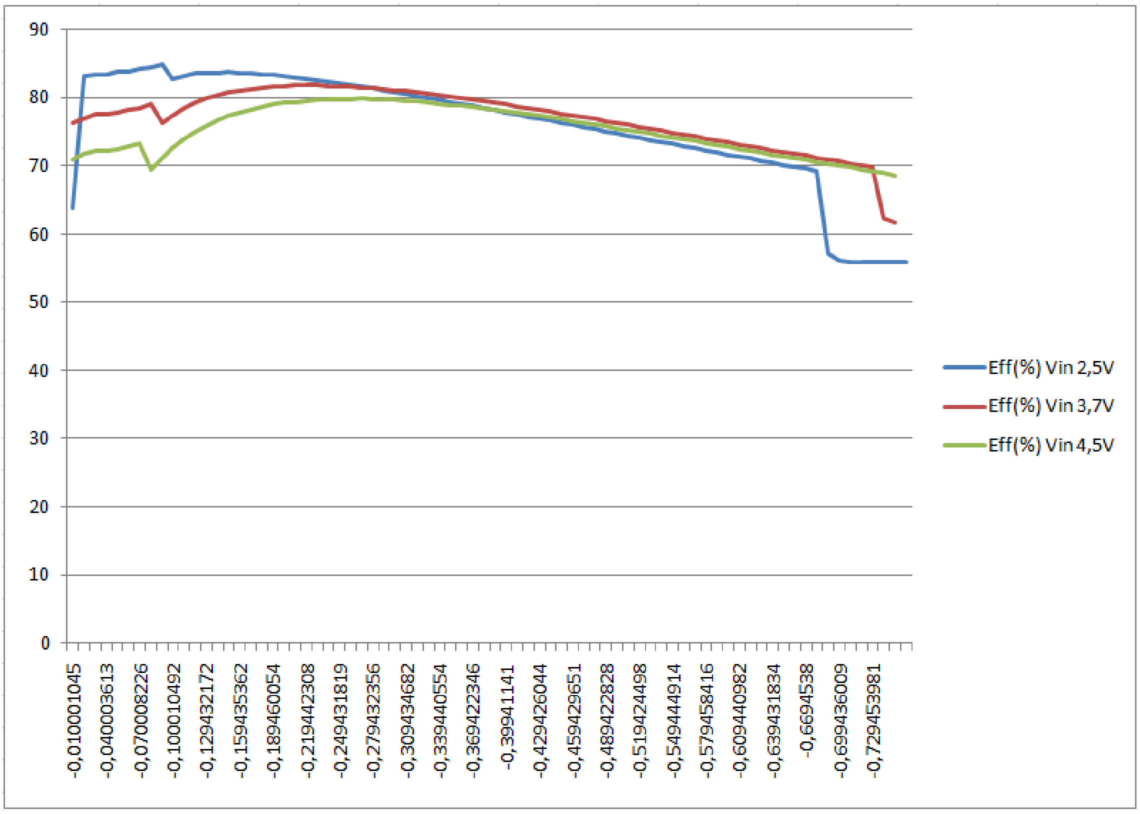


Figure 30. Efficiency with different supply voltages and load currents. Y-axis = efficiency (%), X-axis = load current.

6.3 Usability and flexibility

ATAC was used several months by end users after the development had reached stage where real characterization tests could be executed. User feedback was gathered continuously through this time for future improvements. The GUI of the software was considered successful as being simple and easy to use, yet providing enough customizable parameters. The debugging capabilities (being able to pause a test and manually sending commands) were also considered to be useful and well implemented. The ATAC was able to handle three different kinds of ASIC circuits that were evaluated through the project. New features and support for new adapters and instruments were successfully implemented through the development process due to hardware abstraction layer and modular nature of the software.

Negative feedback was received about the handling of numbers in test script. Because the software lacked unified means of handling numbers, all commands were not directly compatible with each others. Some commands handled numbers with a prefix (0x, 0b, 0d) and others without the prefix. The prefix explicitly tells the ATAC command handler that a character is to be handled as a certain type of number. This issue could be fixed in the future by modifying all commands to take number parameters with a prefix. Another negative feature was that when debugging a test sequence user makes constantly small modifications to the script. However loading the script into ATAC requires multiple interactive actions from the user which becomes annoying after loading the slightly altered test into the program many times. This could be fixed by adding a reload button to the GUI which would reload the current test from file system to ATAC.

Overall user experience gathered from the end users was positive and the ATAC was considered to be a better solution than the previous test sequencer software that was used before ATAC. The fact that user can track the test execution and view measurements at real time combined with the debugging capabilities was the main improvement compared to previous software solutions.

7 Future improvements

Possible future improvements to ATAC are presented in this chapter. Some improvements would extend the features of the ATAC beyond its original specification where as other improvements would be smaller changes that would affect and improve the current features of the ATAC.

7.1 Post processing of characterization data

Post processing the data ATAC generates turned out to be laborious due to large number of data points and data files generated. When the number of Excel data files increased it became hard to manage all the data effectively. Third party data mining software was considered but suitable compatible software was not found. Adding data post processing capabilities into ATAC would provide one integrated solution that could handle all the data handling. Being LabVIEW application ATAC could also be easily made to read data from TDMS files which is a file format developed by the same company as LabVIEW. LabVIEW also provides graph frameworks so measurements could easily be converted into graph format.

7.2 Synthesis of circuit simulators and ATAC

When a new ASIC circuit is designed its operation is simulated by a design engineer. After design phase the circuit is sent for manufacturing. The manufacturing of an ASIC device can take a few months and at that time the engineer cannot proceed with the project. Writing test scripts for ASIC beforehand could be done but it is reasonable only to a certain point because the possibility of bugs in the prewritten test scripts is great. If being able to model lab evaluation environment in circuit simulator the engineer could create and test evaluation scripts on a simulator and then convert the simulation scripts into ATAC test scripts. The conversion should be done automatically by a software so that engineering effort would be minimized. The synthesis of circuit simulators and ATAC was discussed in high level during the development of ATAC but the challenges that exist were not fully assessed yet. However the idea is tempting.

7.3 Other future improvements

Other future improvements include adding support for different instruments and adapters as well as automatic controlling of environmental chambers. Making all commands to require prefix for numbers would simplify test scripts because conversions from one number type to another would simplify and parsing prefixes to numbers with separate commands would become obsolete. Also reloading a test script into ATAC could be made easier by adding a separate reload button. Graphical test script creation could be implemented in the future if the syntax of test commands turns out to be too difficult. Also a manual control window for instruments could be added to configure instruments using graphical interface.

Summary

This paper introduced a control and data collection software, ATAC, for automatic testing and evaluation of ASIC circuits. ASIC circuits were first discussed in general level and the reader was provided with information about testing and evaluation needed in production of ASICs. The basic idea of evaluation is to measure and test whether a new device works as it is designed to. It was shown in this thesis that using ATAC for automatic evaluation of ASICs can provide savings in time and money for a company if a large number of tests need to be performed.

The lab environment where ASIC software was designed to be run was introduced in detail to provide the reader understanding of the whole system. The main components of the system were

- ACE board that routes instruments automatically to different pins of DUT
- Instruments that supplied voltages and currents to DUT and retrieved measurement data from the circuit
- Bench EVM board that provides device specific external components for the ASIC and mounts to ACE board
- Adapters that convert communication from one protocol to another to enable communication across the system.

Object oriented programming has shown its superiority when dealing with large applications which is why OOP was used with LabVIEW to develop ATAC. In addition to modular and scalable structure of the software this enabled construction of hardware abstraction layer so that adding new hardware support to the software is easy and does not involve modifying the main source code. The ATAC was designed so that it supported parallel execution of different tasks which enables the user to work with other parts of the software at the same time a test is being executed. The software development was managed by a Spiral type of life cycle model where different stages: specification, design, implementation and testing were practiced in a cyclic manner. Use of ATAC was described briefly and the reader was provided a basic understanding of how the software is to be used.

Successfulness of the ATAC software project was assessed by comparing the final software to the specification and by user feedback received from the end users who performed several evaluation measurements for real ASIC devices using ATAC. The features and usability of ATAC were considered to be good and clear improvement to previous software solutions. Comparing the amount of data ATAC was able to produce in a same amount of time as a human doing the same measurements the ATAC proved it’s superiority over manual testing. As a result more data could be gathered than before and the device under test could be tested in various different conditions.

References

[1] Davis, Alan M. Bersoff, Edward H. & Comer, Edward R. A Strategy for Comparing Alternative Software Development Life Cycle Models. In: IEEE Trans. Software Eng. Vol. 14. Nr. 10 (1988). S. 1453-1461.

[2] Amara, Amara. Amiel, Frederic & Ea. Thomas. FPGA vs. ASIC for low power applications. In: Microelectronics Journal . Vol. 37. Nr. 8 (2006). S. 669-677.

[3] Nobelprize.org website. The History of the Integrated Circuit. [Cited 23.1.2012]. Available at: http://www.nobelprize.org/educational/physics/integrated_circuit/history/

[4] Boehm, Barry W. A Spiral Model of Software Development and Enhancement. In: Software Engineering Project Management (1987). S. 128-142.

[5] Bellis, Mary. The History of the Integrated Circuit aka Microchip. [Cited 9.1.2012]. Available at: http://inventors.about.com/od/istartinventions/a/intergrated_circuit.htm

[6] Pokkunuri, Bhanu Prasad. Object Oriented Programming. In: SIGPLAN Notices, Vol. 24. Nr. 11 (1989). S. 96-101.

[7] Chance, Elliot. Vipin, Vijayakumar. Wesley, Zink. Richard, Hansen. National Instruments LabVIEW: A Programming Environment for Laboratory Automation and Measurement. Journal of Laboratory Automation. February 2007. Vol. 12. No. 1. S. 17-24.

[8] Cordan, Bill. An Efficient Bus Architecture for System-On-Chip Design. In: IEEE 1990 Custom Integrated Circuits Conference.

[9] Kuon, Ian & Rose, Jonathan. Measuring the Gap Between FPGAs and ASICs. In: IEEE Trans. on CAD of Integrated Circuits and Systems . Vol. 26. Nr. 2 (2007). S. 203-215.

[10] Irazabal, Jean-Marc. Blozis, Steve. I^2C Bus application note. Philips Semiconductors. 2003. [Cited 25.1.2012]. Available at: http://www.nxp.com/documents/application_note/AN10216.pdf

[11] Jehander, Jörgen. Graphical Object-Oriented Programming In LabVIEW. NI Developer Zone. [Cited 15.11.2011]. Available at: http://zone.ni.com/devzone/cda/tut/p/id/3390

[12] LabVIEW 2011 SP1 run-time engine at National Instruments website. Available at: 11.1.2012. http://joule.ni.com/nidu/cds/view/p/id/2292/lang/en

[13] LabVIEW Basics I: Introduction Course Manual. National Instruments. 2008.

[14] LabVIEW Core 2 Course Manual. National Instruments. 2010.

[15] Chen, Michael. Object Oriented Programming in LabVIEW for Acquisition and Control Systems at the Aerodynamics Laboratory of the National Research Council of Canada. 22nd International Congress on Instrumentation in Aerospace Simulation Facilities, from 06/10/2007 to 06/14/2007. Pacific Grove, CA.

[16] National Instruments homepage. LabVIEW. [Cited 15.11.2011]. Available at: http://www.ni.com/labview/

[17] National Instruments homepage. TestStand. [Cited 17.1.2012]. Available at: http://www.ni.com/teststand/

[18] National Instruments VISA (Virtual Instrument Software Architecture). [Cited 11.1.2012]. Available at: http://cnx.org/content/m12288/latest/

[19] NI Developer Zone. How to Mitigate Hardware Obsolescence in Next-Generation Test Systems. [Cited 15.11.2011]. Available at: http://zone.ni.com/devzone/cda/epd/p/id/6307

[20] NI Developer Zone. LabVIEW and Hyperthreading. [Cited 15.11.2011]. Available at: http://zone.ni.com/devzone/cda/tut/p/id/3558

[21] Nuseibeh, Bashar. Weaving Together Requirements and Architectures. In: IEEE Computer. Vol. 34. Nr. 2 (2001). S. 115-117.

[22] Bishop, Peter. A tradeoff between microcontroller, DSP, FPGA and ASIC technologies. 2009. [Cited 15.11.2011]. Available at: http://www.eetimes.com/design/industrial-control/4016917/A-tradeoff-between-microcontroller-DSP-FPGA-and-ASIC-technologies

[23] Dyke, Richard P. Ten & Kunz, John C. Object-Oriented Programming. In: IBM Systems Journal . Vol. 28. Nr. 3 (1989). S. 465-478.

[24] Gupta, Rajesh K. & Zorian, Yervant. Introducing Core-Based System Design. In: IEEE Design & Test of Computers. Vol. 14. Nr. 4 (1997). S. 15-25.

[25] Ovaska, Seppo. Course material of “S-81.2200 Embedded microprocessor systems”. Lecture 1 slides. [Cited 6.2.2012]. Available at: https://noppa.aalto.fi/noppa/kurssi/s-81.2200/etusivu

[26] Pease, Robert A. The Design of Band-gap reference circuits: Trials and Tribulations. In: IEEE Proceedings of the 1990 Bipolar Circuits and Technology Meeting. September 17-18. 1990. Minneapolis, Minnesota.

[27] Taewan, Kim. Yunmo, Chung. Efficient Hardware IP Control and Simulation with LabVIEW. ASIC, 2009. ASICON ’09. IEEE 8th International Conference on.

[28] TDM Excel Add-In for Microsoft Excel Download. [Cited 17.1.2012]. Available at: http://zone.ni.com/devzone/cda/epd/p/id/2944

[29] VI Technology Automated IC Characterization and Evaluation System. [Cited 15.11.2011]. Available at: http://www.vpc.com/products/applications/application_articles.cfm?ArticleID=24

[30] Spiral model image from Designing Project Management webpage. [Cited 14.2.2012]. Available at: http://www.designingprojectmanagement.com/SoftwareProcessModels.html

Appendix

A Supported commands in ATAC

instr_init: <instrument name>, <2/4-wire>, <front/rear>

Initializes specified instrument. Instrument terminal (front/rear) is selected as well as whether the instrument uses 4-wire mode for accurate sensing.

instr_close: <instrument 1 name>, <instrument 2 name>, ... , <instrument N name>

Closes communication to all specified instruments.

instr_get: <instrument name>, <parameter 1>, ... <parameter N>, <var 1>, ... , <var x>, <channel 1>, ... , <channel y>

Instructs specified instrument to take measurement and return result. Parameters 1…N are instrument specific parameters and depend on whether the instrument is multimeter, source meter or battery simulator. After instrument specific parameters variables and channels can be given for storing measurement result.

instr_set: <instrument name>, <function>, <value>, <compliance level>

Instructs specified instrument to source desired voltage or current. Instrument can source meter, battery simulator or power source.

connect: <ACE board terminal x> --> <bench EVM column y>, ... , <ACE board terminal z> --> <bench EVM column k>

Connects instrument from specified ACE board terminal to a specified bench EVM board column. ACE board terminal can be one of the following: A1, A2, A3, A4, B1, B2, B3, B4. Bench EVM column can be one of the following: column 0, column 1, column 2, column 3, column 4, column 5, column 6, column 7.

disconnect: <ACE board terminal x> --> <bench EVM column y>, ... , <ACE board terminal z> --> <bench EVM column k>

Disconnects specified instrument connection. See connect command for more information about the parameters.

disconnect: Instrument matrix

Disconnects all instruments from bench EVM board.

disconnect: Power relays

Opens all power relay connections.

disconnect: All

Disconnects all instruments from bench EVM board and opens all power relay connections.

resload: <value>

Sets resistance value of ACE board resistance load to a specified value.

set_local_meas_folder: <relative path>

Modifies existing measurement folder path. The modified folder path is valid only in a test where this command is executed.

set_global_meas_folder: <relative path>

Modifies existing measurement folder path. The modified folder path is valid for all tests that are executed after this command.

write_read_uart: <adapter name>, <string to send>, <variable where read data is stored>

Sends UART command using specified adapter. Read data is stored in a variable specified by user. UART settings must be configured before executing this command.

write_read_spi: <adapter name>, <data to send in hex format separated with whitespace>, <variable where read data is stored>

Sends SPI command using specified adapter. Read data is stored in a variable specified by user. SPI settings must be configured before executing this command. Data to send is given in hexadecimal format. For example: 80 01 FF sends 24 bits in groups of 8 bits.

write_i2c: <adapter name>, <address>, <register>, <data>

Sends I^{2}C write command using specified adapter. I^{2}C settings must be configured before executing this command.

read_i2c: <adapter name>, <address>, <register>, <variable where read data will be stored>

Sends I^{2}C read command using specified adapter. I^{2}C settings must be configured before executing this command. Last parameter is a variable where the read data will be stored.

write_gpio: <adapter name>, gpio <x> = <high/low>, ... , gpio <z> = <high/low>

Sets selected GPIO lines high or low. GPIO settings must be configured before executing this command.

read_gpio: <adapter name>, <variable 1> = gpio <x>, ... , <variable n> = gpio <z>

Reads selected GPIO line states to specified variables. GPIO settings must be configured before executing this command.

var: <operation 1>, ..., <operation N>

Performs variable operations. Operations can be initializing variables or performing calculations. The operation parameter can be following types:

- x
- x = 0
- x++
- x—
- x += 3
- x -= 2.4
- x *= 2
- x /= 2
- x = 1 + 2
- x = 3 – 1
- x = 2*4
- x = 4/2

Variable’s value can be accessed anywhere from a test script by writing variable name and adding [ ] around it.

array: <array name> = <element 1>, <element 2>, ... , <element n>

Sets array elements. Array elements can be accessed anywhere from a test script by writing {<array name>.<element index>}

wait_ms: <delay in milliseconds>

Delays test execution for amount of time in milliseconds specified by user.

wait_s: <delay in seconds>

Delays test execution for amount of time in seconds specified by user.

pause:

Pauses test execution until user instructs the program to continue.

channels: <channel 1 name>, ... , <channel N name>

Specifies channels that are used for storage for measurement data. Channel data is shown in main window graph and the data is stored into TDMS file when test is finished.

channel_put: <channel name> <-- <value to add>

Appends a decimal valued number to a specified channel data.

channel_put: <value to add> --> <channel name>

Appends a decimal valued number to a specified channel data.

increment_bits: <variable name>, <range>, <increment amount>

Range defines bit indexes that are used to get a sub set of bits of a variable value. This set is incremented by the specified amount. Example of range definition: 0-3. This gets bits from variable that are at indexes 0, 1, 2 and 3 starting from least significant bit.

decrement_bits: <variable name>, <range>, <decrement amount>

Range defines bit indexes that are used to get a sub set of bits of a variable value. This set is decremented by the specified amount. Example of range definition: 0-3. This gets bits from variable that are at indexes 0, 1, 2 and 3 starting from least significant bit.

set_trim_bits: <variable name>, <range>, <measurement channel>, <target value>

Finds value from measurement channel that is a closest match to a specified target value. Writes index of closest value in channel to a variable value subset defined by range. Example of range definition: 0-3. This gets bits from variable that are at indexes 0, 1, 2 and 3 starting from least significant bit.

reverse_bits: <variable name>

Reverts the bit representation of a variable value and stores the result back to the variable.

concate_strings: <variable name> = <string 1>, ... , <string n>

Concatenates all the strings and stores the result in a variable.

string_subset: <variable name> = <original string>, <range>

Gets subset of original string and stores the result in a variable. The indexes of original string that are included in a sub string are defined by range parameter. Example of range definition: 0-3. This gets four first characters from original string starting from left.

hex2bin: <variable name> = <hex value>

Takes hex value and stores it in binary format in a specified variable.

bin2hex: <variable name> = <bin value>

Takes binary value and stores it in hexadecimal format in a specified variable.

binary_and: <variable name> = <value 1>, <value 2>

Performs binary AND operation for the two values and stores result to a variable.

binary_or: <variable name> = <value 1>, <value 2>

Performs binary OR operation for the two values and stores result to a variable.

maximum: <variable name> = <value 1>, ... , <value n>

Gets largest value from the given values and stores it into a variable.

maximum: <variable name> = <value 1>, ... , <value n>

Gets smallest value from the given values and stores it into a variable.

absolute: <variable name> = <value>

Gets absolute value of the given parameter and stores the result to a variable.

custom_module: <relative path to a module>

Executes an external LabVIEW VI located in a path specified by user.

do:

Starts a do-while loop structure.

while: <value 1> <operator> <value 2>

Exits do-while loop structure if the condition specified is true. Operator can be <, >, <=, >=, == or !=.

if: <value 1> <operator> <value 2>

Starts conditional structure if condition is true. Operator can be <, >, <=, >=, == or !=.

else_if: <value 1> <operator> <value 2>

Starts conditional structure alternative to previous conditional structure if condition is true. Operator can be <, >, <=, >=, == or !=.

else:

Starts conditional structure alternative to previous conditional structure if previous structures were not executed.

end_if:

Ends conditional structure. Must be placed at the end of conditional structure.