Benjamin Steinwender, MSc

A Distributed Controller Network for Modular Power Stress Tests

DISSERTATION

zur Erlangung des akademischen Grades
Doktor der Technischen Wissenschaften

Alpen-Adria-Universität Klagenfurt
Fakultät für Technische Wissenschaften

Betreuer	Prof. Dr. Wilfried Elmenreich
Universität	Alpen-Adria-Universität Klagenfurt
Institut	Vernetzte und Eingebettete Systeme
2. Begutachter	Prof. Dr. Andreas Steininger
Universität	Technische Universität Wien
Institut	Technische Informatik
Industriebetreuer	Dr. Michael Glavanovics
Firma	KAI GmbH - Kompetenzzentrum für Automobil- und Industrie-Elektronik

Klagenfurt, Juni 2016

Affidavit

I hereby declare in lieu of an oath that

- the submitted academic work is entirely my own work and that no auxiliary materials have been used other than those indicated,
- I have fully disclosed all assistance received from third parties during the process of writing the work, including any significant advice from supervisors,
- any contents taken from the works of third parties or my own works that have been included either literally or in spirit have been appropriately marked and the respective source of the information has been clearly identified with precise bibliographical references (e.g. in footnotes),
- to date, I have not submitted this work to an examining authority either in Austria or abroad and that
- the digital version of the work submitted for the purpose of plagiarism assessment is fully consistent with the printed version.

I am aware that a declaration contrary to the facts will have legal consequences.

Benjamin Steinwender, MSc   Klagenfurt, am 10. Juni 2016

Acknowledgments

Completing a PhD thesis cannot be done without the support of several people.

First of all, I want to thank Prof. Wilfried Elmenreich for guiding me in the subject and providing numerous fruitful discussions. The pointers to investigate several topics are greatly acknowledged.

This thesis was part of a larger research project – *Enhanced Materials, Methods & Applications for Power devices and Systems* – carried out at Kompetenzzentrum Automobil- und Industrie-Elektronik (KAI) in cooperation with Infineon Technologies Austria[1]. Therefore, I would like to thank Josef Fugger and Michael Glavanovics for offering me this topic. Michael, being my industrial supervisor, also helped me to break down the industry requirements into understandable scientific terms for this project.

A special thanks goes to my fellow colleagues: Sascha Einspieler, who helped me creating the HTOL prototype application in his Master's Thesis and implementing the majority of the microcontroller firmware. Gerald Palatin carried out his Bachelor's Thesis and Master's Thesis projects partly under my supervision. Klaus Plankensteiner developed the Test Plan Builder application in course of his Master's Thesis project. Sergei Bauer helped me implementing parts of the host software. Yevhen Nikitin, who helps us porting the microcontroller firmware to a new embedded target.

Many of the investigations presented in this thesis would not have been possible without the help from the hardware team: Roland, Sybille, Alexander and Markus. Together, we created the prototype schematics, PCB layouts and they helped with the debugging, so I could focus on the theoretical work. As a result, I provided them the test system architecture and software tools to speed up the development of their test applications.

Finally, I would like to express my gratitude towards my family and friends, who always supported me and encouraged me to pursue this project. Thank you for your mental support!

[1]This work was jointly funded by the Austrian Research Promotion Agency (FFG, Project No. 854247) and the Carinthian Economic Promotion Fund (KWF, contract KWF-1521/28101/40388).

Kurzfassung

DAS Feststellen der Zuverlässigkeit von Leistungshalbleitern wird von Abnehmern und entsprechenden Zuverlässigkeitsstandards (JESD22 und AEC-Q100) vorgeschrieben. Konventionelle Testsysteme bieten lediglich fix definierte Stresspulse für eine große Anzahl an getesteten Bauteilen in einem Klimaschrank an. Üblicherweise wird jedoch auf das Zurücklesen von Statusinformationen oder Messdaten während des Testablaufs verzichtet, da dies einen zu großen Aufwand in der Verkabelung bedeutet. Vorhergehende Bestrebungen haben bereits erfolgreich diese Messdaten und Statusinformationen erfasst. Dabei mussten jedoch Einschränkungen bei der Flexibilität des Testsystems in Kauf genommen werden.

Die Architektur des in dieser Arbeit vorgestellten Zuverlässigkeitstestsystems präsentiert einen neuen, modularen Ansatz. Durch die Verfügbarkeit von Mikrocontrollern für den automobilen Temperaturbereich (bis 125 °C) kann ein kleines, mit einer Kommunikationsschnittstelle ausgestattetes Steuermodul in der Nähe des getesteten Bauteils innerhalb der Klimakammer platziert werden. Die Steuersignale und die Messleitungen sind dadurch verkürzt und bieten eine verbesserte Signalqualität. Die Mikrocontroller verwenden die Kommunikationsschnittstelle zur Kommunikation mit einem Laborrechner. Dieser Hauptrechner verteilt die Testvorschriften an die einzelnen Steuermodule und synchronisiert deren ausgeführten Aktionen. Außerdem steuert der Hauptrechner zusätzlich angeschlossene Laborgeräte, visualisiert und speichert die gemessenen Daten.

Die Testvorschriften der Steuermodule können flexibel erstellt werden, ohne dass die Basissoftware dieser Module neu übersetzt und aufgespielt werden muss. Die Testansteuerung basiert auf dem Modell endlicher Zustandsautomaten. Im Ruhemodus kann das Modell durch ein anderes ausgetauscht werden. Um die zu testenden Bauteile anzusteuern und zu messen, wird der Einsatz der Skriptsprache *Lua* vorgeschlagen. Die Testvorschriften werden mit dem *Test Plan Builder* erstellt, welcher über die Möglichkeiten der verwendeten Mikrocontrollermodule und der angebotenen Lua Programmschnittstelle Bescheid weiß. Die fertige Testvorschrift wird mittels der Kommunikationsschnittstelle an das Steuermodul geschickt.

Drei verschiedene Testapplikationen demonstrieren die Einsatzflexibilität und Leistungsfähigkeit des neuen, modularen Testsystems.

Abstract

RELIABILITY testing of power semiconductor devices is required by customers and reliability standards like the JESD22 and AEC-Q100. Conventional test systems feature the application of fixed stress patterns for a large amount of devices in an environmental chamber at the same time. However, reading status information or acquisition of analog measurements is usually not performed due to the missing return signal interface. Previous efforts have already succeeded at providing these necessary measurement capabilities, although at the cost of complexity of the test setup.

The reliability test system architecture in this thesis presents a new, modular test system approach. Due the availability of microcontrollers for the automotive temperature range (up to 125 °C), a small networked controller module can be placed close to the tested device inside the environmental chamber. Therefore, the stimuli and measurement signal lines are local and the signal quality is improved. In order to provide the required flexibility, the microcontrollers use a serial bus interface to communicate with a host computer. The host distributes the test programs to the control modules and synchronizes their actions. Furthermore, the host is used for controlling externally connected instruments as well as visualizing and storing measurement data.

The test procedure for individual devices can be created without the need to change and re-compile the microcontroller firmware. The test control is based on the finite state machine model. The model can be changed when the controller is in idle state. Furthermore, the *Lua* script interpreter is proposed to control and measure the tested devices through the microcontroller periphery. To create the test programs for this modular test system, the *Test Plan Builder* tool is provided. It has knowledge about the configuration of the used hardware targets and the Lua functions to help the users design their test application. The test plan is transferred to the microcontrollers via the communication channel.

Three different test application examples demonstrate the versatility and performance of the new modular test system.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Life is like riding a bicycle.
To keep your balance
you must keep moving.

(Albert Einstein)

Contents

Before production release of power semiconductors, standards mandate the execution of qualification tests [1, 2]. As indicated by Figure 1.1, multiple Devices Under Test (DUTs) are subjected to electrical stress patterns. Multiple semiconductor failure modes are known, where the majority of them are highly temperature dependent [3]. Therefore, the DUTs are placed inside an environmental chamber to be tested at different ambient temperatures. Control and Data AcQuisition (DAQ) systems are required to apply defined stress patterns (CTRL – control) to the DUTs and to measure (MSR) responses. A host system may collect these data and store it. Due to the variety of different products tested, the re-use of existing components is limited.

This thesis introduces new concepts for modular, flexible and configurable power semiconductor life testing. These concepts are based on individual *intelligent* driving, control and protection modules. The modules are interconnected by local and global serial bus structures for the exchange of data, test parameters and timing information.

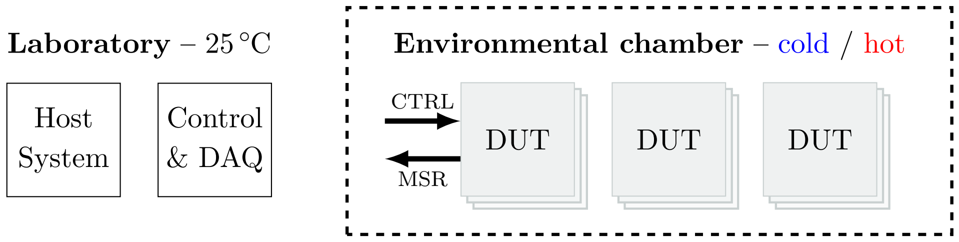


Figure 1.1: Components of a Stress Test System

The research questions answered within this thesis are:

- Can a modular, distributed test system be designed so that it is superior in terms of flexibility to a centralized system?
- Can the modular test system still be configured and programmed in such a way that it can easily be operated in real-life test laboratories?

1.1 Motivation

To develop a semiconductor life test for qualification and reliability testing these days requires tremendous effort. Typically, Commercial off-the-shelf (COTS) hardware in form of rack-based PCI eXtensions for Instrumentation (PXI) systems and interface cards is used. A custom software to handle the requirements is designed covering the individual requirements. In terms of re-usability, software components may be applied to similar projects. However, the system itself will be stand-alone and upgrading the system usually requires buying new interface cards and creating new software to satisfy the updated test requirements.

The AEC Q100 standard [1] describes the reliability qualification testing procedures required by the automotive industry. In a similar way, the JESD22 family of the JEDEC standards specifies the test methods for general industrial and consumer semiconductor applications [2]. The basic description of the test setup requires a specific number of DUTs to be tested. At intermediate time points, samples need to be removed from the test setup to be measured at specific Automated Test Equipment (ATE) systems. In situ monitoring of the devices is not required by standards. However, automated measurements during the stress testing reduce manual intervention during the test and, therefore, prevent possible human mistakes (e.g. mixing up samples) during this step.

During qualification testing, the requester of such a test would expect no device to fail in order to pass qualification. However, if failure data is not known, a reliability assessment is not possible. For that reason, end-of-life testing is typically performed beforehand at increased stress levels. Based on the obtained data, the lifetime can be predicted based on statistical models [4–6]. It may happen that the stress level is too low (or the tested devices are too robust) and most of the devices survive the test. Thus, only censored data is available. Statistical models can still be extracted from these data [7].

A significant amount of custom power electronics has to be developed and interfaced to operate such a life test. The complexity herein does not lie in the hardware development, but in the configuration and interfacing of the various modules, in order to provide a powerful yet convenient way for the test engineer to operate the system. Most hardware constraints can be met using COTS hardware. Powerful hardware (like a dual-core Advanced RISC Machines (ARM) with a Field Programmable Gate Array (FPGA) on the same chip[1]) to run the test program has been recently made available commercially. However, this and other similar products only operate up to 85 °C ambient temperature – which makes them useless for reliability life testing at higher ambient temperatures.

Developing a custom modularized test system on the other hand brings several benefits to the overall design procedure:

- Modularization of the individual parts enables simpler development, debugging and replacement of certain parts of the test system. Already existing and proven components can be re-utilized, while parts that do not satisfy the requirements can be upgraded or replaced by newer ones.
- A separate software development team can focus on the software design, while the hardware developers can focus on providing functional test hardware.
- When the test system architecture stays the same for a variety of different test applications, the test development time decreases significantly. Further, only test execution code needs to be adapted, while managing routines for the test hardware have already been tested.
- To improve different modules, it is beneficial to address one change at a time for a smaller development team.

[1] http://www.xilinx.com/products/silicon-devices/soc.html

1.2 Background

Increasing system integration and miniaturization of power semiconductors leads to high electrical, thermal and mechanical stress during operation. As a result, the structure of the power transistor degrades over its lifetime and along with it its performance. Therefore, reliability stress testing of power devices and applications prior to production release is required by customers and reliability standards (AEC-Q100 [1]).

In order to provide the means for testing multiple devices at the same time, single-device ATE cannot be used. ATE systems typically offer a centralized control instance with numerous stimuli and measurement devices connected, in order to perform parametric sweeps on all pins of the DUTs. These systems are optimized to run a quick production or characterization test to evaluate the devices' performance. Such an ATE is expensive and the test time is kept to a minimum, usually taking less than a second. Thus, running extensive stress tests is not cost effective and a variety of test systems are usually in use.

1.3 Evolution of Semiconductor Life Test Systems

1.3.1 HTOL

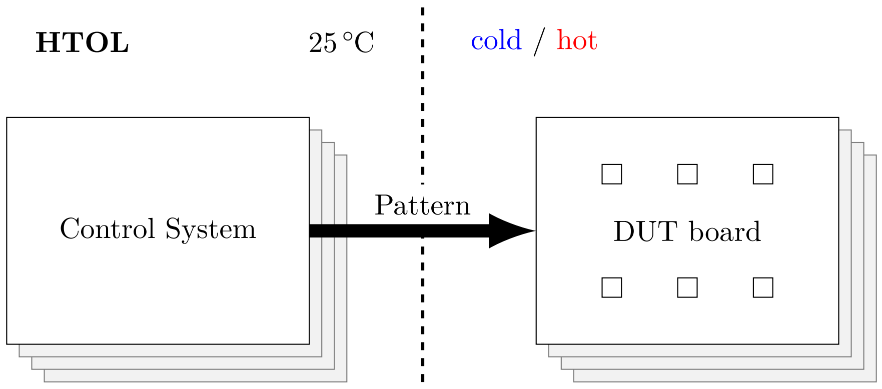


Figure 1.2: HTOL System Architecture

For High Temperature Operating Life (HTOL) test procedures according to the AEC-Q100 and JESD22-A108 standards [1, 8], multiple independent control systems are necessary to test the required amount of DUTs. They generally only provide fixed, repetitive control patterns to the DUTs and do not read device measurements or status (Figure 1.2). Improved solutions include control and readout possibilities in order to provide in situ measurement capabilities [9]. As specified by the JESD22-A108 standard, it is common to periodically stop the test, take all the DUTs out of the environmental test chamber to perform characterization measurements with the ATE and continue with the stress test after the characterization [10].

1.3.2 ACUTE / ARCTIS

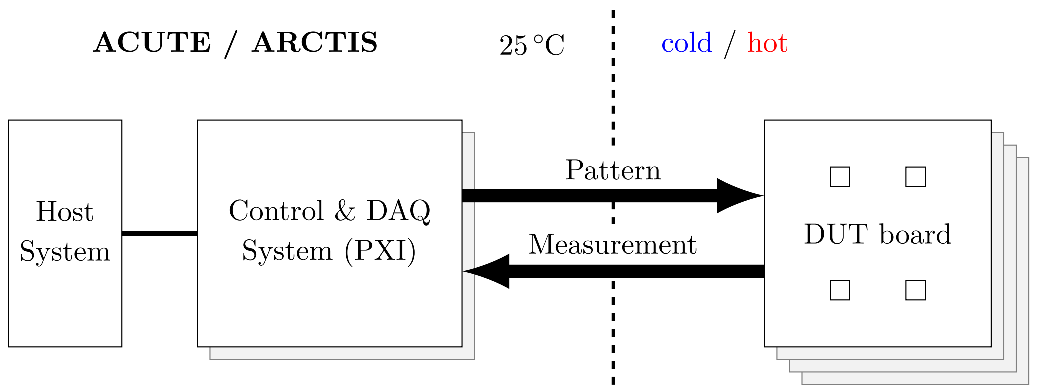


Figure 1.3: ACUTE System Architecture

The Active Cycle Universal Test Equipment (ACUTE) and the Advanced Repetitive Clamping Test Integrated System (ARCTIS) (Figure 1.3) are life test systems for smart power semiconductor switches providing short circuit switching and inductive clamping stress respectively [11–13]. They were developed by Infineon Technologies and KAI GmbH since 2004 and enhanced by multiple follow-up research projects in the successive years [14, 15]. The DUTs are located in a temperature climate chamber, in order to provide the required automotive temperature ranges from −40 °C to 125 °C. Both systems are able to apply periodic stress pulses for up to 256 devices, while performing in situ DAQ measurements of the tested devices.

The real-time control system is based on a monolithic hardware setup using PXI connected FPGA cards and external instruments. The control and data acquisition is performed on the FPGAs whereas data analysis is done on the host computer.

To provide measurement capabilities for one voltage and one current per DUT, the system features approximately 3000 individual signal connections through thermally isolated openings into the climate chamber. Therefore, the limitations of this system are mainly given by the signal count. It is understandable that such a system cannot be easily modified to perform additional measurements. Further, applying different stress pulse patterns requires a redesign of the FPGA code.

1.3.3 Distributed Measurement Systems

Previous efforts have already resulted in creating a distributed measurement architecture to re-utilize hardware and software components in terms of a client-server architecture [16] or an eXecutable Verification Plan (XVP) [17,18]. Test plans created within this XVP framework are eXtensible Markup Language (XML) based files that describe the features and the behavior of a single semiconductor device. These test plans can be compiled into different recipes. The recipes, in turn, can be used as either input for simulations using a Simulation Program with Integrated Circuit Emphasis, tests using an ATE or device characterization. Test programs in this format feature linear sequences and handle only single devices. However, for reliability assessment multiple devices need to be tested over longer time periods to obtain statistical data.

1.3.4 MoPS

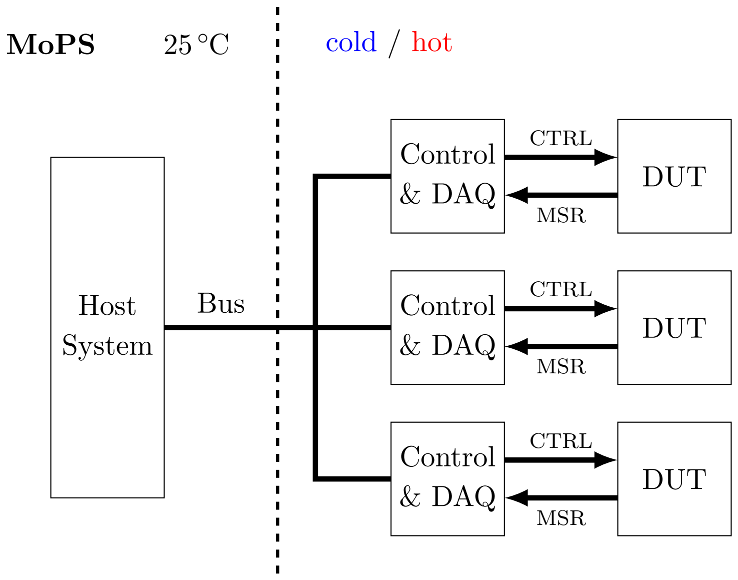


Figure 1.4: MoPS System Architecture

In order to improve the reconfiguration possibilities, a new power stress test architecture is proposed within this PhD project [19]. The architecture builds on the previously mentioned idea of configurable test procedures (XVP) by re-implementing it on smaller controllers and adding communication capabilities to enable parallel testing of multiple devices. As depicted in Figure 1.4, the new features include the control and data acquisition unit to be in proximity to the tested device or application. Therefore, it must be physically placed inside the climate chamber. The resulting shorter signal paths compared to the previous systems lead to improved signal quality. Further benefits are simplified signal connections, because the controller and the DUT are located close to each other. Old-fashioned, massively parallel cable harnesses are not necessary anymore, since the Printed Circuit Board (PCB) with the control and test hardware can be plugged directly into the PCB with the test application. The required connections to the outside of the climate chamber consist of a power connection and a data connection. Thereby, the controllers and the test application is supplied. Furthermore, the controllers are able to receive the test program and to send back status and measurement data.

1.4 Requirements

Within the last years, a lot of knowledge has been acquired by investigating and developing semiconductor reliability life test systems. From the previous section, we can derive the following set of requirements for the control of the new test system generation:

Fixed firmware There are several reasons to use a fixed test software. First, the software development resources are rather small for the variety of test applications addressed with this modular power stress test concept. Secondly, updating the firmware in a live system – where high voltages and high electrical power are present – is a dangerous task for the lab engineer, as some electrical components are not easily accessible. In addition, an approach is required that enables to change the basic behavior of the controller close to the tested device or application.

Flexible test procedure The test hardware target (i.e. the microcontroller) needs to perform numerous tasks. Many of them are common to the majority of applications: especially measurement data acquisition and communication with the host system. Observed device status and acquired measurement data may be used for further investigations. Therefore, device status and measurement data must be sent to a possibly centralized host infrastructure. In general, the executed test application determines the test configuration and the execution sequence, which may be greatly different from the previous application. The DUTs and their respective test periphery need to be configured and patterns need to be sent during the test. Depending on internal variables (e.g. counters) and external states (e.g. measurement values), different actions have to be carried out. Thus, the firmware of the microcontroller must be capable of executing such flexible procedures.

Configuration Because the configuration and test patterns are so diverse, it is not sufficient to enter this information in a Graphical User Interface (GUI) and discard the data as soon as the test has been concluded. The settings need to be stored in some files or database structure to be able to compare different tests. To create flexible test sequences (i.e. test code), the test engineers require some help to create these files. Further, the test system hardware setup must also be identified and stored in order to provide comparable reliability tests. These files shall be stored in a human readable text format.

1.5 Structure of the Thesis

This thesis presents the software framework developed for creation and deployment of test execution code. The various hardware prototypes and software projects have been developed in close cooperation with Bachelor and Master students [20–25].

The thesis is structured as following:

Chapter 2 – Concepts & Related Work gives an overview of the previously existing concepts used for this project.

Chapter 3 – The MoPS Distributed System explains the general architecture of the modular test system including the required components.

Chapter 4 – System Configuration & Programming focuses on the configuration of the test system by the operator and test requester. Further, the creation and distribution of the test descriptions, also called *test plans* is described.

Chapter 5 – Prototype Implementations gives an example of the specific test implementations based on the modular power stress test and compares the setup situations.

Chapter 6 – Results & Discussion presents the results of the distributed modular architecture and discusses the applications in the given real-life implementation scenarios.

Finally, the thesis will be concluded in Chapter 7 – Conclusion & Outlook.

2 Concepts & Related Work

Engineers like to solve problems. If there are no problems handily available, they will create their own problems.

(Scott Adams)

Contents

Numerous fields need to be addressed when creating a modular test system as proposed in Chapter 1. The microcontroller will be located close to the DUTs and therefore remote from the host computer. The microcontroller controls the DUT including its required guard and load modules (see Section 3.5) by applying stimuli and stress patterns. The host system is responsible for sending configuration data to the microcontroller as well as collecting and storing measurement data from the microcontroller hardware.

For that reason, communication is an important part of this system. The microcontroller needs to be able to receive, decode and execute the desired test procedures. Since the effort on low-level microcontroller programming for the target users should be kept low, a high-level language interface will be required.

This chapter first gives an overview of *Industrial Communication*, with the focus on the classification and suitable protocols for use in the later chapters. Next, the term *Smart Devices* and their current usage will be explained. The users of the test system will be required to specify parts of complex routines as program code. Thus, a section will introduce the concepts of comprehensible, simple *Programming Languages*. Further, possibilities to define and apply *Test Configurations* will be reviewed. Finally, this chapter deals with different *Software Deployment Strategies*.

2.1 Industrial Communication

As given by the modular test system requirements, the controller units will be connected to the host system through a serial data bus. The selection of the used bus depends on the capabilities of the used controller and the possibilities of a host computer to access the bus. It must be possible for multiple participants to access the bus and communicate to a possibly central host system for data storage. Furthermore, the communication electronics must be capable of withstanding a wide temperature range of $-40\,^{\circ}\mathrm{C}$ to $125\,^{\circ}\mathrm{C}$. According to the requirements, a suitable candidate had to be selected from several options which are described in the following paragraphs. Specifications are taken from the respective bus standards as quoted.

2.1.1 PROFIBUS

PROcess FIeld BUS (PROFIBUS)[1] is a field bus network with its origins in a cooperation between the German government and several industrial automation companies in 1989 [26, 27]. The specification was later defined in several standards, including the IEC 61158 fieldbus standard.

PROFIBUS uses differential signaling based on RS-485, but can also be implemented using fiber optics for a wider distance. A maximum number of 127 nodes is allowed. Distances between 100 m and 1.200 km allow transfer rates up to 12 Mbit/s. A message can carry up to 244 B of data.

A PROFIBUS network typically consists of a master control instance – the Programmable Logic Controller (PLC) – that periodically polls sensor data from the attached slave nodes and transmits set point values to actuators. The PLC, therefore, is the central brain of such network, where the slave nodes do not require a lot of processing power. Multiple master nodes are possible according to the PROFIBUS Fieldbus Message Specification (FMS). A token is exchanged between the master nodes, in order to define which master may query the nodes.

Two different versions, PROFIBUS Decentralised Peripherals (PROFIBUS DP) and PROFIBUS Process Automation (PROFIBUS PA), which are in use today, have been derived from the FMS. PROFIBUS DP is commonly used to query sensors and control actuators in a production line. PROFIBUS PA is designed for use in hazardous areas by using robust cabling. The maximum number of connected nodes for use with PROFIBUS PA is limited to 30 slave devices and the data transmission rate is fixed to 31.25 kbit/s [28].

2.1.2 CAN

Controller Area Network (CAN) [29, 30] is a serial bus network used widely in automotive and automation industry. It was developed by Bosch in 1983 in order to reduce the cabling effort in modern automotive vehicles [31]. The Bosch specification originally only included the Medium Access Control (MAC) protocol. The CAN ISO standard includes the Bosch specification, but also defines the physical layer. The High-Speed CAN version allows data rates up to 1 Mbit/s with a theoretical maximum

[1] `http://www.profibus.com`

bus length up to 40 m. In 2012, CAN FD (flexible data-rate) which allows higher transmission speed and increased data payload up to 64 B has been presented [32].

The most widely used high-speed CAN uses two signal lines, CAN_LO and CAN_HI, and one reference line (CAN_GND) for data transmission. The idle voltage of the signal lines is typically 2.5 V. CAN_LO signals the complementary value of CAN_HI. The difference between CAN_HI and CAN_LO is evaluated, allowing two bit levels to be transmitted. The bus lines are shared among the participating nodes. On a CAN network, up to 128 bus participants can be connected, depending on the drivers of the physical layer and the network length.

The so-called Carrier Sense Multiple Access / Collision Resolution principle is implemented, in order to establish a communication: A bus member waits until no other node transmits and then starts sending data. In contrast to Ethernet, CAN does not abort the message when it recognizes a transmission collision, but implements a specific collision resolution. Due to dominant and recessive bit encoding, the lower priority message is overruled by the higher priority message. Therefore, the higher priority message will always be transmitted. The lower priority senders automatically back off the bus and retry again later.

The variable latency of these messages increases the difficulty to meet deadlines. Numerous publications investigated the schedulability of messages being transmitted for real-time use [33, 34]. Since then, bus utilization of CAN networks has been increased to about 80 % of the theoretical throughput rate.

Nodes are not directly addressed on the bus, but their messages are encoded using a unique object Identification (ID). This ID is also used for the Carrier Sense Multiple Access (CSMA) arbitration described above. Two different message formats can be used: using standard frames with 11 bit identifiers (known as CAN 2.0 A) and extended frames with 29 bit identifiers (known as CAN 2.0 B). Except for the different length of the identifiers, the messages are built up equally.

	Arbitration		Control		Data	Footer			
Start	Identifier	RTR	res.	DLC	Data	CRC	ACK	EOF	IFS
1 bit	11 bit or 31 bit	1 bit	2 bit	4 bit	0 B to 8 B	16 bit	2 bit	7 bit	3 bit

Figure 2.1: CAN message format [31]

Figure 2.1 shows the CAN message frame for both standard data frames and extended data frames, consisting of the following fields:

Start-of-frame a dominant bit marks the start of a new CAN frame.

Identifier – Arbitration the ID field is transmitted next, in order to perform the bit wise arbitration based on the encoded priority. The standard CAN frame has an 11 bit identifier. By using one of the reserved bits in the standard format as *ID extension* following the first arbitration field, it is possible to send extended frames. Additional 18 bit of the second part of the ID are transmitted afterwards.

RTR – remote transmission request is used to differ between a data frame and a remote frame.

res. – reserved bits two reserved bits as part of the control field for future extensions.

DLC – data length coding specifies the number of data bytes transmitted in the body of the frame. For a remote frame, DLC specifies the number of expected data bytes.

Data 0 B to 8 B of data containing the body of the message follow. The length is determined by the DLC field.

CRC – cyclic redundancy check the result of the Cyclic Redundancy Check (CRC) for the header and body is sent together with a 1 bit delimiter.

ACK – acknowledgement a 1 bit time slot is transmitted recessively from the sender. Any receiver may set a dominant bus value in order to signalize correct reception of the message.

EOF – End-of-frame indicates the end of this message.

IFS – inter-frame-space before the next message can be sent, is a mandatory pause on the bus. Afterwards, the bus is idle and new messages can be sent by any node, following the arbitration procedure.

Only the lower layers of the Open Systems Interconnect (OSI) model [35] are specified by the CAN standard. Several higher level protocols have been developed to provide additional features on top of the basic data exchange possibilities. For example the following:

ISO-TP [36] allows sending messages larger than the maximum limit of 8 B of the standard frames by splitting the data into multiple CAN frames. These messages also include some information for the receiver to be able to reconstruct the message. Thereby, it deals with layers 3 and 4 of the OSI model.

CANopen [37] was developed within the CAN-in-Automation group, now also being an international standard (CENELEC EN 50325-4). The communication protocol is based on CAN networks. CANopen specifies a device model consisting of three layers: the communication, the object dictionary and the application layer. Communication models supported are master/slave, client/server and producer/consumer. The communication is performed using Process Data Objects (PDOs) for real-time fast exchange of process data and Service Data Objects (SDOs) for accessing the object dictionary. PDOs are exchanged without additional overhead and can be transmitted asynchronously (any time), synchronously (after a synchronization message) or on-demand. SDOs are used to send configuration and parametric data and implement point-to-point communication including fragmentation and reassembly. The object dictionary provides a logic addressing scheme for accessing the communication and device parameters. The application layer implements the devices' functionality and interacts with the Input & Output (IO) signals of the physical process.

DeviceNet [38] implements higher layer protocols aimed at industrial automation. The network also supplies power for the connected nodes. A DeviceNet network supports up to 64 individual nodes that are addressed using MAC based ID. These addresses are assigned dynamically and duplicate addresses are recognized during the start-up. Based on CAN 2.0 A, only the shorter 11 bit message identifiers are used. DeviceNet specifies an object model that describes node behavior. Messages are exchanged using the master/slave or producer/consumer communication scheme. Fragmentation and re-assembly of messages larger than 8 B is also supported.

2.1.3 Ethernet

In 1973, Ethernet was developed at Xerox PARC [35, 39]. The first Ethernet standard was published as IEEE 802.3 in 1983. It was an improvement over the existing ALOAHnet, by adding CSMA to the MAC layer. Originally, Ethernet was used on a shared, coaxial cable. Afterwards, twisted-pair cables with full duplex communication channels were introduced, being the main reason to win the competition against other Local Area Network (LAN) communication technologies such as Token Ring and Token Bus [40].

Ethernet is well known for the Carrier Sense Multiple Access / Collision Detection (CSMA/CD) arbitration access mechanism. A bus participant observes the common medium (the Ether) and may begin transmission when it is free. This Carrier Sense part is also used to detect collisions on the network. When a collision is observed, a jamming signal is broadcast in order to immediately abort transmission of all conflicting senders. The current packet is discarded and enqueued for another transmission retry. The retransmission must wait a randomized time, depending on the sender's hardware address and the number of retries. Therefore, the retry of the two competing parties will likely happen at different times and the probability of another collision to happen is less.

Preamble		Header			Body	Footer
Preamble	Start	Destination	Source	Type	Data	CRC
7 B	1 B	6 B	6 B	2 B	46 B to 1.500 kB	4 B

Figure 2.2: Ethernet frame [35]

Figure 2.2 shows the composition of the Ethernet II frame. Version 1 of the Ethernet frame, which used smaller destination and source addresses, was never in productive use.

Preamble The frame starts with a preamble used to synchronize the clocks of the nodes on the network. 7 B of alternating ones and zeros are transmitted.

Start The Start-of-frame delimiter marks the end of the preamble and the beginning of the Ethernet frame.

Destination The header of the packet then starts with the destination MAC address.

Source The source MAC address specifies the sender of the packet.

Type Ethernet packets specify the type of the transmitted data in the following field. A value less or equal to 1500 is interpreted as the length of the payload. Higher values are commonly used to interpret the payload data for higher layers of the OSI model.

Payload The data portion of the packet must have a minimum length of 46 B. This size is required in order to guarantee full occupation of the medium and enabling the collision detection feature. The maximum packet length is 1.500 kB.

CRC Ethernet uses a 32 bit CRC sum to provide simple error detection on the receiving side.

The MAC address for Ethernet is a 48 bit (6 B) number which must be unique on the network. A single bit within the most significant byte of the MAC address is used to distinguish whether the MAC address is universally administered or locally by the network administrator. As a result, implementing custom addresses in a separate network is possible without the need of registering or buying a vast amount of MAC addresses.

The introduction of Ethernet *switches* enables network participants to use point-to-point connection links and therefore making the CSMA/CD media access requirement obsolete. With the use of twisted pair cabling, sending and receiving of data at the same time became possible by using a separate electrical channel for each direction. A switch receives the packets per connected line in a queue and inspects the header of an Ethernet packet to decide to which portion of the network it must be forwarded. By remembering the source MAC addresses of the Ethernet packets, a destination table will be assembled. Thereby, only part of the Ether is occupied and multiple nodes can send and receive data independently at the same time.

Industrial Ethernet

While the standard Ethernet uses a shared medium, it still provides high utilization of the overall bandwidth [39]. These networks provide only a best-effort guarantee of messages being transmitted. Industrial field buses on the other hand require deterministic worst-case delivery times for process automation.

Industrial Ethernet can be classified into four categories according to [41, 42]:

1. **Full Ethernet** according to IEEE 802.3 uses COTS equipment for participating nodes and network devices. It can be used together with other Industrial Ethernet solutions and standard Ethernet. However, only Quality of Service and best-effort service can be guaranteed.

2. **Ethernet Compatible** networks use specific devices on standard infrastructure (i.e. switches). To guarantee real-time performance, network traffic management takes place. The benefit of using specific network devices is also the biggest drawback.

3. Common Ethernet devices using standard 802.3 layers according to the OSI hierarchy. The higher layers are modified in order to provide deterministic services.

4. Field buses using Ethernet links provide a modified MAC layer to enable real-time communication. Hard real-time performance can be achieved. However, these networks cannot coexist with existing standard Ethernet devices.

2.1.4 EtherCAT

Ethernet for Control Automation Technology (EtherCAT) [43,44] is an Ethernet-based fieldbus developed by Beckhoff. It has recently been included in the IEC 61158 fieldbus specification [28]. EtherCAT is a master-slave bus with one active master allowed. The slaves are typically connected in a ring topology, but line, tree and star or any combination are also possible.

The master device in the network does not require special hardware and can be implemented on a COTS computer using standard full-duplex Ethernet Network Interface Cards (NICs). Slave devices however require a modified MAC unit. Therefore, EtherCAT belongs to the last category listed in the previous section. Packets are received on the Ethernet receive channel (RX) from the upstream connection and sent to the transmission channel (TX) of the downstream connection. The last slave of an EtherCAT network segment detects the missing downstream link and automatically returns the frames on its receiving side. In addition, the *circulation* status bit is set, to prevent frames from circulating in separated segments. If the *circulation* bit is already set, the frame is destroyed. Thus, the master can immediately detect cable failure or a network separation. Cable redundancy and hot swap of slaves are also supported by using a second NIC on the master node.

EtherCAT frames are standard Ethernet frames with the Ethernet type field set to `0x88a4`. EtherCAT can also be inserted in User Datagram Protocol (UDP) frames to enable Internet Protocol (IP) routing. In contrast to Ethernet, the frames are not stored and inspected at the receiving side, but they are processed while they are being forwarded to the next receiver. Hence, data is read and written on-the-fly. Since each slave can modify the data, the Ethernet CRC is checked and re-computed for each frame. Erroneous frames are immediately discarded when detecting the error.

A single EtherCAT frame consists of its frame header and several datagrams – called Protocol Data Units (PDUs) – which address individual parts of the process image. Therefore, a single frame can be used to address multiple variables and/or slaves at the same time improving the network utilization. Each PDU contains a separate field for the command, address, data and working counter. The address can be positional (the slave node position in the line architecture), direct (node address) or logical. The first two methods are physical addresses which directly access memory on the associated slave. Logic addresses are translated using a Fieldbus Memory Management Unit working similar to the Memory Management Unit of a modern processor. These logic addresses allow mapping of very small bit-wise data signals into the larger process image.

EtherCAT also features a distributed clock for slave synchronization. Broadcast frames are used to capture the individual clock offset values of the slaves. After acquiring these offset times, they are used to compensate the different arrival times of the frames due to the ring topology.

Further, EtherCAT also provides a mailbox service to enable several additional capabilities. Ethernet over EtherCAT can tunnel all Ethernet frames. CANopen over EtherCAT implements the PDO message mechanism. File access over EtherCAT can transfer firmware images and other files similar to File Transfer Protocol. Servo drive over EtherCAT maps the servo drive profile to the EtherCAT mailbox.

2.2 Microcontrollers

The first microcontroller was introduced in 1974 [45]. In contrast to micro processors, microcontrollers have all required components (Arithmetic Logic Unit, Random-Access Memory (RAM), Read-Only Memory) on the same chip. Consequently, they do not need additional external components. Microcontrollers used in factory and test automation commonly are slave devices on digital bus interfaces. They sense and process physical quantities, communicate with a master device and interact with the environment. These devices are also known as smart transducers [46, 47]. The main concept at this stage still relies on measuring or actuating purposes which still require a real-time connection to a, possibly centralized, controller [48]. Therefore, many architectures emphasize the real-time capabilities of the communication system [49, 50]. With increasing capabilities of a node, the smart devices become smart controllers, which are able to locally control a system part autonomously. An overview of control and management methods for such smart devices is given in [51].

Today's microcontrollers feature many peripheral modules to perform specialized tasks in addition to the main Central Processing Unit (CPU). Typically, serial communication (Universal Asynchronous Receiver Transmitter (UART), Serial Peripheral Interface (SPI), Inter-Integrated Circuit, CAN, and Ethernet), several hardware timers, Pulse Width Modulation (PWM) and analog converters are present. Using Direct Memory Access (DMA) controllers and special trigger lines within the microcontroller, these modules can communicate with each other without requiring any software interaction through Interrupt Service Routines (ISRs).

Microcontrollers with a 32 bit CPU core, like the ARM Cortex-M[2], provide the performance for the efficient execution of interpreted script languages. The latest developments also present the qualification of these high performance microcontrollers for use in the automotive ambient temperature range up to 125 °C.

[2]http://www.arm.com/products/processors/cortex-m

2.3 Programming Languages

This section gives an overview of available interpreted languages for use on a microcontroller target. Further, used languages and software libraries on the host system are outlined here.

2.3.1 Code Interpreter

Over the last years, networked boot-loaders have emerged that allow transferring executable code during power up of the controller via serial data buses such as UART, CAN or Ethernet [52, 53]. Using these mechanisms, updating the controller firmware can be simplified substantially, since it is not required to plug the flash tool into each controller individually. However, this does not solve the problem that for each different task a new firmware image needs to be created.

Improved concepts implement a common base firmware that supports the execution of script commands. Several projects have already investigated approaches to provide embedded Virtual Machines (VMs) for Lua (*eLua* [54]) or Python (*p14p* "Python-on-a-chip" [55], the *Owl Embedded Python System* [56] and MicroPython [57]) which allow the execution of user provided code on a microcontroller.

Python

Python [58] is a high-level, interpreted programming language, which first appeared in 1991. Multiple paradigms like object-oriented programming, structured programming and functional programming are supported. Its most distinguishing feature is to use indentation as syntax element to describe blocks of code, whereas common languages like C and Java use curly braces. The language follows the principles of simplicity, clarity and readability.

Python contains several basic built-in data types: `str` (string), `int` (an integer value of arbitrary length), `float` (floating point number), `complex` (complex numbers) and `bool` (boolean values `True` and `False`). Additionally, Python also provides the advanced types: `list` (a mutable list of values of mixed types), `tuple` (an immutable ordered set of mixed types), `set` (a mutable unordered set) and `dict` (a dictionary of key-value pairs). In addition, custom types deriving from the class `object` can be created.

The reference implementation of Python *CPython* is written in C. Python programs are first compiled into the Python bytecode to be executed in the Python VM. Python can be extended by writing libraries in either C or Python. Therefore, Python is suitable for custom extension and accessing microcontroller hardware.

Barr [56] describes a sophisticated software framework based on the Python VM that removes the hitherto required compile-flash cycle of microcontroller programming. This approach features an interactive Python interpreter on the microcontroller. The original Python VM is modified and only a subset of Python commands is made available. The Python program however needs to be compiled into bytecode on a desktop machine and is transferred to the microcontroller via a serial data bus for execution. Additionally, the bytecode can also be stored on the controller to be run independently.

In contrast, the *MicroPython* project features a reimplementation of the Python language on a microcontroller [57]. It contains all language features as well as an on-line interpreter. Furthermore, a read-eval-print loop can be accessed via its serial connection and programs can also be stored in the embedded device's flash memory.

Lua

Lua [59] is a scripting language designed for use in embedded systems. Created in 1993, its main goals are to be embeddable, small and simple. Lua is implemented in C and also runs on a VM.

There are fewer basic data types present in Lua compared to Python: `nil`, `boolean`, `number` (since Lua 5.3, it is possible to distinguish between integers and floating point numbers [60]), `string`, `userdata` (to hold arbitrary C data structures), `function`, `thread`. Lua has only one built-in data structure, the `table`. Tables are associative arrays where the keys can be any Lua value except `nil`. They can also be accessed in an object-like way (i.e. the field access `t.x` is equivalent to the indexed access `t["x"]`).

Multiple programming paradigms are possible [61]. Object-oriented programming can be achieved through meta-tables where they provide additional features to ordinary Lua `table` and `userdata` values. Functions in Lua are first-class values. They are always anonymous, can be assigned to a value and also passed to other functions as arguments. Therefore, functional programming is also supported.

Lua is not a standalone language, but a library that can be included into the host application [62]. An extensive Application Programming Interface (API) is provided to

Listing 2.1: Lua C-API Example [62]

```c
static int l_sqrt (lua_State *L) {
    double n = luaL_checknumber(L, 1);
    lua_pushnumber(L, sqrt(n));
    return 1;  /* number of results */
}
```

create custom modules and run custom code side-by-side with Lua. All Lua functions operate on either the Lua registry or use the Lua stack interface by retrieving and returning values (see Listing 2.1).

Many programs and embedded systems feature Lua scripting capabilities. The *eLua*-project runs a full Lua interpreter on the microcontroller. Thus, the script can be compiled on-line and all language features are available. eLua also provides an interactive command access for simple debugging through a serial connection. Additionally, Lua supports the loading of pre-compiled bytecode in addition to raw source files.

Evaluation

Both script language projects provide a powerful C-API for the incorporation of custom C-functions, enabling simple access to the controller's hardware periphery. In addition, they provide means for a standalone program execution.

The Lua API however is simpler and does not require a special *value* type in C. All values exposed to C are native data types such as integers, floats and character pointers (strings). Therefore, there is no need for manual garbage collection of these C data structures in Lua.

The previously mentioned existing projects that aim to put a script interpreter onto microcontrollers fall short in terms of distributing the program code to multiple controllers at the same time. However, distribution is essential if multiple controllers are situated in the same test system.

Further, numerous different test applications need to be addressed with a common base platform. In general, different functions for a specified output pin are required. Thus, the effort to create a new test description must be kept small, preferably without flashing a new firmware to the microcontroller.

2.3.2 LabVIEW

National Instruments LabVIEW is a proprietary development platform for DAQ, test automation and embedded system design. LabVIEW provides native support for Virtual Instrument Software Architecture (VISA) via General Purpose Input Bus (GPIB) [63], USB, serial as well as raw Ethernet (Transmission Control Protocol (TCP) sockets) and LAN eXtensions for Instrumentation (LXI) connections. Therefore, it is very simple to interface COTS laboratory equipment such as power supplies, electronic loads or oscilloscopes for control and measurement purposes. Code is entered graphically by strictly following the data-flow programming paradigm. Within the development environment, the entered code is immediately compiled to native code for the CPU and ready for execution. There is no extra compile step required.

Being a high-level language, it is much simpler to create and debug software than writing it in C. Further, LabVIEW inherently provides parallel execution on multiple CPU cores. For synchronization purposes between those parallel threads, queues (similar to mailboxes), notifiers and events can be used. LabVIEW further supports object-oriented software design enabling many architectural concepts borrowed from other languages like C++ or Java.

Within the host software project described in this thesis (see Section 3.1), several readily available libraries are used:

- The *Actor Framework* [64] is a software architecture for the LabVIEW environment based on the actor model [65]. An *Actor* is a self-contained, executable object instance that can send and receive messages. Thus, a Model–View–Controller interface can be implemented, where the Actor-derived class contains the model.
- The *Character Lineator*[3] is a serialization library for LabVIEW. It is provided on the National Instruments community forums and capable of serializing and de-serializing LabVIEW data structures into arbitrary formats. So far, the library can transform all LabVIEW basic data types as well as LabVIEW classes from and to a simple variant of XML, JavaScript Object Notation (JSON) and binary formats. The serialization framework can easily be extended for custom formats like the Lua format.

During the design of the host software, scripting capabilities for use in LabVIEW have been evaluated. An important requirement is that the scripting languages

[3] https://decibel.ni.com/content/docs/DOC-24015

on both microcontroller and host platform are the same, so the lab engineer does not need to learn two different interfaces. *Lua for LabVIEW*[4] previously known as *LuaVIEW* [66] provides the Lua scripting engine for LabVIEW. Thus, Lua script code can be executed with call-back functions written in LabVIEW (e.g. for instrument control). However, *Lua for LabVIEW* is a closed source commercial library and requires a license for use.

2.3.3 Graphviz

Listing 2.2: Graphviz dot script for the Finite State Machine (FSM) display

```
digraph g_1 {
    //    default settings:
    node [fontsize=14];
    edge [fontsize=12, arrowsize=0.75, splines=curved];
    //    current FSM state in red:
    "IDLE" [color=red, fontcolor=red];
    //    all transitions with set events colored in blue:
    "IDLE" -> "INIT" [label="start", fontcolor=blue, color=blue]
    "INIT" -> "WAIT" [label="@else"]
    "WAIT" -> "RUN" [label="@eru3"]
    "RUN" -> "CLEANUP" [label="stop"]
    "CLEANUP" -> "IDLE" [label="@else"]
}
```

Graphviz [67] (short for “graph visualization”) consists of a language (“*dot*”) capable of describing directed and undirected graphs. Metadata and style annotations can be added to the graph description to describe the visual appearance. Further, it includes a set of tools (programs) to render images from this description. Each tool uses a different layout algorithm to display the nodes of the graph. Numerous image output formats including Portable Document Format, BitMaP and Portable Network Graphics are possible.

2.4 Markup Languages for Representing Test Configurations

Modular configurable software needs some kind of configuration files to specify parameters for the compiled application at run time. Such parameters may be the used

[4]http://luaforlabview.com

Ethernet port numbers, attached external instruments and their physical addresses or timing and debugging parameters. In order to provide this information, text formats – preferably human-readable – are required.

2.4.1 INI File

The INI file format[5] is used for some operating systems and applications. The name was derived from its main use, as "initialization" files, and the typically used file extension ".INI".

Listing 2.3: INI example

```
[section]
key = value
```

Within the INI file (see Listing 2.3), each configuration entry is written on its own line. The configuration entries are key-value-pairs, delimited by the equality sign (=). Values are typically strings, where other types (e.g. numbers) need to be converted by the parser.

Entries can optionally be grouped by sections, therefore providing a minimum hierarchy. However, deeper hierarchy levels are not defined and usually the text parsers need to take care of this.

2.4.2 XML

XML [68] is a very generic markup language, designed to store and transport data. The XML format uses *tags* – string names inside angular brackets – to describe data hierarchically.

A simple example[6] is displayed in Listing 2.4. There are multiple benefits of using XML over a plain text format such as the INI file format:

- XML files can have arbitrary nested hierarchy levels and references to describe the data

[5] https://technet.microsoft.com/en-us/library/cc731332.aspx
[6] http://www.w3schools.com/xml

Listing 2.4: XML example file[6]

```xml
<?xml version="1.0" encoding="UTF-8"?>
<note>
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
</note>
```

Listing 2.5: XML Schema example for Listing 2.4[6]

```xml
<xs:element name="note">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="to" type="xs:string"/>
            <xs:element name="from" type="xs:string"/>
            <xs:element name="heading" type="xs:string"/>
            <xs:element name="body" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

- Data can have a type description, allowing also binary data to be used
- The data can be validated using Document Type Declaration (DTD) or XML Schema
- XML provides a query language (XPath) to filter data in a purely functional style.

Since our test plans will be stored in plain text and users might modify the data, validation of the XML structure is essential before parsing it. Due to its structure, XML is very simple to read for humans and to parse for programs. However, practical use of validation of an XML document is limited [68]:

DTDs are not written in XML and thus cannot be validated using a meta type description. Using DTD, character data (strings) cannot be limited to a subset of valid characters, by e.g. specifying a regular expression. Modular type definitions and evolution of thereof are not supported, which makes it difficult to define a larger set of related DTDs.

To tackle these problems (namespaces, modularization, written in XML, . . .), XML Schema has been developed. Nonetheless, XML Schema is generally too complicated [68]. XML Schema does not provide the specification of the XML root element. Further, the element and data declarations are not context-sensitive, thus the schema cannot limit the existence of specific attributes if a specific value is set. Hence, XML Schema fails to describe its own XML syntax.

2.4.3 JSON

JSON [69] is a standardized open format for data exchange derived from the JavaScript syntax. It is heavily used in web technologies for asynchronous communication between browsers and web servers. JSON consists of six data types (see Listing 2.6):

null an empty value.

boolean true and false values.

number decimal, floating point and exponential number representation.

string any UTF-8 encoded string enclosed in quotes.

array an ordered list of values of any JSON type using square brackets notation with commas as separators between elements.

object an unordered set of key-value pairs enclosed in curly braces. The keys are JSON strings, separated from their value by a colon.

Listing 2.6: JSON example

```
{
    "string":"Hello",
    "array":[0, 1, 2],
    "number":2.99792458e8,
    "object":{
        "enabled":false,
        "keys":null
    }
}
```

Through the use of objects and arrays, a structured representation of the encoded data is possible. In many programming languages, the representation can be used to serialize and reconstruct objects with built-in functions like the JavaScript `JSON.parse()` and `JSON.stringify()`.

In contrast to the XML and INI format, the JSON standard does not specify comments. Extensions, such as richer data types like date and time or comments, must be handled by the encoding and decoding side. According to the project website[7], there are JSON libraries available for more than 60 programming languages as of February 2016. Similar to XML Schema, a schema description for the JSON data format is available as working draft [70]. Tools and software libraries for validating JSON data according to a given JSON Schema are also available[8].

2.5 Software Deployment Strategies

Using a high number of microcontrollers in a system requires a clever way to update their firmware. It is not feasible to update the software for each device manually.

2.5.1 Compile-Flash-Cycle

Microcontroller software is typically stored in the flash memory of the embedded device. The binary instructions are written to the memory using the standard programming or debug interfaces (Joint Test Action Group, Serial Wire Debug, Device Access Port), provided on the microcontroller pins. To perform changes, the software first has to be changed and re-compiled. Then, it can be flashed onto the target. However, using the modular test system approach, there will be dozens of microcontrollers present in the environmental chamber where these local programming interfaces would be hard to access. For that reason, a networked solution is required.

2.5.2 Boot Loader

A boot loader is a component that can write data received on a bus interface to the embedded flash of the microcontroller. Reference designs and architectures for serial

[7] `http://json.org`
[8] `http://json-schema.org`

interfaces like UART and CAN exist [52, 71]. Therefore, the boot loader can basically update itself, on the data bus that is already provided into the climate chamber. Using more advanced bus interfaces, the boot loader becomes also more complicated.

The industry standard for deploying operating systems via Ethernet is the Preboot eXecution Environment using Dynamic Host Configuration Protocol (DHCP) and Trivial File Transfer Protocol infrastructure. However, microcontrollers are easily overloaded by the overhead of providing these features.

2.6 Chapter Summary

In this chapter, an introduction to several industrial communication possibilities has been given. For the first prototype implementation, CAN has been selected because of its simplicity. The CAN frames are sent event triggered with a priority value. Thus, important status information can still be delivered timely. To improve the data rate, the second prototype uses Ethernet. Thereby, a larger number of bus participants can be addressed by using Ethernet switches. The real-time EtherCAT implementation is of limited use for this modular test system. The main problem is that EtherCAT (like most of the time-triggered communication protocols) require a preset slave list. It does not allow dynamically adding and removing slave devices – which is a major requirement for this system architecture.

The control module must be capable of running at high ambient temperatures (up to 125 °C). Instead of microprocessors, microcontrollers have been chosen for implementing the control and measurement modules, due to their flexibility and availability for the automotive temperature range.

The inclusion of an interpreted scripting language within the microcontroller firmware has been decided in favor of Lua. The smaller memory usage and the simple C-API support the decision. The use of the LabVIEW programming environment and the Graphviz modeling language for the host software has been explained.

Further, various test configuration file formats have been described. Due to the availability of the JSON for numerous programming languages, JSON will be the data exchange format for this project.

In the beginning of the project, microcontroller development will start using the classic compile-flash cycle. In the later stages – reaching beta release versions – the use of boot loader mechanisms for deploying larger quantities of microcontrollers in the climate chamber is planned.

3

The MoPS Distributed System

Divide each difficulty into as many parts as is feasible and necessary to resolve it.

(René Descartes)

Contents

THIS thesis introduces a new concept of reliability testing for power semiconductors – Modular Power Stress (MoPS). MoPS has the goal to provide a flexible infrastructure for customizable stress test applications. Such a system is then capable of running various test applications with a common base system architecture. The test engineers and test operators only need to learn a simple comprehensive system, in contrast to the variety of different test system architectures that already exist.

To handle the complex requirements, the proposed test system is split up into smaller parts. Since the *device subsystems* also contain intelligent processing elements, we call the MoPS test system a *distributed test system* [19]. Having powerful controllers available, the real-time control and communication effort on the host computer can be reduced significantly.

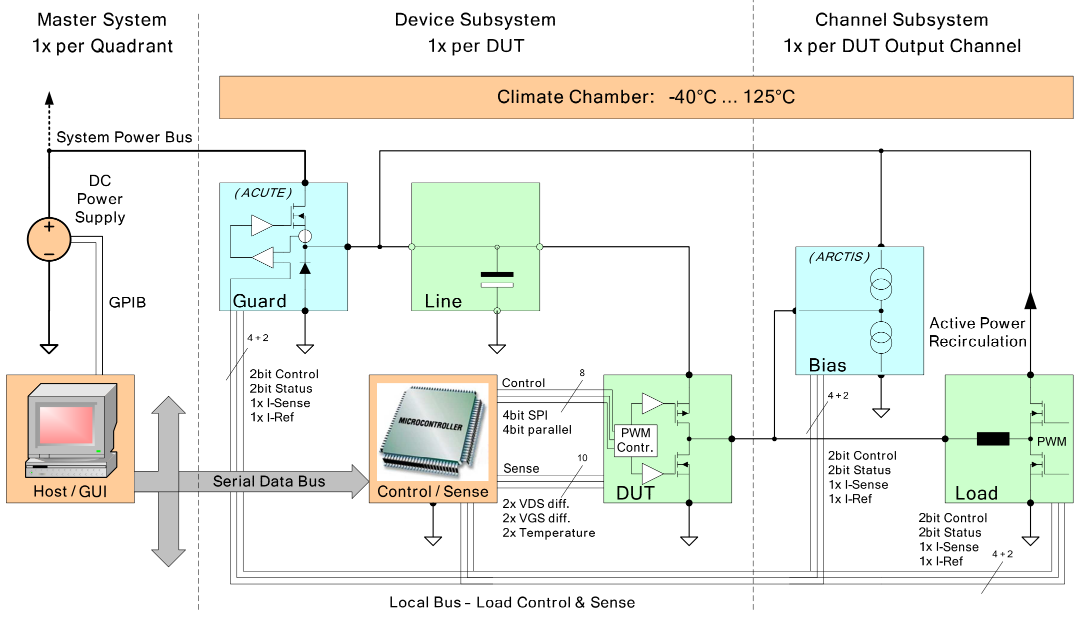


Figure 3.1: Modular Power Stress test Architecture [19]

The main components of such a modular architecture are given in Figure 3.1.

Master System The centralized host is responsible for controlling external devices such as voltage supplies or electronic loads. It is further capable of loading the test procedure, transferring it to the *distributed execution and measurement nodes* and managing the test.

Device Subsystem Each of these distributed nodes is the central heart of the respective device subsystem. The controller does not only apply stress patterns to the DUT, but also manages the surrounding hardware periphery via the local bus. Various modules – such as protection, bias and load – are connected to this serial and parallel, mixed-signal bus.

Serial Data Bus The node interacts with the master system via a *serial data bus* to receive the test plan, management commands and report back status and measurement results. Since this bus concerns the outside communication of a single node, it is called *global bus*.

By providing such an architecture, the main requirements are met:

- Each DUT is situated very close to the respective stress and measurement controller. Power electric circuits (i.e. the modules) are close to the DUT to reduce interference.
- Only very few cables – namely the power bus to supply the DUTs and the data bus to supply the controller with the test configuration – penetrate the climate chamber casing.
- The devices can still be individually protected as in previous test system generations. Further, individual control provides the possibility to stress the devices under different operating conditions in a simple way.

However, since there is no single central instance to execute the test, it is a significant effort to create the test code and configure this distributed test system architecture. Chapter 4 explains the details how to configure the system and Chapter 5 describes the steps of executing a custom test plan.

This chapter describes the host layer in Section 3.1, the communication channel in Section 3.2 and the executing hardware subsystem (i.e. *hardware targets*) in Section 3.3. The components used in the microcontroller firmware are described in Section 3.4. A short overview of various periphery modules is given in Section 3.5. Parts of this chapter have been published in a paper at the 11th IEEE International Conference on Industrial Informatics in 2013 [19].

3.1 Host Layer

While the life test is executed mostly on the distributed controller nodes, the combined test system still requires a central management unit. The tasks of this central instance include

- to provide a graphical user interface to the operator. The test system will be operated in a lab environment and important information must be presented visually to the users in a comprehensible way.

- to load the test definitions and transferring the test recipes to the executing controller nodes. The test plan is stored in files for archiving, documentation and comparison purposes. The test system must read these files and distribute the test to the target hardware.

- to store information about and results from the tested DUTs as well as the measuring node. Analog quantities such as voltage, current and temperatures are continuously recorded and must be preserved for later analysis.

- to access external hardware that the controllers cannot interface directly. Since multiple microcontroller nodes will use the same Power Supply Unit (PSU), they must be centrally controlled.

- to sequence and synchronize multiple controller nodes. Although real-time requirements on the communication channel are relaxed, there exist several situations, where sequencing of test channels is important: A power supply might be used for several DUTs simultaneously, but the devices cannot be turned on at the same time (e.g. due to high start-up power requirements). In this case, the host program will interact with the nodes and control the externally connected devices.

Laboratory instruments like power supplies, electronic loads, oscilloscopes, are often required in power electronic life tests. These instruments are typically connected via parallel bus (GPIB) or serial buses (RS-232, USB, Ethernet). Modern devices already feature Ethernet based higher level protocols such as LXI. Due to cost and space restrictions, multiple DUTs are connected to a single instrument. Therefore, the host layer has to control these devices accordingly.

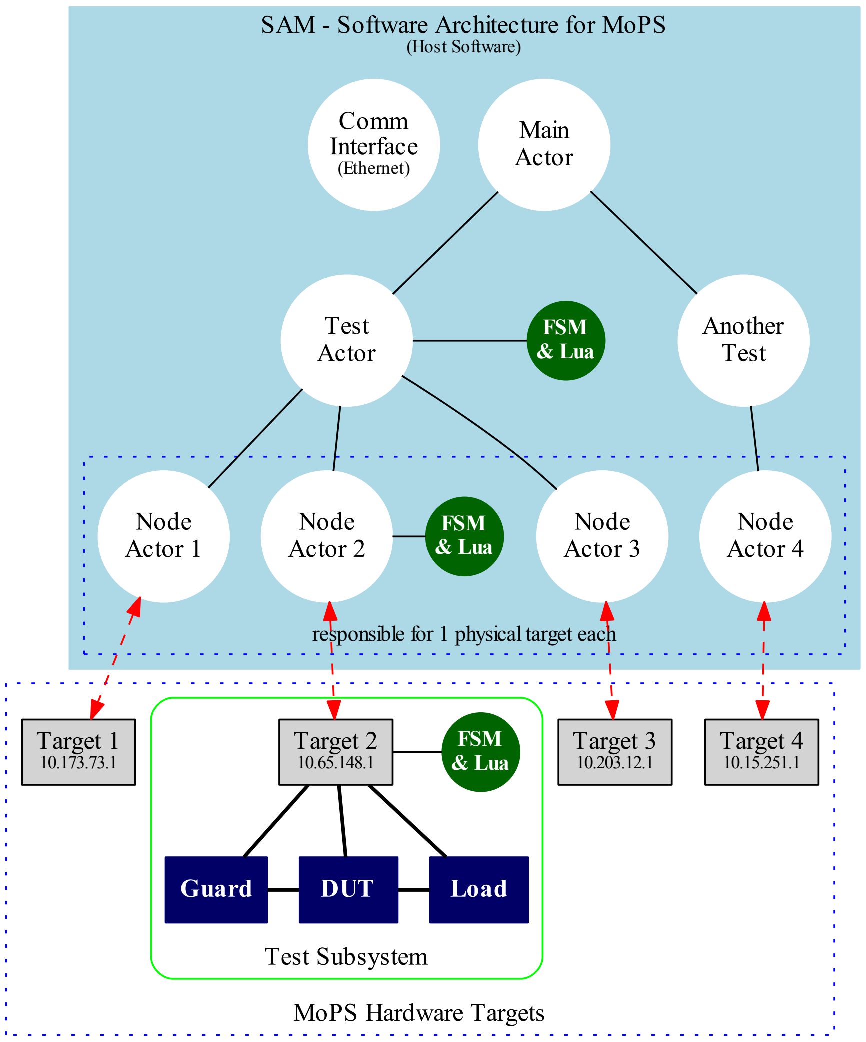


Figure 3.2: Software Architecture for MoPS [72]

3.1.1 Software Architecture for MoPS

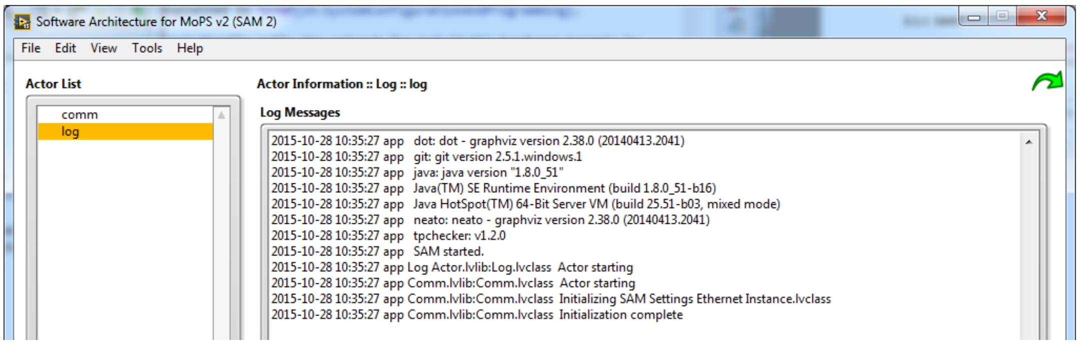
Software Architecture for MoPS (SAM) is a specific implementation of the host layer, developed using the LabVIEW programming environment. The *Actor Framework* (see Section 2.3.2) is used to provide an extensible and reconfigurable software architecture. Each Actor may either present a GUI display to the user or run in the background to handle a specific task. Upon interacting with the GUI, messages are sent to the Actor core handler to update the model and further refresh the view.

SAM is organized in a hierarchical structure. Figure 3.2 gives an overview of the host software. The *Main Actor* reads the system configuration and starts the

Communication Interface (comm) and the *Log Actor* (log). The details of the system configuration are explained in Chapter 4.

When the SAM software is started, it periodically polls the network for MoPS hardware targets by broadcasting the `NODEINFO` command (see Section 3.2). Every available and compatible target will report back its status. Thereby, SAM recognizes new hardware targets and the *Main Actor* starts a dedicated *Node Actor* to handle the communication and test execution for each specific target. Thus, each executing node has a virtual representative within the host software that acts as a proxy. In a similar way, each external instrument gets an instance of the *Instrument Actor* assigned, which makes it possible for SAM to interact with the device.

Upon loading a test plan file, a *Test Actor* is started. It is the main responsible agent for running a test on the test system. Within the test plan, a number of targets and the DUTs to be tested are described in the oven plan as well as the required instruments. The Test Actor, hence, assigns the requested Node Actors and Instrument Actors and reserves their usage. An actor may only be reserved by one test, as it may only execute one test at the same time. Furthermore, the Test Actor may request the usage of connected power supplies, as specified in the oven plan.


Software Architecture for MoPS v2 (SAM 2)
File
Edit
View
Tools
Help
Actor List
comm
log
Actor Information :: Log :: log
Log Messages

```
2015-10-28 10:35:27 app   dot: dot - graphviz version 2.38.0 (20140413.2041)
2015-10-28 10:35:27 app   git: git version 2.5.1.windows.1
2015-10-28 10:35:27 app   java: java version "1.8.0_51"
2015-10-28 10:35:27 app   Java(TM) SE Runtime Environment (build 1.8.0_51-b16)
2015-10-28 10:35:27 app   Java HotSpot(TM) 64-Bit Server VM (build 25.51-b03, mixed mode)
2015-10-28 10:35:27 app   neato: neato - graphviz version 2.38.0 (20140413.2041)
2015-10-28 10:35:27 app   tpchecker: v1.2.0
2015-10-28 10:35:27 app   SAM started.
2015-10-28 10:35:27 app Log Actor.lvlib:Log.lvclass  Actor starting
2015-10-28 10:35:27 app Comm.lvlib:Comm.lvclass  Actor starting
2015-10-28 10:35:27 app Comm.lvlib:Comm.lvclass  Initializing SAM Settings Ethernet Instance.lvclass
2015-10-28 10:35:27 app Comm.lvlib:Comm.lvclass  Initialization complete
```

Figure 3.3: Software Architecture for MoPS GUI

Main Actor During the initialization of the application, the configuration file is read and the main actor is started. Required external software tools are checked and their versions are reported during the start-up of the software. The Main Actor then starts the log facility and initializes the communication interface. Figure 3.3 shows the main GUI. The panel is divided into two parts. The left hand side displays a tree, where the child actors are listed, and the right hand side is a container to display the child actors' GUI. Within the tree, the child actors are categorized. System Actors, such as the Communication (*comm*) and the Log Actor (*log*), are displayed in the root of the tree. Unassigned Nodes are summarized in the *Unassigned* category. Each Test Actor defines its own category. Requested nodes and external devices are displayed as children of the Test Actor.

Log Actor The Log Actor can print messages it receives to a graphical display and also write the messages to a text file or a database. The selection of the output format is done in the configuration file. A separate tool is available for displaying and filtering the stored messages during an active test as well as offline for debugging purposes [73].

Communication Interface The Communication Interface deals with sending messages to and receiving from the hardware targets via the *global bus* interface. SAM deals with the node messages in an abstract way. Details about the message format are given in Section 3.2. The communication interface translates the address and payload information into a suitable packet to be transmitted. For the curious user, statistics like number of messages and total transmitted and received Bytes are collected and displayed.

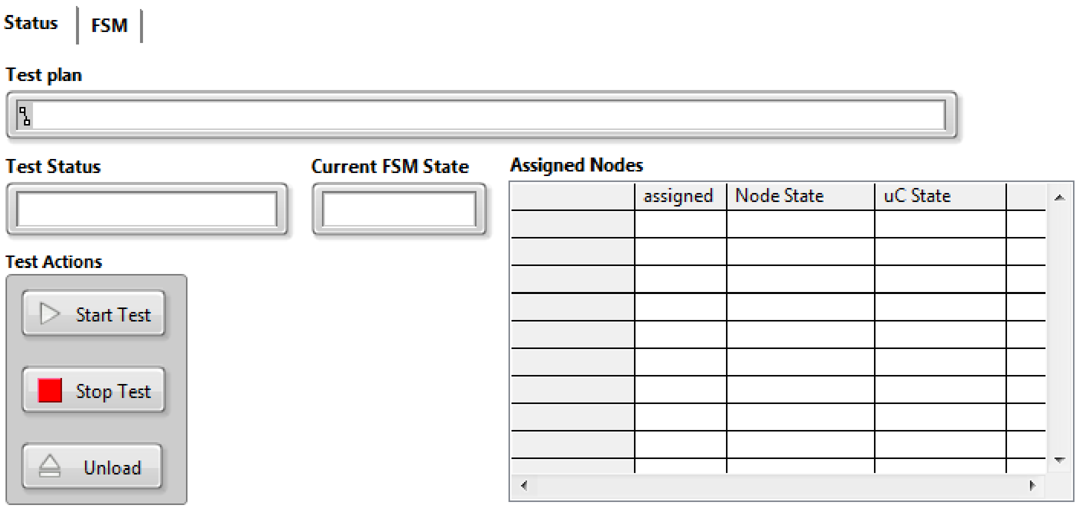


Figure 3.4: SAM Test Actor

Test Actor Upon loading a test plan, the read data is first verified by the external Test Plan Checker tool (see Section 4.3.2). Then, the requested hardware targets and external instruments are read from the oven plan and requested from the Main Actor. If all requested agents are available, they are assigned to the specific test. Finally, the Test Actor is started by SAM and its GUI panel is added to SAM (Figure 3.4). To simplify the look-up of the assignment for the user, the tree control is rearranged as previously described. Each Test Actor may also execute the state machine given in the test plan. An instance of the *Lua Actor* is started too. The Test Actor displays the test status to the user and provides the means to start and stop the test. Further, the FSM diagram of the test procedure (see Section 4.2) is visualized by drawing the image (see Section 5.2.3). The host software is capable of loading and running multiple tests concurrently.

Node Actor An instance of the Node Actor is responsible for managing a single hardware target. Therefore, an instance is created for each recognized target on the network managed by SAM. The GUI of the Node Actor displays information about the node and the test. Since both the Node Actor and the hardware target run an FSM, the Node Actor displays an image for each state machine. Measurement data acquired by the hardware target may also be viewed in this actor.

Lua Actor To decouple executing a loaded FSM from the Node and Test Actors, a separate background actor has been created. This *Lua Actor* is the same for both test and node parents, except for some differences in the provided API. Details about the implementation of the Lua engine and the operation with LabVIEW are given in Section 5.2.1.

Instrument Actor For each connected external instrument, a resource actor is created [74]. The actor itself does not care about the type of the instrument as it loads the required drivers depending on the SAM configuration. Thus, it abstracts the interface to the instrument to make it possible to change the instrument without rewriting the test code.

3.1.2 MoPS Tiny Host

In addition to the test system software, a small application has been created to send messages to the hardware targets (Figure 3.5). This so-called *Tiny Host* tool is used to test the communication interface. It provides an interface to send messages to a hardware target manually and to execute commands in the Lua interpreter running on the microcontroller. Further, Lua script files can be sent to the target and executed. These Lua scripts are processed without the need of an FSM description. Therefore, the laboratory evaluations of a new piece of hardware can be done without a full test plan. Due to its simplicity, hardware events cannot be evaluated using the *Tiny Host*. Furthermore, the *Tiny Host* does not feature any safety checks or instrument control.

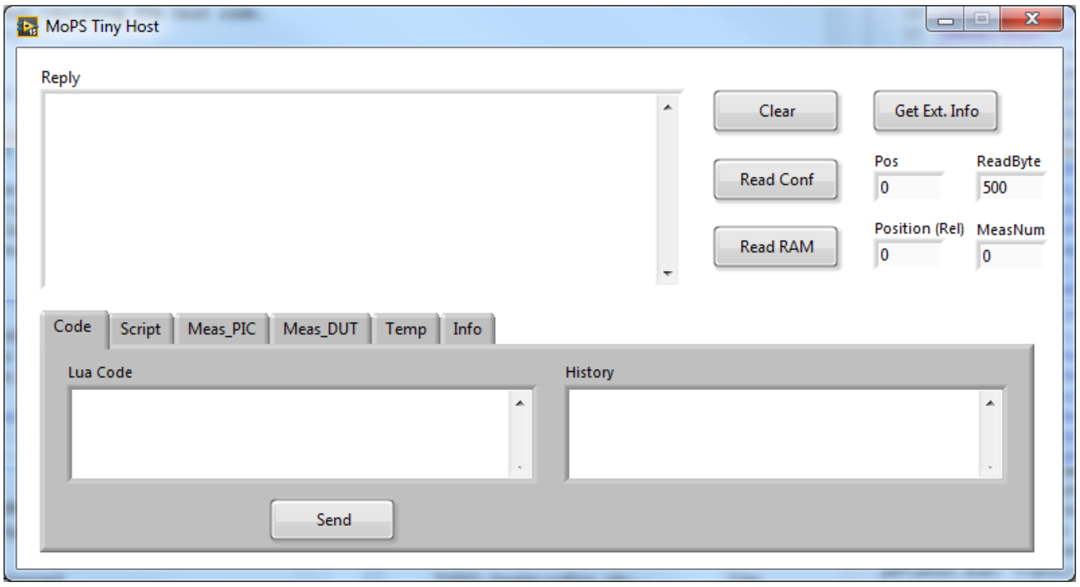


Figure 3.5: MoPS Tiny Host

3.2 Communication Channel

A serial data interface to each test node is required to reduce the amount of cables. The interface is used to connect the nodes and let them receive stress test patterns as well as report device status to a central data storage.

3.2.1 Selection

The communication interface is selected according to several requirements [19]. It should be possible to send single messages for configuring the controller nodes as well as receive parameter measurements at relaxed time intervals (about 1 kB/s to 10 kB/s per DUT). Optionally, device waveforms are requested at dedicated time intervals (e.g. on device failure). Further, broadcast messages are desired in order to notify all bus members at the same time instant. In addition, the communication interface ideally provides synchronization features in order to sequence multiple controller nodes. When high power DUTs are tested, the power output of the supplies may be limited. Therefore, the DUTs need to be tested intermittently and the controllers need to be synchronized.

The interface also needs to withstand a harsh environment in the test system laboratory while based on standard microcontroller periphery modules in order to use commercially available hardware. Further, hot plug capability of the nodes is desired, so the test operation does not need to be interrupted when a single node is replaced.

Finally, the communication interface should be able to be configured and programmed in a simple way. The used protocol stack should have little performance impact on the controller node.

Among possible candidates, CAN (see Section 2.1.2) and switched Ethernet (see Section 2.1.3) fit well to the requirements. Ethernet provides high data rates but requires advanced microcontrollers with built-in medium access control unit to fully utilize the increased bandwidth. The advantages of CAN are that many automotive controllers feature built-in CAN modules and our expected small message sizes keep the bus traffic at a reasonably low level.

3.2.2 CAN-based Interface

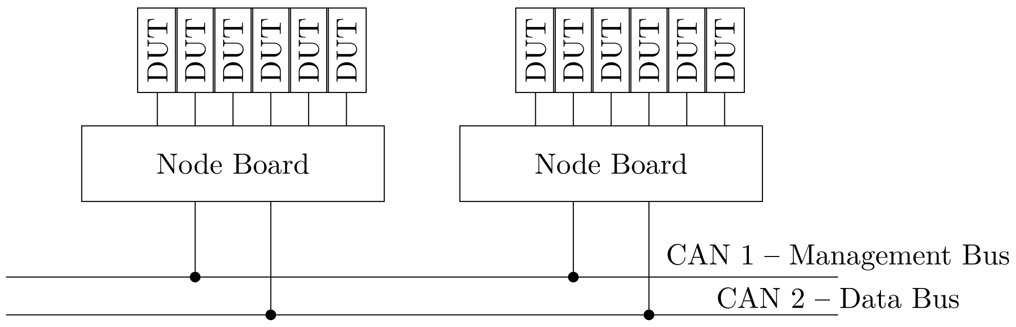


Figure 3.6: MoPS Setup using CAN Interface

The first version of SAM was implemented together with a CAN communication interface [20]. Figure 3.6 shows the basic setup. The communication is split up among two CAN interfaces, according to the principle of different interfaces for different tasks [75]. In this implementation, the interfaces *Diagnostic and Maintenance* (called “Management Bus”) and *Configuration and Planning* (called “Data Bus”) are used. Therefore, it is possible to separate the diagnostic traffic required for status information of the node and the test procedure from reading the measurement data.

Extended CAN message types as shown in Figure 3.7 are used, where the 29 bit identification part is further divided into four fields: the four most significant bits are used to describe the frame type, which provides the possibility to increase priority for global system messages. Further, they are used to distinguish between node-to-host

28 25	24 20	19 14	13 0
Frame Type 4 bit	Command 5 bit	Node Address 6 bit	Offset 14 bit

Figure 3.7: Customized CAN ID format

and host-to-node messages. A special frame type is also used for baptize messages. The 5 bit command part is used to describe the intention of the message. Then, the 6 bit assigned address of the bus participant follows. Thus, up to 63 controller nodes can be connected, where the special address "0" is reserved for the host. Finally, a 14 bit file system offset is used to provide the address for writing and reading data to either the configuration or the measurement memory range of the smart controller node. Specifying the address offset in the message identifier enables the use of all eight data bytes in the message payload area. Not all possible bit combinations of the frame type and command fields are used, so further extensions can be applied without changing the presented arrangement.

A summary of the commands supported by the first version using CAN communication is given below:

REQBPT (`0x01`) *Request Baptize* – CAN does not intrinsically support addresses of the bus participants. Therefore, upon booting, the node reads its internal unique ID and builds a special baptize frame. This frame contains part of the unique ID in the offset field in order to reduce the chance for collisions. The full unique ID as well as the software version of the node is transmitted in the data part of the CAN frame. After sending the request frame to the host, the node waits for the assignment of a short address. This mechanism is borrowed from the TTP/A protocol [76].

ASSBPT (`0x02`) *Assign Baptize* – The host receives the full unique ID of the node and retrieves a short address from the node look-up table. The same short address is assigned for the same node. The assignment frame is accepted by all nodes who do not have a short address yet. Upon reception of this command, the unique ID in the frame data part is compared to the internal ID by each of the nodes, to check whether the broadcast frame is meant for the specific node.

NODEINFO (`0x03`) *Node Info* – The host may query a node about its information. The node replies with the following information: its hardware version, its software version, its test type and its test software state.

WRTCONF (`0x04`) *Write Node Configuration* – The host transfers configuration data to the node based on the offset given in the CAN message ID. Writing the configuration is only possible when the test software is in the *IDLE* state. Up to 8 B of data can be written.

READCONF (`0x05`) *Read Node Configuration* – The host requests reading the configuration memory. The addressed node responds with 8 B of data from the given offset.

READMEAS (`0x06`) *Read Measurement Data* – The host requests reading the separate measurement memory. The node responds with 8 B of data from the given offset.

READCRC (`0x07`) *Request Configuration CRC* – The host tells the node that it has finished transmitting the configuration data and requests the CRC computation of the configuration memory. The node will reply with the computed CRC value, so that the host knows whether or not all data has been written correctly.

INITHW (`0x08`) *Initialize Hardware* – In order not to interfere with the test hardware, the test node's periphery is set to a tri-state input in *IDLE* state. When the configuration has been written correctly, the host sends the initialization command with the expected CRC value. Upon reception, the node will configure its hardware interfaces according to the previously transmitted configuration. The node is now ready for testing.

DEINITHW (`0x09`) *De-initialize Hardware* – The host tells the node to deactivate its hardware interfaces and clear the configuration memory. This command can only be requested for initialized or stopped tests.

RUNTEST (`0x0A`) *Start Test Procedure* – After proper hardware initialization, the testing procedure specified in the configuration may be started.

STOPTEST (`0x0B`) *Stop Test Procedure* – A running test may be requested to stop any time. The node will finish its current task and stop the test immediately afterwards.

TESTSTATUS (`0x0C`) *Test Status* – While NODEINFO returns information about the testing node, TESTSTATUS is used to report the status of the test, like the number of DUTs alive.

BUTTON (`0x0D`) *Push-Button Request* – During the test setup, it may be important to know the position or address of a particular node. The test node may provide a push button for the operator, which when pressed will inform the host software about this event. Therefore, the individual node can be identified.

IDENTIFY (`0x0E`) *Identify Request* – Similar to the BUTTON event, the operator may find a specific node among several connected nodes. The IDENTIFY message flashes the selected nodes' Light Emitting Diodes (LEDs) in order to visually find the node.

SELFTEST (`0x0F`) *Self-test Data* – Like writing and reading the configuration memory, the nodes' self-test memory can also be read.

In a prototype setup for a device qualification test, reasonable transmission rates for the configuration phase could be achieved. In particular, an improved HTOL system was developed [19]. The final setup consists of 20 microcontroller units, each of them operating multiple DUTs. The microcontrollers receive their test configuration via the CAN interface in less than 1.5 s. Since the microcontrollers operate autonomously, downloading this configuration is not time-critical and usually done only once per qualification test that runs for 1000 h. Two status information messages are polled once every second for each connected node. Worst-case estimations for the maximum allowed number of 63 nodes show less than 5 % bus load for the status and diagnostic messages.

However, measurement data acquired by the microcontroller cannot be transferred frequently. In the prototype test application, we had to reduce the transmitted payload size to about 8.6 kB. Due to limitations of the CAN driver on the host, reading the data takes in average 6 s for a single microcontroller node. Thus, measurements of 63 nodes can be read by the host software every 6.5 min. This time interval although comparably large is sufficient, since measurements are stored only every 15 min to 30 min in a life test setup running for 1000 h. Nonetheless, when reading larger sets of measurement data (in the order of a few MB), this limitation becomes a problem. To solve this issue, we moved from CAN bus to switched Ethernet to increase the data throughput.

3.2.3 Ethernet-based Interface

The availability of automotive microcontrollers with integrated Ethernet MAC enables the use of a faster communication interface. Consequently, support for Ethernet communication between the hardware nodes and the host computer has been added in the MoPS system.

UDP on top of the IP is used for the MoPS test system architecture. UDP, a connectionless communication mode, is very lightweight compared to the well-known

TCP. To enable secure communication, Datagram Transport Layer Security (DTLS), which is similar to Transport Layer Security for TCP connections, is available for UDP connections [77]. Using Ethernet implies several requirements for the test node. These being that for the data link layer (MAC) and the network layer (IP), unique addresses within the network are required.

Setting the MAC address

It is possible to assign locally administered MAC addresses for Ethernet (see Section 2.1.3). Setting the specific bit allows to freely choose the remaining bits while not interfering with statically assigned global MAC addresses. As it would be very tedious to assign the addresses manually, they are computed on per chip basis. The used microcontrollers (XMC4500) feature an integrated 128 bit unique ID. These 128 bit can be reduced using the 32 bit Ethernet CRC function and the MAC address can be set before starting up.

Setting the IP address

Obtaining an IP address is usually done using a centralized DHCP server. Due to restrictions in the lab environment, running a second DHCP server software on the host system in addition to the standard office network environment is not allowed. The risk of erroneously connecting the network interface used for lab operations to the office network and, therefore, interfering with the infrastructure is too high. Further, it is not desired to connect the test nodes to the office LAN, since it would allow any user to access the test nodes and interfere with the test. Even when using encrypted communication channels, the rather weak microcontrollers may easily be overloaded by Denial-of-Service (DoS) attacks. Thus, it is first required for the host computer to have separate network interface cards for the office LAN and the test system LAN. Secondly, static IP addresses must be assigned.

Since assigning the MAC address is done automatically, a similar procedure is used for the IP address. In order to reduce the possibility of address collisions, the maximum possible private address space is selected. The class-A network `10.0.0.0/8` provides a private use network with up to 2^{24} hosts [78]. The first and the last address within such a Classless Inter-Domain Routing network always have some special functionality assigned. Thus, not all possible addresses can be used for individual nodes. Furthermore, the host computer and possibly networked laboratory

Table 3.1: Example Node IP addresses

Device	Unique ID	MAC address	IP address
Host	n.a.	globally assigned	10.0.0.250
Node 1	01810206 05e0014f 82060010 0a000000	02-00-ab-5b-23-94	10.211.174.1
Node 2	01810c06 06e0014f 82060010 0a000000	02-00-7f-a5-07-24	10.47.101.1

instruments must be connected on the same network. For that reason, a 16 bit CRC is computed from the previously mentioned 128 bit unique ID. The final IP address is then composed in the following way: `10.x.y.z`, with `x` and `y` being the higher and lower byte of the CRC result respectively. The last octet `z` is “1” for test nodes and different for the host computer and other Ethernet based laboratory equipment. Examples are provided in Table 3.1.

In contrast to CAN, Ethernet does not use a specialized identifier field. The header fields contain the addresses for proper delivery to the targeted node. In order to send instructions to the node, the data area of the frame has to be used. The payload field is large enough to hold up to 1.5 kB. Even though the used lwIP TCP/IP stack [79] can deal with fragmentation[1], the performance hit on the microcontroller is not acceptable. Therefore, using IP, up to 576 bytes IP data (up to 512 bytes UDP data) can be safely transmitted, before fragmentation and reassembly has to be taken care of [80]. The MoPS command format for Ethernet based communication is given below. The size of the MoPS data field is limited by the size of the MoPS header.

Table 3.2: MoPS Ethernet message fields

Bytes	Description
0 .. 2	Packet Type – The packet type field is a 3-character string. Currently, `CMD` or `LOG` are possible. `CMD` is used for message from and to the testing node. `LOG` messages are only sent from the node as debug and informational messages.
3	Delimiter Byte – The colon “:” is used as 1 B delimiter sequence.
4 .. 11	Time – Absolute UNIX time stamp. The 32 bit number is printed as eight hexadecimal characters.
12	Delimiter Byte

[1] `http://lists.gnu.org/archive/html/lwip-users/2010-01/msg00094.html`

Table 3.2: MoPS Ethernet message fields

Bytes	Description
13 .. 20	Fractional Second – Fractional seconds of the time stamp, also printed as eight hexadecimal characters. The fractional part is given as number of increments of the CPU clock of the current target microcontroller.
21	Delimiter Byte
22 .. 31	Packet Number – The message sequence number is printed as variable width decimal number.
32	Delimiter Byte
33 .. 41	Command – Depending on the packet type, this field inserts a sub command compared to the original CAN message commands. The sub command is a human readable printed string.
42	Delimiter Byte
43 .. 52	Offset – The optional offset field is used to specify the memory location for reading and writing to different memory sections. The value can be given as either decimal or hexadecimal value prefixed with `0x`.
53	Delimiter Byte
54 .. 511	Data – Up to 458 byte data can be included in a MoPS command message. The data part is preceded by a delimiter byte. As the data field is the last field and the length is known by the UDP and IP length fields, binary as well as string data can be transmitted.
0 .. 511	UDP Payload – As the header is variably sized, a maximum size of 54 byte has to be considered. Including the 458 byte data part, the UDP packet does not exceed the maximum size.

Compared to the CAN communication, the addresses for the Ethernet based communication are computed from the internal unique serial number of the microcontroller chip. Therefore, the baptizing commands (**REQBPT** and **ASSBPT**) are obsolete. Further, the **BUTTON** command is not used anymore, because any test related event can be sent using the newly provided **EVENT** command.

The remaining commands have largely the same functionality. However, new commands are introduced: **EVENT**, **PUBLISH**, **PRINT**, and **LOGLVL**. The following list explains the commands.

NODEINFO *Node Info* – The host may query the node about its information. The node replies with the following information: version information about its used hardware and software, its software state; time and memory information; and its Ethernet address information (unique ID, MAC address and IP address).

WRTCONF *Write Node Configuration* – The host transfers configuration data to the node based on the offset given after the MoPS command. Writing configuration is only possible, when the test software is in the *CONFIG* mode.

READCONF *Read Node Configuration* – The host requests reading the node's configuration memory. When the target node is in the *CONFIG* mode and there is no node configuration being written, the Electronic Data Sheet (EDS) can be read from this memory location (see Section 3.4.4).

READMEAS *Read Measurement Data* – The host requests reading the separate measurement memory. Measurement data is written from various hardware sources (Analog-to-Digital Conversion (ADC), Delta-Sigma Demodulator (DSD), communication interfaces) via DMA controllers into the separate memory.

READCRC *Request Configuration CRC* – The host tells the node that it has finished transmitting configuration data and requests CRC computation of the configuration memory. The node will reply with the computed CRC value, so the host knows if all data has been written correctly.

INITHW *Initialize Hardware* – In order not to interfere with the test hardware, the test node's periphery in the *CONFIG* mode is set to a tri-state input. When the configuration has been written correctly, the host sends the initialization command with the expected CRC value. Upon reception, the node will parse the data in the configuration memory and initialize the FSM (see Section 4.2.3). The node is now ready for testing.

DEINITHW *De-initialize Hardware* – The host tells the node to return from the *TEST* mode to the *CONFIG* mode. In order to perform this command, the test FSM must be in the **IDLE** state. Therefore, this command can only be requested for initialized or stopped tests. The configuration memory is cleared and filled with the EDS data.

STARTTEST *Start Test Procedure* – After proper initialization of the FSM, the test procedure specified in the configuration may be started. This command enables processing of events in the test program FSM.

STOPTEST *Stop Test Procedure* – A running test may be requested to stop at any time. The node sets the *stop* event for the test program's FSM. It is the responsibility of the test plan designer to ensure, a test can be stopped. This requirement is checked by the Test Plan Builder (TP-Builder) software, before the test plan is loaded onto the target node (see Section 4.3.2).

TESTINFO *Test Status* – While NODEINFO returns information about the testing node, TESTINFO is used to report the status about the test. The current FSM state and the full event table is sent to the host software.

IDENTIFY *Identify Request* – The IDENTIFY message flashes the selected nodes' LEDs in order to visually find the node.

EVENT *Set FSM event* – The test node runs a reconfigurable FSM (see Section 4.2.1). This command sets an event for the FSM.

PUBLISH *Publish measurement memory location* – In order for the host software to know which memory area is dynamically allocated by different modules, the location information is published (see Section 5.2.4).

PRINT *Print log message* – As the Ethernet communication channel has sufficient throughput, log messages from the test FSM and the Lua interpreter can be sent to the host software for further processing and storage.

LOGLVL *Set log level* – Using this command, the log verbosity of the hardware target can be set. The following levels are available:

LUA FSM Lua code print or error message

SEVERE Severe errors that require program exit (e.g., the application ran out of memory).

ERROR Error messages that cannot be recovered, but the program can continue to run.

WARN Recoverable problem that you should be notified about (e.g., invalid value in a configuration file, the default is used).

INFO Informational messages.

ENTRY Log entry and exit to all functions.

PARAM Log entry and exit to all functions with parameters passed and values returned (including global effects if any).

DEBUG General debugging messages, which is useful information that can be output on a single line.

Using the Ethernet based communication interface, the measurement transfer data rate is improved by a factor of 200 over the CAN based interface. Transferring 4 MB of raw data to the host takes about 14 s, which is about 285.714 kB/s. It is not favorable to ask the hardware target to send all of the data at the same time, as it would overload the controller (the microcontroller will not be able to handle the test procedure in the meanwhile) or the host (if multiple microcontrollers send all their data at the same time). For that reason, measurement data is queried from the host application one packet after the other. Reading the data from the microcontroller requires both the request message and the reply message, thus adding a loop delay.

3.3 Distributed Control & Sense Node

The third foundation of the Modular Power Stress test system is the distributed control node. Its main purpose is to run the test procedure in proximity to the tested device or application. Thus, the controller has to receive the test configuration, parse and execute it. Further, status of the tested application and various analog signals need to be monitored.

The decision to use automotive microcontrollers that support the required temperature range was heavily influenced by the fact that microcontrollers – in contrast to FPGAs – possess various readily-available peripheral modules like communication and analog interfaces. The specific controller to be used was selected based on the evaluation in three preliminary studies:

The first version of a local stress test microcontroller was called *SmartMoPS*, as it is used to test Smart Power Switches (SPS) (see Section 3.3.1). In the meantime, the investigation of the communication channel, as mentioned above, was already ongoing. Therefore, a separate piece of hardware, the *HTOL Node Board*, was created and investigated (see Section 3.3.2). Concurrently, the newly released, high-performance automotive microcontroller series cross-market Microcontroller (XMC)[2], using the

[2]https://www.infineon.com/xmc

32 bit ARM Cortex-M4F core, was evaluated in a DC-converter test application (see Section 3.3.3).

The findings of these projects lead to the miniaturization of the Node Board yielding a very compact and capable distributed test node, the *MicroMoPS*. The hardware and software features are explained in Section 3.3.4.

3.3.1 SmartMoPS

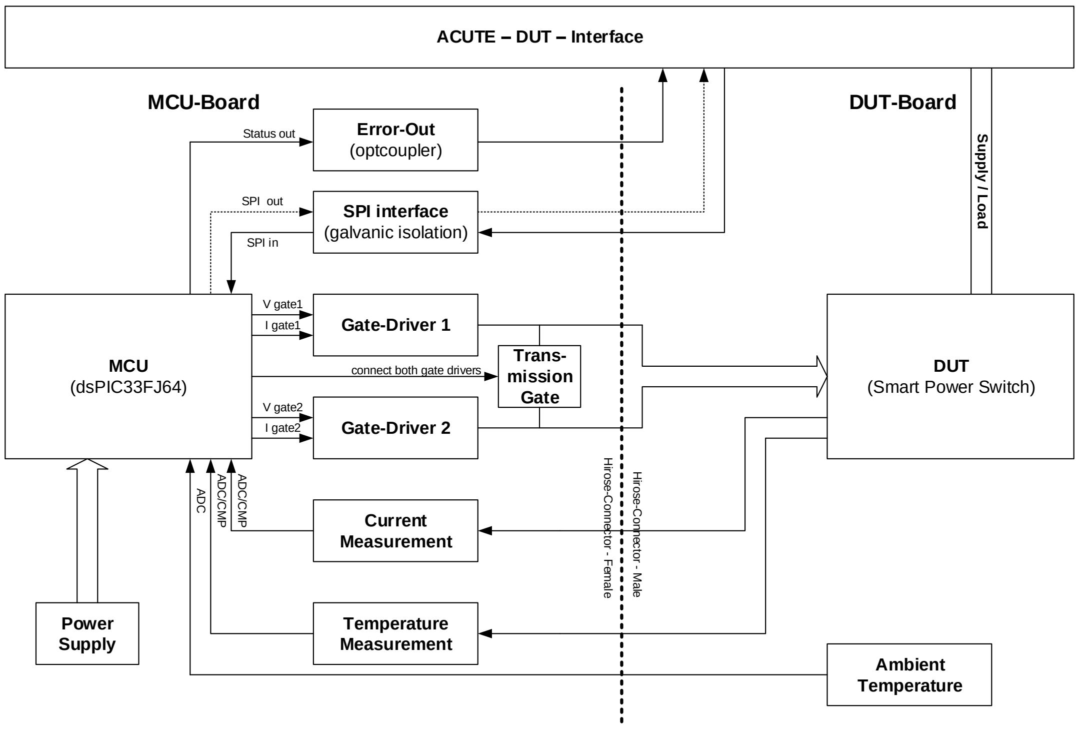


Figure 3.8: SmartMoPS – Intelligent Test Substrate

The *SmartMoPS* (see Figure 3.8) is a small, 2-channel, 8 bit digitally controlled gate driver on a 50 mm × 60 mm substrate PCB for testing automotive SPS within the ACUTE system [22, 81]. The two gate drivers can each source and sink variable gate current with 16 programmable levels (4 bit resolution). The SmartMoPS is used to test various stress conditions on the power transistor test structures, before the actual on-chip gate driver is finalized. The board is a modular extension that can be plugged into the main DUT carrier board, which is used for conventional ACUTE testing. Up to 4 × 64 DUTs are tested in parallel in this test setup. The *SmartMoPS* features a 16 bit microcontroller using the SPI interface to receive the configuration and to send back measurement and status information. Temperature, as well as voltages and currents of the DUT can be measured. The real-time control unit of the ACUTE system allows only sending 21 SPI commands per pulse cycle. Each tested device can be individually addressed within the test system. However, no SPI return channel is available in the climate chamber. Thus, the SPI protocol enables a limited set of arbitrary waveforms to be applied to the DUT. Nevertheless, the analog measurement data is used for the self-test procedure, gate driver calibration and verification.

Operating the microcontroller in the oven at ambient temperatures up to 125 °C works very well. As stated before, the existing test system does not feature SPI return channels, thus the SmartMoPS measurement capabilities are of limited use. The DUT performance is still monitored and evaluated using the ACUTE system, which features one voltage and one current measurement per tested channel. The SPI protocol is very robust, even in the harsh environment where high load currents up to 200 A and enormous current slew rates in the order of 1 A/µs are easily reached in proximity to the digital control unit.

3.3.2 HTOL Node Board

The HTOL node is used to control a qualification test in an already existing climate chamber infrastructure. Within the climate chamber, carrier boards with the tested devices are located. The signal and control interface is located at the backside of the climate chamber [20]. Therefore, it is not subjected to the full automotive ambient temperature range. The node board has also been used to implement and verify the first version of the communication interface (see Section 3.2.2). Figure 3.9 shows the block diagram of the HTOL node board. It features a 16 bit automotive microcontroller to provide the required CAN communication interface. The supply interface is used for connecting the power supplies and the CAN interface. The connection from the node board to the carrier board, which contains up to 8 DUTs, is provided via the signal interface. These signals include DUT supply voltages, digital control patterns as well as digital and analog sense lines to monitor the DUTs.

Further, the guard and driver modules are implemented on the same PCB. The guard module measures the DUT supply voltages and currents. The power supplies are controlled via the host system and shared by all node boards. The driver module consists of the digital output drivers and a digitally controlled voltage regulator. This allows adjusting the output voltage to enable testing DUTs at their maximum rated voltage or interfacing low-voltage devices.

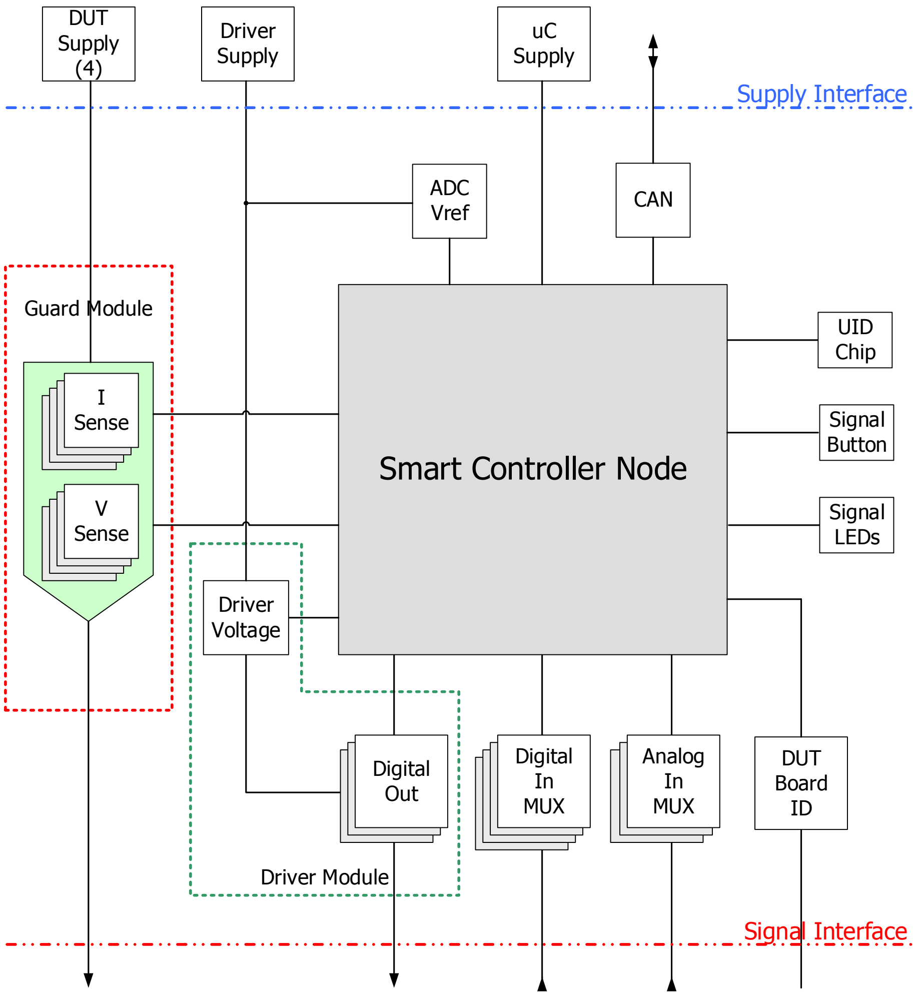


Figure 3.9: HTOL Controller Board

The analog and digital signal conditioning blocks contain multiplexers to reduce the vast amount of signals (7 channels × 8 DUTs) to a smaller number that can be handled by the microcontroller. In addition, signal LEDs and a signal button for simple location of the node upon request by the operator or through the host software are available. Further serial interfaces conclude the board design. They are used to identify the currently controlled DUT stress board and controller node.

The HTOL node board demonstrates that not only simple current patterns (see Section 3.3.1 – SmartMoPS) can be applied, but also feedback from the DUTs can be evaluated and processed at the location of the DUT. Consequently, only relevant data needs to be transferred to the host system. The configuration sent to the microcontroller defines the complete test setup for the tested device. After having received this data, the node board is able to autonomously perform the test procedure. It reacts to status queries from the SAM host software, but also sends out event triggered messages when it recognizes abnormal behavior of one of its tested devices.

3.3.3 DC-Converter Stress Board

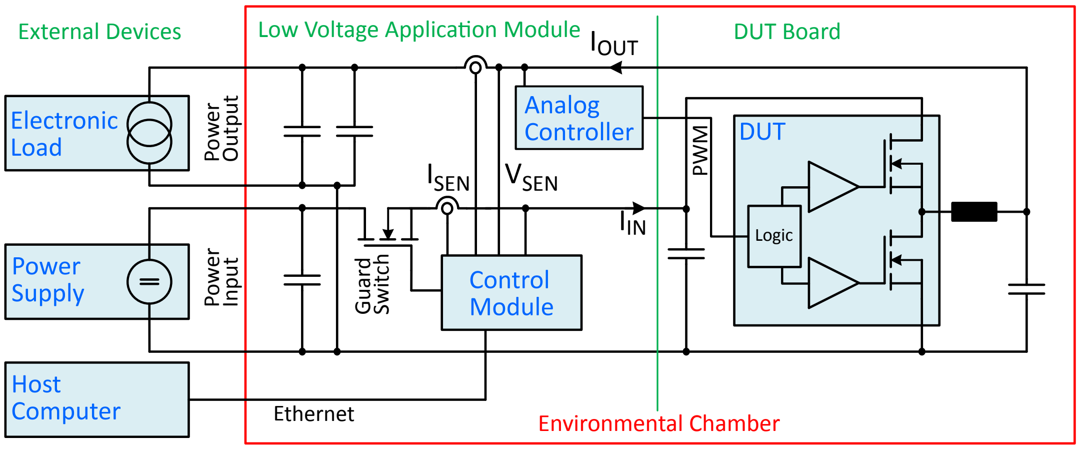


Figure 3.10: DC-Converter Board [82]

The DC-Converter stress board (Figure 3.10) has been designed to assess the performance and reliability of point-of-load DC-DC converters for CPUs. Two research investigations have been covered by implementing this test application:

The test application requires a controller to regulate the output voltage. Therefore, a microcontroller based on the XMC family, the XMC4500, has been placed on the application PCB along with a separate Dynamic RAM (DRAM) chip and Ethernet communication. The microcontroller is used to access digital control lines of the power conversion chip. Analog signals, such as current and voltage are read and provided to the digital control loop that defines the PWM for the half bridge. Further, the protection modules on the PCB are monitored in order to recognize the shutdown of the test due to over-temperature or over-current. The DRAM is used to increase the available memory of the microcontroller to store large amounts of analog measurement data. The Ethernet interface is used to transfer the measurement data to the host computer. This application does not yet have a configurable firmware, as it was not required for the simple test. The stress level is set by the input voltage (which is set by an external power supply) and the output current (which is consumed by an electric load). The first investigation dealt with storing and transmitting measurement data.

The second evaluation considered the use of a scripting language to control externally connected instruments. A special LuaVIEW module (see Section 2.3.2) has been created as a wrapper for instrument control. There, the API for the instrument is defined. When the module is loaded with the LuaVIEW framework, the module functions become available in the Lua environment and can be used to interact with the instrument using LabVIEW based VISA drivers. The benefits of using the Lua script language to define the test procedure in contrast to hard coded routines in the target host application are the following:

- The test engineer is capable of creating dedicated test routines, without the need to understand the complex software architecture.

- LuaVIEW based modules can be created, tested and debugged independently of the host application. Further, these modules can be reused in other applications if required.

The use of LuaVIEW quickly enhances the experience with LabVIEW based software. However, there are several drawbacks of the LuaVIEW library. Not all deficiencies are due to limitations of the library, but also because of how LabVIEW works.

- Lua was created with a powerful API in mind [83]. Thus, Lua is not a stand-alone library, but a set of functions that can be used within a C-based program or library. Within the C programming language, it is possible to register callback functions to the Lua state. These functions are executed, when the appropriate Lua functions is called. Lua within C allows arbitrary nesting of C→Lua→C function calls.

  In the stand-alone *LuaVIEW* library, only LabVIEW→Lua **or** Lua→LabVIEW function calls are possible. This restriction is mainly based on the missing functionality of LabVIEW to register callback functions within a C-based library. Since it is not possible to register LabVIEW Virtual Instruments (VIs) to the LuaVIEW state, Lua function prototypes are registered with a function index. Whenever LuaVIEW wants to execute such a registered function, the index is returned to LabVIEW, where the function body can be implemented in a case structure. The drawback with this solution is that the LuaVIEW engine is in control of the execution. However, custom LabVIEW code can be called at any time.

- Many programming languages have an exception handling mechanism for throwing and catching errors. LabVIEW however is a data-flow based language. Thus, jumps are prohibited eliminating throwing and catching errors. Lua, which is based on the C programming language, on the other hand provides such a mechanism. When LuaVIEW experiences an error while executing the script, the corresponding Lua state will be closed and an error will be reported to LabVIEW. Unfortunately, this effectively terminates the script execution, without the possibility to recover from soft errors by wrapping the LuaVIEW execution into a proper LabVIEW based error handling.

The scripts are typically provided by users and thus errors are expected to be present. Therefore, the Test-Plan-Checker (see Section 4.3.2) was created to check the script code before it is loaded and executed.

The successful evaluation of the XMC microcontroller implemented on the DC converter stress board lead to the design of an independent modular test control node (see Section 3.3.4). To enable further test applications under the same framework, the DC converter test board has been redesigned (see Section 5.4) to use the MicroMoPS as plugable control node [82].

3.3.4 MicroMoPS

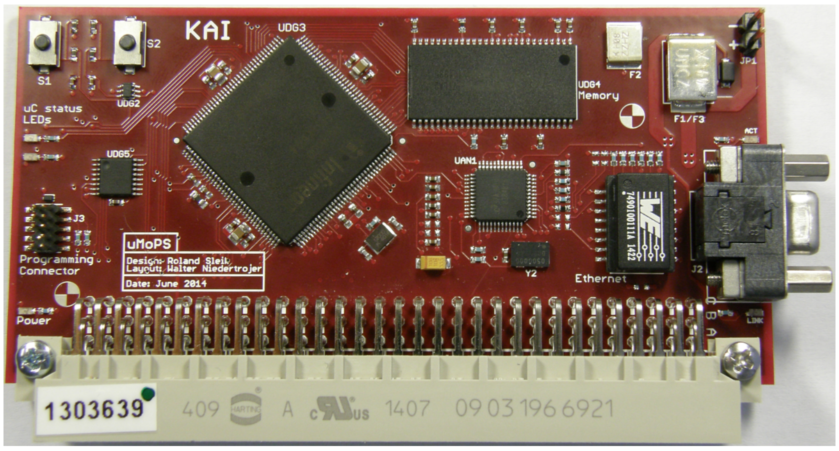


Figure 3.11: MicroMoPS Hardware Target

The findings of the three hardware related projects *SmartMoPS* – microcontroller within the climate chamber – (see Section 3.3.1), *HTOL Node* – communication and configuration via CAN bus – (see Section 3.3.2) and *DC-DC converter* – evaluation of XMC4500, Ethernet and DRAM – (see Section 3.3.3) lead to the development of a unified test execution hardware target, the *MicroMoPS* node. The requirements for this small (50 mm × 100 mm) module have been gathered from various reliability application engineers. The prototype test application is a 15 kW Power Factor Correction (PFC) converter stage for reliability assessment of the used semiconductor devices (see Chapter 5). This application requires several digital control and sense lines, PWM output, analog measurement and DSD. The currently available hardware (see Figure 3.11) features:

- The XMC4500 microcontroller based on the ARM Cortex-M4F processor
- Small form factor of 50 mm × 100 mm × 10 mm
- Automotive temperature range grade 1 (−40 °C to 125 °C [1]) and rugged connectors for use in the climate chamber
- Ethernet communication (100BASE-TX) with autonomous MAC address and IP address calculation derived from the internal unique serial number of the XMC microcontroller
- 8 MB acquisition memory allows storage of 4 MSamples of analog measurement data and digital responses (e.g. from SPI)
- 2 status LEDs to indicate run and error states
- 64 bit global time organized as 32 bit UNIX time and 32 bit fractional seconds
- 4 independent analog input modules capable of acquiring up to 24 input signals with a resolution of 12 bit at rates up to 1.8 MHz
- 4 independent analog output channels with 12 bit resolution
- 12 General Purpose Input Outputs (GPIOs)
- 4 digital event inputs that can trigger events for the FSM
- 4 DSD channels to acquire analog data from galvanically isolated sensors (e.g. in high voltage applications)
- 6 PWM outputs, divided into 4 units with single-phase output and 2 units with an inverted output for half-bridge control
- 1 SPI bus with 2 chip select lines to provide a digital interface to the test application
- Up to 32 software timers with 1 ms resolution for triggering custom events in the test FSM (see Section 4.2.1)

All IO modules can be directly interfaced using the Lua script engine running on the microcontroller firmware (see Section 4.1.3).

The use of the MicroMoPS microcontroller target has proved to be a powerful control module. Running the closed loop Proportional-Integral (PI) controller for this PFC power converter application could be reasonably handled while acquiring measurement data. The FSM based test plan design (see Section 4.2.1) simplified the start-up routines for this test application. The DC-DC converter stress test board (Section 3.3.3) is currently under redesign to enable plugging the MicroMoPS board.

3.4 The MoPS-CORE Microcontroller Firmware

The main reason to use microcontrollers is justified by the possibility to interact with the integrated on-chip hardware modules like analog converters, timers and digital interfaces. The MoPS-CORE microcontroller firmware is the common firmware for all XMC-based microcontroller hardware targets.

Listing 3.1: MoPS-CORE main loop

```c
void main_loop(void) {
    while (1) {
        measure_time();
        handle_comm_msg();
        guard_feed();
        led_active();
        handle_test_FSM();
    }
}
```

Due to CPU resource limitations and memory constraints, the firmware is not based on an actual operating system. The different tasks run cooperatively (Listing 3.1).

The time of the main loop is measured for statistical and investigation purposes. The time results are transferred to the host software upon request with the NODEINFO message. Next, messages that are received via the Ethernet communication channel are parsed. These messages either ask for the microcontroller status information, transfer the test procedure, or control the test by setting events in the FSM. Further, MoPS-CORE also contains a watch-dog guard, which is a hardware timer capable of resetting the microcontroller, unless it is periodically reset by the software. This guard can be enabled using Lua module functions and is reset once every iteration in the main loop. The firmware supports two status LEDs as visual indicators in case the controller software got stuck and the guard has not been enabled. Finally, the firmware runs the FSM handler (see Section 4.2.1) for custom test procedures. Within the states of the FSM, Lua code can be executed to interface the hardware modules of the microcontroller.

Several microcontroller hardware units may generate ISRs that need to be handled asynchronously in the background. Therefore, these routines either map those requests to events in the FSM diagram, or trigger DMA transfers as defined by the test procedure.

3.4.1 Lua Interpreter

It is not sufficient to only change the parameters of the test program (like timing values or output voltage levels), but it is usually required to also change the behavior and sequence of certain actions. The requesters of these application tests are typically company-internal hardware designers and product engineers. They prefer directly configuring a test sequence rather than low-level programming of microcontrollers. Thus, a dedicated firmware concept with a simple run-time configuration approach is required.

A set of commands is desired which can be executed directly on the microcontroller to interface the hardware. Of course, one could build a simple parser for reading commands and parameters from a textual input. However, since low-level programming of microcontrollers is difficult, this task is error-prone and a very large amount of commands must be implemented. Furthermore, such a design is inflexible and poses many risks to introduce bugs. The favored solution is to use an interpreted language, which also supports features such as loops, conditions and functions. Creating an interpreter that allows also features like loops and function calls is not trivial and can be a time-consuming task. Fortunately, several interpreters for use on embedded systems already exist (see Section 2.3).

Lua has been chosen for the implementation, because it is small, utilizes a minimum of RAM while still being powerful. Further, Lua does not only contain the VM on the microcontroller, but also the compiler is available on the target. However, the real benefit comes with the simple yet highly sophisticated C-API that allows an implementation of custom modules and hardware access routines. The interface between the Lua VM and C-functions is provided by a modifiable virtual stack. Thus, the C-code may receive parameters from Lua function calls and can provide results back to the Lua virtual machine. The configuration of the firmware and the available hardware modules is described in Section 4.1.3.

3.4.2 Hardware Interaction

A variety of different applications should be addressed without the need to change the basic firmware of a given microcontroller target. Especially, different applications require different functions on the external pins of the microcontroller.

Listing 3.2: Accessing a GPIO class module instance

```lua
pin = gpio("p5.10") -- obtain GPIO instance
pin:setOutput()     -- configure as output
pin:write(1)        -- set pin to high
```

In order to provide simple script commands as depicted in Listing 3.2, the Lua C-API is used. For each hardware unit, either modules (see Table 3.3) or classes (see Table 3.4) are provided. Modules are used when a global state is accessed or modified. Classes are used when access to an individual instance of a hardware periphery unit is desired in a direct way.

Table 3.3: MoPS-CORE Lua modules

Module	Description
comm	Query information about the communication module
dram	Query information about the external DRAM
guard	The system guard observes the code running and may reset the micro-controller
uc	Interact with the microcontroller test control (the FSM)
time	Query the microcontroller time

Within a test script, the user first needs to obtain a GPIO instance by calling the single public function `gpio` from the `gpio` class-module. The invoked C-function then takes care of setting up the periphery access and returns the Lua object `pin`. A Lua metatable is used to protect the Lua object instance from incorrect usage [83]. Consecutively, the user may configure the pin as output and write a value to it. According to this simple example, we have implemented classes for most of the periphery units available on-chip (Table 3.4).

Table 3.4: MoPS-CORE Lua classes

Class	Description
ai	Analog input – measure voltages
ao	Analog output – provide voltages
dsd	DSD – decodes the delta-sigma bit stream
eru	External event detection – can set events in the FSM
gpio	GPIO – control a single digital pin
gpioport	GPIO Port – multiple pins grouped together
pic	PI-controller – provides closed loop control between AI measured voltages and PWM output, the controller function is implemented in C and provides improved performance over Lua based functions
pwm	PWM – controls a hardware timer
spi	SPI – send & receive SPI messages
timer	System timer – sets periodic or one-shot events

3.4.3 Background Routines

In addition, various background functions need to be carried out by the controller such as closed loop PI control, PWM generation and analog waveform acquisition. Modern microcontrollers already provide powerful hardware modules that can do most of the required work in parallel to the software code. Therefore, the implemented Lua modules described above are used to configure these hardware units through the microcontroller registers.

Some hardware periphery modules will return data, like the ADC or serial interfaces, or trigger interrupts. In order to avoid race conditions and deadlocks when writing data to the memory, the following two options are provided:

Listing 3.3: Non-blocking example of AI module

```lua
scan0 = ai("scan0")  -- create analog module instance
scan0:setChan(1)     -- select channel of analog module
scan0:setFreq(10000) -- set 10 kS/s
scan0:setMem(100)    -- set storage for 100 samples
scan0:start(100)     -- acquire 100 samples in the background
```

Listing 3.4: Blocking example of SPI module

```
spi0 = spi("spi0")            -- open SPI instance
spi0:configure(1000, 16)   -- configure 1 MBaud, 16 bit
spi0:setCS(0x1)               -- select device at select line 1
recv = spi0:sendReceive(0xc0de)   -- receive data into variable recv
```

Non-blocking: Within the FSM test procedure, the user code has to allocate a memory region for the specific module by stating the required number of samples to be stored (see Listing 3.3). The result data is then written by the DMA controllers. The data can be accessed later, upon completion of the DMA transfer. Additionally, the host can also request data from an allocated memory region via the communication interface.

Blocking: The user code may read the requested hardware resource, but the command will not return a result until the value is available (see Listing 3.4). Due to this blocking nature, further script commands as well as the FSM handler are delayed. Therefore, only commands with deterministic execution time may use this option.

For the PI-controller, input and output signals as well as the control loop parameters are configured via Lua script commands. Now the invoked ISR function can be written in a purely functional style. This gives a large performance benefit compared to functions executed in Lua. However, the controller algorithm is fixed and may only be altered by adding a new routine in the C source code.

3.4.4 Electronic Data Sheet

The MoPS test system needs to verify, if the available distributed test nodes are capable of running the requested test. Therefore, the configuration of the specific hardware targets has to be known. The Test-Plan-Builder (Section 4.3.2) requires this information so that the test design engineers know about the specific modules while creating the test plan. The SAM host software (Section 3.1) checks if the test requirements are satisfied with the currently connected hardware targets. For that reason, every hardware target generates and provides an Electronic Data Sheet (EDS). The idea of EDS is borrowed from smart transducers [84]. The EDS includes configuration information that is otherwise only present in the microcontroller development environment DAVE.

Upon booting, the microcontroller firmware uses the pre-defined compiled hardware configuration to generate a JSON string to be transferred to the host application. The JSON library C source code has been taken from the Comprehensive C Archive Network[3]. When the EDS is read from the controller, it is uploaded to the MoPS project web server. The Test-Plan-Builder downloads new and updated EDS files from the web server and uses them for displaying visual hints (like code auto-completion) to the user.

The EDS is organized into three parts:

hwInfo *Hardware information* – The name, IP and MAC addresses as well as unique ID can be found here. In addition, the Git version hashes for the software (the MoPS-CORE firmware) and hardware (the configuration mapping) project can be found here to help investigate errors. The Git version information are also important for the TP-Builder to provide the correct set of available functions.

events *FSM events* – A list of available hardware target events for the FSM is given in this list. These events can be triggered by creating an instance of the Lua object referring to the appropriate modules.

modules *Lua accessible hardware modules* – A list contains the names of the of the available modules on the specific hardware target. Thus, the test engineer knows how to access the hardware periphery modules on the microcontroller. Additional information, such as DRAM size, the number of chip select lines for the SPI module instances or analog scaling information, is given. The use of the analog scaling information is described in Section 5.2.4.

3.5 Peripheral Modules

The modular stress test system described in this thesis deals with subjecting DUTs to electrical power pulses. Therefore, a control and sense module can in fact provide the pulse patterns, but cannot apply high current or high voltage pulses on its own. Several additional hardware modules are required according to Figure 3.1. These modules are connected via parallel, serial or mixed bus interfaces and operated by the control module, while the test application software is running.

The design of these additional hardware modules is not focus of this thesis. They are created by the hardware developers at KAI [82]. For the first implementation,

[3] http://ccodearchive.net/

mainly parallel control of the modules is implemented. In the test application, the control module only needs to provide digital pins for switching the module on or off and reading back the status. Special features of the modules can be dealt in the test procedure.

The following modules have already been designed and used depending on the requirements. Each module can be individually identified using a unique serial number chip to track the usage and to provide analog scaling information (see Section 5.2.4).

Carrier Board A carrier board is provided to hold the following modules. The carrier board routes the connection signals to the modules with or without further attenuation. The modules are typically connected with rugged connectors. However, module features may also be placed directly on the carrier PCB to save the number of separate modules required.

DUT Module The DUT is usually placed on a separate PCB to be able to exchange worn out or failed DUTs faster and to test multiple DUTs with the same hardware setup. In some cases, device sockets for the DUTs are also used.

Bias Module A bias module is required to apply defined voltage levels or currents while measuring device characteristics.

Guard Module The guard module is responsible for protecting the device subsystem from catastrophic failure of the DUT. In case the DUT draws too much power from the supply, the guard prevents the system from damages by switching off the power from the individual DUT. The limits for the guard are set by the control module and may be changed dynamically to follow the power consumption of the DUT during its test.

Line Module Active modules or passive impedances are required for specific applications like short circuit tests [85].

Load Module To load the DUT module with nominal operating current within the test application, active or passive load modules are required. Using the proposed setup of Figure 3.1, active power recirculation may be possible to save energy and help cooling the test setup.

3.6 Chapter Summary

This chapter introduced the MoPS distributed test system. By dividing the required tasks into smaller modules, they become manageable more easily. Further, the re-use of existing hardware and software modules is greatly encouraged.

The host architecture SAM is described. Based on the actor model, it defines loosely coupled software modules. Different actors perform different tasks. By introducing a hierarchy, the main software loads a *Test Actor*, which in turn distributes the test configuration among several *Node Actors*. The test description is executed in a separate *Lua Actor*. The *Node Actors* act as virtual representatives of the physical embedded test and control modules. Further support actors exist to control the external instruments and the communication channels to the hardware targets.

A brief overview of the event based communication mechanisms CAN and Ethernet is given. The first prototype test application used the CAN communication channel. Due to limitations of the effective CAN data rate, the succeeding projects implemented an Ethernet based communication channel. The benefit of using the actor-based software architecture made this transition very simple. All that had to be done was to add another communication actor. The use of Ethernet increased the usable payload, thus reducing the number of messages required to transfer the acquired measurement data.

In order to define and design a capable embedded hardware target, three studies were carried out. By developing a small microcontroller test application (the *Smart-MoPS*), the use of microcontrollers in the automotive ambient temperature range was successfully evaluated. The *HTOL Node Board* demonstrated the versatility of combining a fixed firmware and configuration via the communication channel. However, since the *HTOL Node Board* suffers the limitations caused by the CAN bus and the microcontroller processing power, the ARM-based XMC microcontroller was evaluated. Based on the results, a microcontroller hardware target for productive use with the MoPS test system is presented.

The firmware for the microcontroller is described. The Lua scripting language allows custom test programs to be executed in a fixed firmware. To access a microcontroller hardware periphery unit, Lua class modules (through the use of Lua metatables) are provided to the user. Two possible function call paradigms are available to read data in the background or to directly provide data in the Lua VM. The hardware configuration of such a microcontroller target is reported with an Electronic Data Sheet.

4 System Configuration & Programming

Special cases aren't special
enough to break the rules.

(Tim Peters)

Contents

The MoPS test system is a complex distributed yet flexible system. Consequently, it requires simple features and interfaces for the users to configure and control the system. To setup the system, the externally attached instruments need to be specified, recognized and validated upon system start-up. The modules required for the test procedure provide an EDS which is used to download the test code properly and to read back the analog and digital measurement values.

The configuration parameters of the different modules and systems need to be remotely accessible in order to create a test procedure that can be correctly executed.

The Test-Plan Builder (Section 4.3.2) is capable of reading various configurations and providing the proper features to the test code.

This chapter describes the configuration possibilities of the host software and the hardware targets in Section 4.1. The organisation and creation of the test plan are explained in Section 4.2. Section 4.3 describes the integration of the microcontroller test nodes into the SAM host application. Last, the distribution of the software, the API documentation and the hardware configuration files are explained in Section 4.4. Parts of this chapter have been published at the 13th IEEE International Conference on Industrial Informatics in 2015 [72].

4.1 Configuration Options

MoPS is not only a single test system, but a general architecture to be used for different test applications. Similar to the already existing ACUTE and ARCTIS systems, multiple test system instances with varying configuration settings are required. As mentioned in Chapter 3, multiple applications using the same host software have been covered: The HTOL test system (see Section 3.3.2), the DC converter test system (see Section 3.3.3) and the prototype implementation (see Chapter 5) are controlled by the SAM host program (Section 3.1).

In order to support such flexibility, configuration files are provided. Depending on the application usage, JSON-files and C-files are available. JSON based configuration is used for all applications that run on a standard host computer. C-file based configuration is used for the C-source code for the hardware targets. Using pre-processor directives, certain code sections are enabled or disabled.

4.1.1 JSON Format Enhancements

The configuration parameters for the SAM host software (Section 3.1), the test plan created and exported by the TestPlan Builder (Section 4.3.2) and the EDS (Section 3.4.4) are based on the JSON text format. Using the evaluation results from Section 2.4.3, the JSON format provides a human readable text that can be read and written by the TP-Builder application as well as the LabVIEW host software.

For configuration files, comments within the file next to the configuration variable are vital to explain certain parameters to the user or to provide an example value. Further, a quick and simple way to disable parts of the configuration is very important during testing and for the debugging of the software code. However, the original JSON specification does not include comments. In order to make comments similar to JavaScript comments available, the code of the Github-hosted, free project `strip-json-comments`[1] was ported to enable that functionality for LabVIEW based software. Therefore, C++-style comments `//` and `/* */` are now possible for JSON files readable by the SAM LabVIEW application.

4.1.2 SAM Configuration

The host software SAM needs to load its test system deployment configuration from the local configuration file. The JSON file format has been chosen because of the benefits listed in Section 2.4.3. The `SAM.json` configuration file is organized into the sections `tools` (Listing 4.1) and `instruments` (Listing 4.2) as well as several numerical arguments to enable special features and configure the timing of broadcast messages. Various external applications, tools and libraries are used together with the MoPS test system. However, not every MoPS test system instance requires all of these dependencies.

Listing 4.1: SAM tools configuration

```json
{"tools":[{
    "name":"java",
    "install dir":"",
    "version info":"-version",
    "update URL":"",
    "version URL":""
},{
    "ClassName":"SAM Tool Jar",
    "name":"tpchecker",
    "install dir":"<data>/lib/",
    "version info":"-V",
    "update URL":"https://server/directory/tpchecker-latest.jar",
    "version URL":"https://server/directory/version.txt"
}]}
```

The `tool` section describes external programs that are additionally used with SAM. The main purpose of this interface is to make use of already available software without

[1] `https://github.com/sindresorhus/strip-json-comments` commit: 69a1a17

the need to re-implement existing functionality. As an example, Java is required to run the TP-Builder tool (see Section 4.3.2). The fields for a tool configuration are explained below:

ClassName The *Character Lineator* library can load child classes of the configuration. In the example above, the *tpchecker* tool is a Java application and therefore requires a slightly different calling syntax.

name The name is used within SAM to reference to the requested program.

install dir Installation directory – the path to the program. If this is not specified, it will be read from the operating system path environment variable.

version info The required command line parameter to get the program version of the used tool. This version information is used together with the next two Uniform Resource Locators (URLs) to update the program automatically through SAM (see Section 4.4).

update URL The URL to the most recent available binary release of the tool.

version URL The URL to a file describing the latest available version of the tool.

Listing 4.2: SAM instruments configuration

```
{"instruments":[{
    "ClassName":"SI PSU",
    "Name":"PSU1",
    "Address":"GPIB0::1::Instr",
    "Type":"Agilent_HP66"
},{
    "ClassName":"SI Load",
    "Name":"Load1",
    "Address":"GPIB0::2::Instr",
    "Type":"HHPMLI_0100"
}]}
```

The `instrument` section describes externally attached devices to be operated by SAM for running the test procedures. In order to know which instruments are available when loading a test plan, the connected instruments must be specified in the test system configuration. Upon starting the SAM host software, an *Instrument Actor* is created for each entry in the list. The Actor receives the configuration of the instrument and is further available for commands being sent to the device using the given name. The available instruments for a test system are also reported to a web server, so that the TP-Builder application (Section 4.3.2) can refer to them while creating the test procedure. The fields for the instrument configuration are defined below:

ClassName The class name specifies the LabVIEW class to be loaded for each type of instrument. The LabVIEW class specified the available API functions, which differ for example between power supplies and external loads.

Name The name is used for communicating with the instrument within the test procedure.

Address The address is a VISA resource to specify the communication port SAM can talk to the instrument. It is possible for SAM to automatically list connected devices. However, if two instruments of the same type are available, this may lead to conflicting configurations. Further, communication interfaces like Ethernet have a too large address space for SAM to search for all connected devices.

Type The type specifies which specific instrument driver needs to be used. Various vendors implement only a subset of the standardized commands [63] for their instruments, therefore requiring custom codes to be sent to the devices.

4.1.3 Configuring the MoPS-CORE Firmware

The general description of the MoPS-CORE microcontroller firmware can be found in Section 3.4. The MoPS-CORE firmware has been designed in a way that easily allows creating different configurations, depending on the requirements. The configuration of the microcontroller hardware target is performed within the microcontroller C source code.

The required tool chain for compiling the MoPS-CORE firmware for the XMC microcontroller is the Infineon DAVE development platform. DAVE is used to configure the pin assignment of the microcontroller, because it contains the hardware models that are required to perform validation of the selected pin combinations for the selected microcontroller. Pre-processor switches are used to compile only the code sections that are required for a given target. A simple C-header file `MoPS_config.h` within the root directory of the DAVE project enables support for the different peripheral on-chip hardware modules. The purpose is to reduce the code size, when support for certain periphery modules is not needed.

As indicated by Listing 3.2, the test system user needs to write simple Lua scripts to access the microcontroller periphery units. However, there are several steps required in order to provide the firmware image to access the physical pin in such a way (see Figure 4.1).

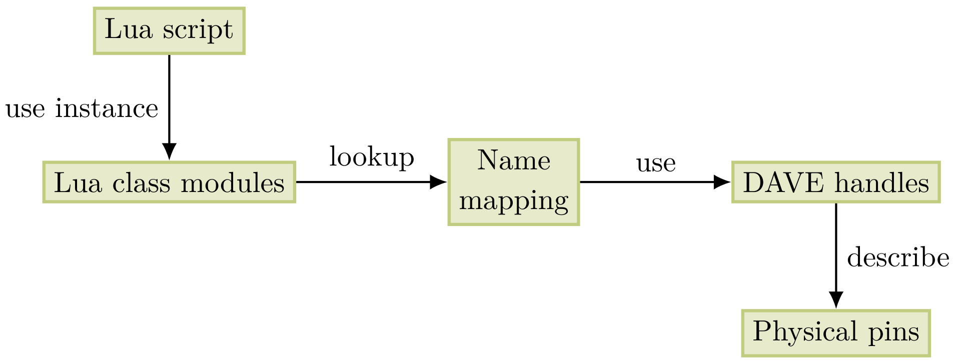


Figure 4.1: Accessing a physical pin through Lua

Physical pins

Using DAVE, a specific pin location can be selected by the user, or automatically assigned by the solver algorithm within DAVE. To create a configuration, the user typically selects only the number of hardware resources he or she wants to use with a new hardware target (e.g. 8 GPIOs, 4 PWM units, ...). The solver tries to find a possible combination and reports the automatically chosen pins. The user, nonetheless, has the ability to manually assign the desired pins. When the configuration is not possible, because the remaining available pins do not provide the selected function, an error is reported. The resulting pin mapping is then used to design the target hardware circuit and PCB layout. In theory, this workflow can save several hours compared to finding possible valid pin combinations manually.

DAVE handles

When a DAVE configuration has been created, the source code can be generated from this model. Using this process, DAVE provides C-data structures (so-called "handles") and API functions to use the selected periphery modules in the C-based

Listing 4.3: The DAVE digital pin description

```c
const IO004_HandleType IO004_Handle0 = {
    .PortNr   = 5U , /* Mapped Port */
    .PortPin  = 10U, /* Mapped Pin */
    .PortRegs = (IO004_PORTS_TypeDef*)PORT5_BASE /* Base Address */
};
```

application. Additional source libraries (e.g. lwIP for Ethernet communication) are also copied to the source directory. Usually, the generated handles provide the microcontroller port and pin description as well as the address of the register that needs to be accessed to operate the module (see Listing 4.3).

Name assignment

When the physical pin is known and accessible with the previously explained C structure, a meaningful name is assigned. The C-source file `MoPS_config.c` specifies the link between the DAVE resource handles and the MoPS-CORE system code. An excerpt of this mapping can be seen in Listing 4.4. This code section specifies two GPIO resources ("p5.10" and "p5.11") to be usable within the Lua script. Using this loosely coupled name-based interface allows a very simple configuration for different hardware targets using the same base firmware.

Listing 4.4: The MoPS-CORE digital pin to name mapping

```c
#if EN_MoPS_GPIO
#include "mops_core/MoPS_gpio.h"
const MoPS_gpio_reg MoPS_gpio_reg_handles[] = {
    {    .name = "p5.10",    .handle = &IO004_Handle0,     },
    {    .name = "p5.11",    .handle = &IO004_Handle1,     },
    END_OF_LIST,
};
#endif
```

More complicated modules, like the analog input, also contain information about the measurement ranges or scaling information. By providing this information to the firmware, the scaled values can be used directly in the Lua script. Further, the MoPS-CORE firmware can also generate and provide the EDS (see Section 3.4.4).

Lua class module

We may recall from Listing 3.2 that within the Lua script the constructor function `gpio("p5.10")` was called. When Lua encounters the function call to `gpio`, the registered constructor C-function will be called by the Lua VM (Listing 4.5).

Listing 4.5: The MoPS-CORE digital pin Lua constructor

```c
static int gpio_construct(lua_State *L) {
    const char *str = luaL_optstring(L, 1, "");
    const MoPS_gpio_reg *entry = NULL;
    entry = get_handle_by_name(MoPS_gpio_reg_handles, str);
    return gpio_init_module_instance(L, entry);
}
```

First, the string argument – the name of the desired hardware module (i.e. the pin) – is taken from the Lua stack interface. Next, the DAVE handle has to be found by searching through the mapping of Listing 4.4. Finally, the module instance can be initialized using the named entry from the configuration list. The return statement specifies the number of elements placed on the Lua stack interface and is `1` on success (a Lua object has been created) and `0` on failure. Error handling is not included in this simple example, but required when the user provides an unknown instance name. In this case, the Lua variable is `nil` and a debug message may be sent to the host system.

The created module instance is a Lua user data object protected by a metatable. The metatable serves two purposes: first, object oriented access using the colon-operator can be provided. Second, incorrect usage – like calling PWM methods on a GPIO instance is prohibited because the metatable is checked first.

Lua script

Finally, the Lua script code `pin = gpio("p5.10")` can be used (see Listing 3.2) to control the corresponding microcontroller pin 5.10 (see Listing 4.3) defined with "`IO004_Handle0`".

4.1.4 DAVE MoPS-CORE App

Since writing C code is difficult and error-prone, a GUI for DAVE has been developed [24] using the Software Development Kit included in DAVE. The idea is to provide a very minimalistic interface to the test engineer where he or she selects the number of desired hardware units. The assignment of the microcontroller pins is either performed by the DAVE solver, or can be done manually if required. The DAVE tool also provides the pin list via the "Resource Mapping Information", so that the engineer can create the layout of the hardware target. The engineer can then assign user specific names to the hardware instances or use the names provided by default. The name assignment in the configuration C-file (see Listing 4.4) is done by the DAVE App, thus no manual C programming is required anymore. Hence, test engineers can easily create a firmware configuration specifically tailored for their test application.

4.2 Test Plan Definition

In order to develop a test plan suitable for a modular test system, the test requirements need to be considered. By investigating typical power electronics test scenarios, it became evident that the test program on the controller has to take care of:

- Applying configuration, digital and analog stress test patterns.
- Reading system responses and measurement values.
- Receiving and executing messages from the governing test handler on the host (i.e. "host messages", especially *start* and *stop* software events) in order to proceed to the test sequence where dependencies on external hardware are defined.
- Handling trigger signals generated by the application or microcontroller periphery (i.e. interrupt events)
- Evaluating and reacting on internal states (i.e. measured analog signals, return values from digital interfaces)
- Running background services for sending and receiving messages and transferring measured data to the host.

4.2.1 Test Plan Model

The bullet items from the list above can be identified as **actions** and *events*. An **action** is the sequence of instructions which is to be carried out by the controller in order to perform a specific task and evaluate the result. An *event* is either triggered by internal modules in the controller or externally through the test application and requires the controller to perform an **action**. Given these two basic terms, the non real-time behavior of a test procedure resembles the FSM model [86, 87].

The reaction to an *event* can be described by a transition in the state diagram. While the controller is in a certain FSM state, it continuously evaluates all possible events and switches state accordingly. Most of the time, a state switch should not occur, e.g. when the controller is waiting for a special trigger to arrive. This implicit behavior can be modeled by the unconditional *@else* event, forming a transition from each state to itself in a loop. In order to execute a state only once and immediately move to the next state, this loop needs to be cut. This can be achieved by manually pointing the *@else* event to the next state.

The information whether an event has occurred is stored in a fixed-size, pre-allocated table indexed by the event number. Thus, the microcontroller ISRs and the main loop may modify the table, since no dynamic memory allocation – as found in a queue-based system – is required. Furthermore, locking of this global table is not required, as event occurrences are set (by hardware ISR or software) and cleared (by the FSM handler described below) by independent instances.

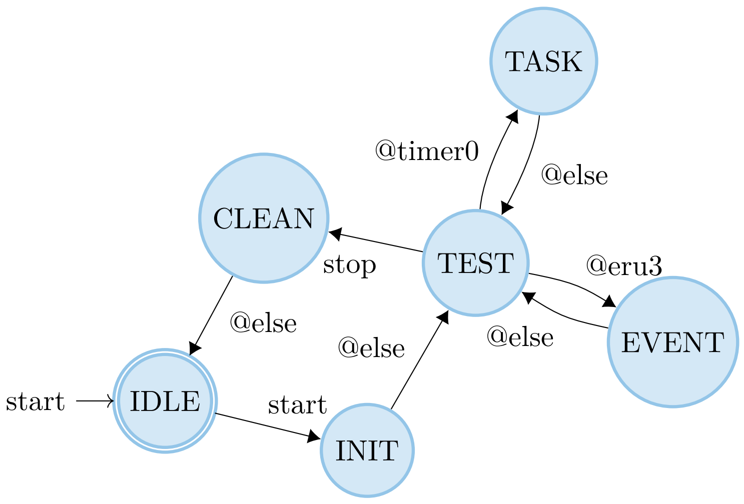


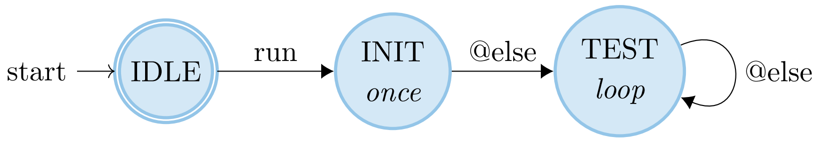
Figure 4.2: Simple controller state machine

A simple example of a state machine structure for such a test sequence is given in Figure 4.2. Once the microcontroller has finished booting, it waits for the reception of a new state machine description via the serial communication interface. It parses the contents and builds up the directed graph. Finally, the controller starts executing the state machine with the **IDLE** state. While being in the **IDLE** state, no user definable actions can be carried out and only background services are being processed. These background services are required to transfer the configuration to the controller and convert it into the state machine. The **IDLE** state is also the final state and the controller can be stopped or the FSM structure may be changed.

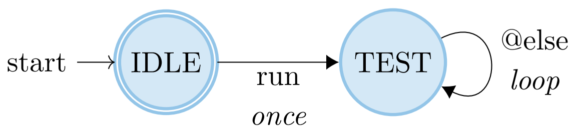
Upon reception of the *start* event from the host, the FSM will propagate into the **INIT** state, perform the one-time setup procedures given by the test description and automatically change into the **TEST** state indicated via the *@else* event. While in the **TEST** state, the automaton in this example listens for the hardware-events *@timer0* (periodic time-triggered work load) and *@eru3* (external events – when e.g. the external analog comparator exceeds a given threshold) and the software-event *stop* (for stopping the test).

While running the FSM, the **action** (i.e. custom test code) needs to be executed. There are two possibilities for attaching the action to the FSM: either to the state or to the transition. This results in a list of transitions with 3-tuples or 4-tuples respectively: If the code is attached to the state, it will be executed continuously while the FSM is in the state. The 3-tuple transition consists of the **current state**, the triggering *event* and the **new state** `(currentState, event, newState)`. In this case, the name of the state is synonymous for the name of the function as state names must be unique in this representation. If the code is attached to the transition, the resulting 4-tuple has the additional field **function**, because the event names are not unique in the FSM (i.e. the same event can cause a transition from multiple states) `(currentState, event, newState, function)`.

The representation with the larger 4-tuple requires more elements per entry, but usually fewer states to model a test procedure compared to the version with the 3-tuple (see Figure 4.3). Figure 4.3a requires a separate state to run the initialization code once, whereas in Figure 4.3b, the initialization code is executed at the transition from the **IDLE** state to the **TEST** state. In both examples, the looped code is related to the *@else* event of the **TEST** state. In Figure 4.3a, the event is required to continuously enter the **TEST** state and execute the attached code. In Figure 4.3b, the *@else* event is further used to stay in the **TEST** state, but the code is executed in the loop transition from **TEST** to **TEST**. After evaluating both possibilities, it was decided to use the approach of Figure 4.3a assigning the code to the state. The reason



(a) FSM model using 3-tuples



(b) FSM model using 4-tuples

Figure 4.3: FSM model comparison

Table 4.1: State transition table describing Figure 4.2 using 3-tuples

#	currentState	event	nextState
1	IDLE	*start*	INIT
2	INIT	*@else*	TEST
3	TEST	*stop*	CLEANUP
4	CLEANUP	*@else*	IDLE
5	TEST	*@timer0*	TASK
6	TASK	*@else*	TEST
7	TEST	*@eru3*	EVENT
8	EVENT	*@else*	TEST

being that placement of the script code inside the state is easier to understand and can be implemented straight forward in the TP-Builder application (see Section 4.3.2).

FSM diagrams can easily be converted into C-code, compiled and transferred to the controller for execution. However, alteration of the procedure (e.g. by adding another state and its corresponding events or synchronization with the host or other controllers) requires a skilled programmer who is capable of implementing these changes as well as performing recompilation and finally flashing the firmware. This solution is not favorable for practical test plan development and debugging in the test laboratory.

Alternatively, allowing the states and transitions to be reconfigured while the FSM is in a specific state (the **IDLE** state) in order to execute arbitrary FSM diagrams improves the process considerably. Therefore, the controller may receive a state transition table as given in Table 4.1. The table is constructed by enumerating all transitions between states. This allows for a full reconstruction of the state machine diagram. To simplify the diagram and to reduce the size of the table, the implicit loop transitions via the *@else* event are neither stored nor transferred.

The transition table is composed on the host computer where the test plan is created. It is then converted into a serial form in order to be transferred via the communication interface. A pair of states and their linking event are converted into numerical IDs each and then collected as a 3-tuple, which represents the transition. The controller receives the list of 3-tuples and reconstructs the table in its internal RAM.

```
   Data: transitionTable, currentState
   Result: newState
 1 newState ← currentState;
 2 foreach transition in transitionTable do
 3     if transition.origin == currentState then
 4         if transition.event == ELSE then
               /* save the state for the @else event, it will be used if
                  no other event occurred                              */
 5             newState ← transition.newState;
 6             continue;
 7         end
 8         if eventOccurred(transition.event) then
               /* the event has occurred; update newState and exit     */
 9             newState ← transition.newState;
10             clearEvent(transition.event);
11             break;
12         end
13     end
14 end
```

Algorithm 1: State machine handler

The FSM handler, as presented in Algorithm 1, then traverses the states in a simple way. If none of the events match the events listed in the transition table, the current state is kept. Therefore, the newState variable is assigned beforehand (Line 1). While iterating through the table, only transitions that originate from the current state are considered. If the current state contains an outgoing transition using the *@else* event, it is saved and the search algorithm continues querying the table in order to match a possible triggered event (Line 4). If the current state contains a transition event that has actually been triggered, the new state is immediately assigned and the search is complete (Line 8).

```
Data: transitionTable, currentState
Result: newState
newState ← currentState;
/* only inspect outgoing transitions from the current state      */
foreach transition in transitionTable[currentState] do
    if transition.event == ELSE then
        newState ← transition.newState;
        continue;
    end
    if eventOccurred(transition.event) then
        newState ← transition.newState;
        clearEvent(transition.event);
        break;
    end
end
```

Algorithm 2: Improved state machine handler

The search behavior is improved by storing the outgoing transitions from each state in a tree-like data structure indexed by the current state (Algorithm 2). The updated algorithm performs a lot better than Algorithm 1 by only searching through a limited number of transitions.

4.2.2 Test Plan File Structure

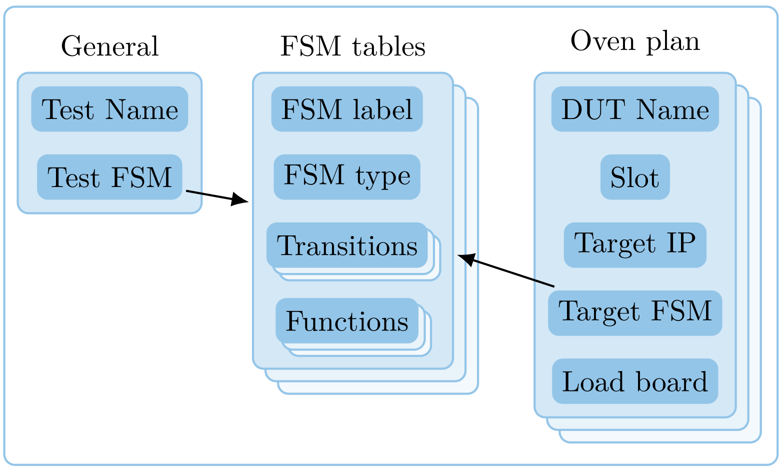


Figure 4.4: Test plan structure

The transition table from Table 4.1 can easily be converted into the JSON serialized form for storing on file or database servers and for loading the test procedure into the test system. The test plan contains general information about the test, the oven plan and multiple FSM tables. There is one main FSM – the so-called *Test FSM* – in the test plan, which is run on the host system. This master FSM can synchronize multiple hardware targets and may also control externally connected instruments such as power supplies through additional API functions available in the SAM application.

The oven plan contains information about the tested devices and applications. Especially the position within the test system and the used target hardware identified by the IP address are included. Further required components are summarized with the load board entry which is also identified and can be uniquely traced.

The FSM tables are marked by a string label. The Test FSM and the hardware target FSM entries link to these tables. Thus, multiple hardware targets may run the same test procedure while the test plan only contains this description once. The FSM tables contain the above mentioned transition table to reconstruct the FSM graph. The *Functions* entry contains the user code for each state in the transitions table.

4.2.3 Test Plan Transformation & Transfer

Embedded controllers, like the XMC microcontroller used in the MicroMoPS hardware target (Section 3.3.4), possess only a limited amount of memory to receive and parse the test procedure. The test plan JSON file however can easily grow to several kilobytes. In order to provide the test code to the hardware target, some transformations are required.

The transition table is converted into a list of triples as described in Section 4.2.1. String names of the states and transitions are converted into integers to reduce memory consumption and speed up the parsing. In addition to the FSM table, the microcontroller also receives the Lua script code. For each FSM state, a Lua function is created with the contents of the user defined Lua script code. The script is compiled into the Lua VM when the test plan is loaded onto the microcontroller. Then, the functions defining the code in the FSM state are known to Lua and can be executed during each iteration while the FSM runs.

4.3 System Integration

The microcontrollers are connected directly to the units under test. Therefore, they can easily apply patterns and monitor the application. However, a typical power stress test also involves external instruments, such as power supplies and active loads as displayed in Figure 3.10. These devices are usually shared between several test applications running on the same host system. The microcontrollers may not be powerful enough to perform advanced analysis procedures or store results to a centralized data location. Furthermore, the FSM on the microcontrollers runs independently of the host application. Thus, a hierarchical approach is required to control the overall test routines.

4.3.1 Communicating State Machines

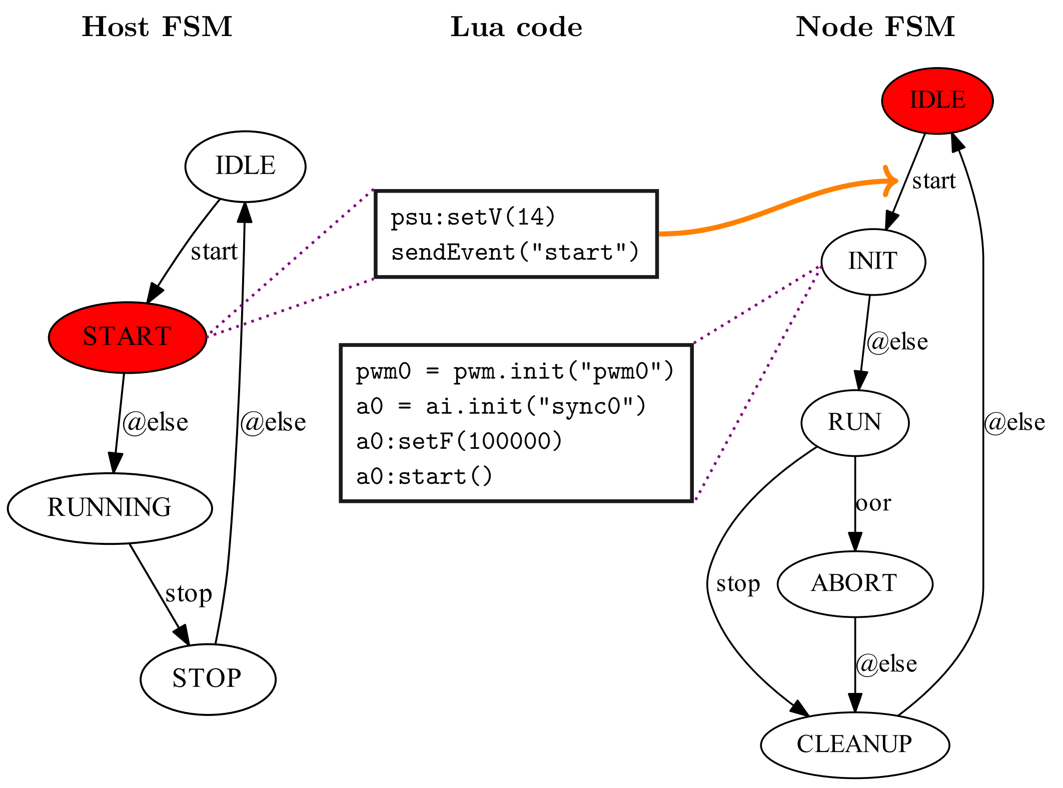
As can be seen in Figure 3.2, the FSM concept is present on all three hierarchy levels: The hardware targets use the state machine handler and the Lua interpreter to access the on-chip hardware periphery. Each Test and Node Actor can create a dedicated FSM Actor for custom data analysis and storage, for communication with the next lower or higher hierarchy level (sending events and transition notifications) or for the purpose of instrument control (Test Actor only).

By sending events via the communication interface, state machines from different hierarchy levels can be synchronized. Once the test definition is loaded, the host program can send events to the hardware targets in order to start the test execution. It can also verify that an external instrument has been configured properly. Therefore, two independent state machines are created; one for the host and one for the microcontroller. Figure 4.5 demonstrates a minimum example for such a test definition:

1. In the **Host FSM**, the *start* event is triggered (e.g. by pressing the “Start test” button on the GUI).

2. The FSM proceeds to the **START** state and executes the associated Lua script code: First, the voltage of the power supply is set. Afterwards, a separate *start* event is sent to the microcontroller target via the communication interface.

3. The **Host FSM** transits to the **RUNNING** state and waits for test completion (indicated by the *stop* event).



```
psu:setV(14)
sendEvent("start")
```

```
pwm0 = pwm.init("pwm0")
a0 = ai.init("sync0")
a0:setF(100000)
a0:start()
```

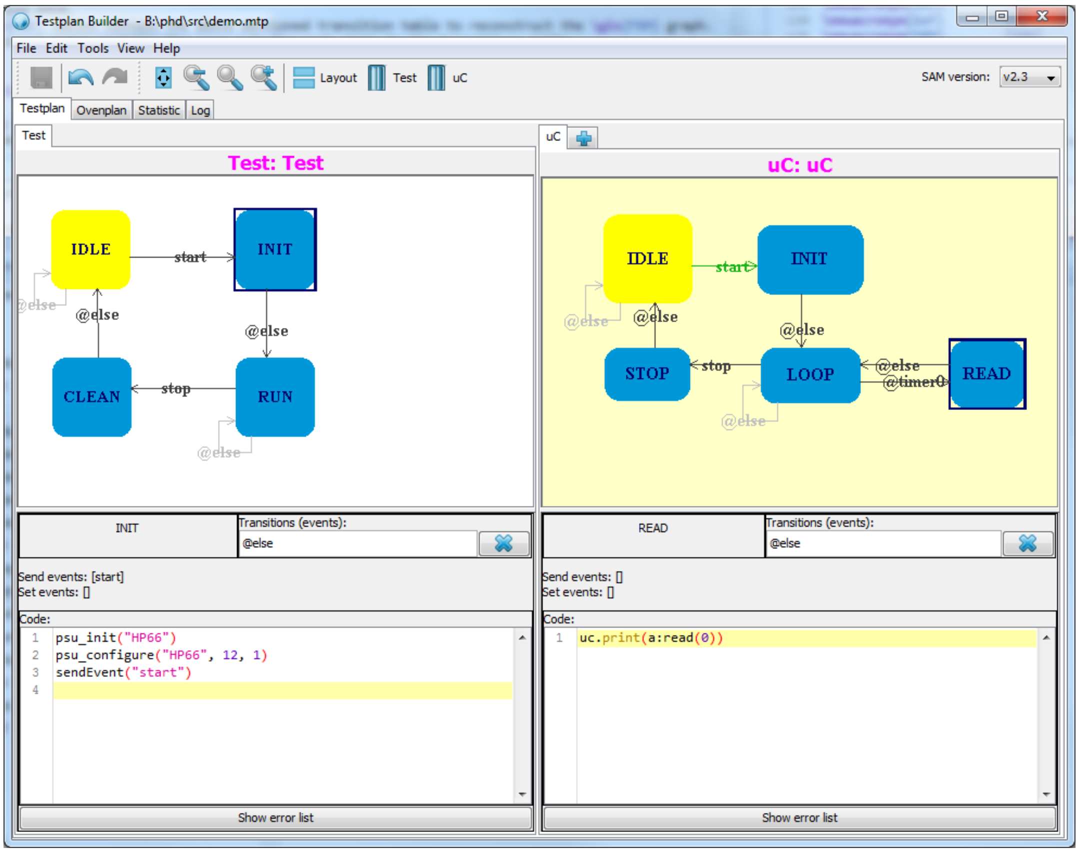
Figure 4.5: Host to Node FSM synchronization

4. The microcontroller receives the *start* event and can now move to the **INIT** state where it executes the assigned Lua function.

All function calls visible in this picture are custom module implementations on top of the Lua standard libraries. In addition, the Host-FSM supports reading measurement values from the microcontroller. According to the evaluation of the response, the Lua-based user script may invoke a transition in the FSM.

4.3.2 Test Plan Builder

When drawn graphically, the FSM model is easy to understand. However, creating this test plan structure in text format is tedious and may lead to errors. Therefore, the Test Plan Builder (TP-Builder) master thesis project was carried out to provide a GUI tool [23]. The TP-Builder aims to ease drawing of interactive FSM diagrams and entering the code into specific states (Figure 4.6).



INIT
Transitions (events): @else
Send events: [start]
Set events: []
Code:

```
psu_init("HP66")
psu_configure("HP66", 12, 1)
sendEvent("start")
```

READ
Transitions (events): @else
Send events: []
Set events: []
Code:

```
uc.print(a:read(0))
```

Figure 4.6: Test Plan Builder

While performing the operations on the FSM diagrams, TP-Builder continuously verifies the FSM structure and the entered Lua script code in order to provide a functional test procedure. According to the test plan model, at least the following checks are performed:

FSM structure A route must exist from the **IDLE** to any FSM state and back. This condition is required to be able to stop a test procedure (by returning to the **IDLE** state) and transfer a new test description.

API functions The Lua script code may only call API functions that are available for the selected target hardware and firmware version. When an unknown function is called during execution of the test, the currently executed Lua script of the corresponding state is interrupted. Further, a soft error is triggered by setting the *@error* event. Therefore, the test software can react to specific errors allowing the test system to return to a safe state. This mechanism is similar to exceptions known in other programming languages.

Events The two API functions `setEvent("event")` and `sendEvent("event")` are provided by MoPS (see Section 4.3.1) to trigger events in the current and deeper FSM levels respectively. TP-Builder checks that the events exist on the corresponding FSM diagram. Further, it is verified that events present in the FSM can be triggered during the execution.

Oven plan Within the TP-Builder, DUT names and positions, their assigned test microcontroller and the used FSM are described. The availability of the microcontroller target is also checked by looking up the selected target in the online published list.

Thus, the tool guarantees that every FSM graph is able to return into the **IDLE** state as required by the model (Section 4.2.1). Further, the TP-Builder fetches the API documentation (Section 3.4) and the hardware description (Section 3.4.4) which allows checking of the user entered Lua script code against the used hardware target configuration and target firmware version.

Test Plan Checker

The TP-Builder can also be run in a command line interface mode, where it will perform only the validity checks of the provided test plan file. The host software SAM uses this interface to check the loaded test plans before they are distributed among the specified hardware targets.

4.4 Software & Documentation Deployment

One important aspect of software development in general is the process of acquiring and updating the required software tools. The main software projects and their libraries are developed and distributed using Git repositories. The location of the binary distributed tools is described in the SAM configuration (Section 4.1.2). Upon start-up, SAM reads the version information of the currently installed tools and compares the local version with the remote version. If the remote version differs, SAM downloads and installs a fresh copy.

In order to distribute the tools like the TP-Builder, a web server has been set up (Figure 4.7). The build tools in the separate projects are able to generate and upload the documentation as well as the compiled binary. Currently, there is no database

```
https://server/MoPS/
 ├─ CORE
 │   ├─ eds
 │   ├─ v1.9
 │   │   ├─ classes/
 │   │   ├─ modules/
 │   │   └─ index.html
 │   └─ v1.10
 │       └─ ...
 ├─ SAM
 │   ├─ config/
 │   ├─ v2.2
 │   │   └─ ...
 │   └─ v2.3
 │       └─ ...
 └─ TP-Builder
     └─ bin
         ├─ version.txt
         └─ tpbuilder-latest.jar
```

Figure 4.7: Distribution server structure

application on the server. Therefore, only privileged developers are able to update files on the server.

The first directory `CORE` contains configurations and documentation of the MoPS-CORE microcontroller firmware. The EDS from the various hardware targets are stored in the directory `eds`. Within the versioned directories (`v1.9`, `v1.10`, ...) the API documentation for the use of the microcontroller firmware is located. The HTML documentation is extracted from comments added to the C source code using the LDoc tool[2]. LDoc is a Lua documentation tool similar to the well known JavaDoc[3] system. It is capable of reading Lua source files, comments within Lua source files and special comment annotations from C source files. Therefore, the literate programming style is supported [88].

In the second directory, `SAM`, the API documentation is provided in the same way through LDoc generated HTML files. SAM provides a LabVIEW script that extracts the available functions into dummy Lua source files that can be processed by LDoc. Furthermore, the `config` directory is located within `SAM`. There, the configuration settings for the different SAM test system instances are stored. These settings include

[2] https://github.com/stevedonovan/ldoc
[3] http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

connected external instruments like power supplies and electronic loads reported by SAM. Thus, the TP-Builder can obtain this configuration and provide the user with correct Lua function options.

Finally, the directory `TP-Builder` contains the compiled version of the TP-Builder application. The `version.txt` file is used by SAM to update the local file, since it is used to validate the loaded test plan through the command line interface (see Section 4.3.2).

4.5 Chapter Summary

This chapter describes the enhancements to the JSON format, which have been implemented in the libraries used for serializing and deserializing the test plan and configuration data. Through the configuration features, the SAM architecture can be deployed to multiple different test systems in a simple way without changing the general software. Further, the configuration options include settings for automatic updates of provided tools (e.g. the TP-Builder) and libraries. External instruments are abstracted and therefore can be changed on the fly – without the need to change the referenced name in the test plan. In addition, the specified instruments are checked during loading the device drivers.

The configuration concept of microcontroller periphery is presented. It has the aim to provide a generic microcontroller firmware for yet unknown target hardware designs. As a result, a named hardware module reference that is accessible by the Lua script is given as a Lua class module. When creating the class instance, the named reference is used to look up the physical pin in the name mapping of the DAVE handles. Access to the hardware pins is provided by Lua functions wrapping the DAVE API functions.

To be able to execute user provided script code and react to events from the hardware periphery units, a special test plan model has been developed. It makes use of the FSM model, where the user script is executed within the states and transitions can be caused by either software or hardware events. The test plan file contains several FSMs, as it specifies a test for multiple targets. Further, the host FSM is also described, which can synchronize and sequence hardware targets. In order to synchronize the FSMs on host level and microcontroller level, a communication mechanism has been introduced. The host provides an API function so that an *EVENT* message is sent to the hardware target. The TP-Builder tool helps the creation of such test plan FSMs and scripts.

The executable software files as well as the documentation are provided on a web-server accessible to all users within the local intranet. The TP-Builder retrieves the documentation to provide extensive help for creating the test plan files. The server is further used to update dynamically loaded components of the SAM host software.

5 Prototype Implementations

Adding functionality is not just
a matter of adding code.

(Wietse Venema)

Contents

THIS chapter deals with the prototype implementations of three MoPS test applications. The presented applications have different requirements, but the same general framework (MoPS) has to be used. It is intended to show the flexibility of this modular concept. Beforehand, the general test flow for the MoPS system is explained, followed by applications with emphasis on the specific features used.

The first application is a high voltage (600 V) PFC boost converter. The second one is a low voltage (12 V) Point-of-Load (PoL) buck converter test application. The last application described is the functional test procedure for the MicroMoPS microcontroller target, which is also implemented in the same MoPS framework.

This chapter concludes with a summary and comparison of the projects and the common features used.

5.1 Test Plan Generation Work Flow

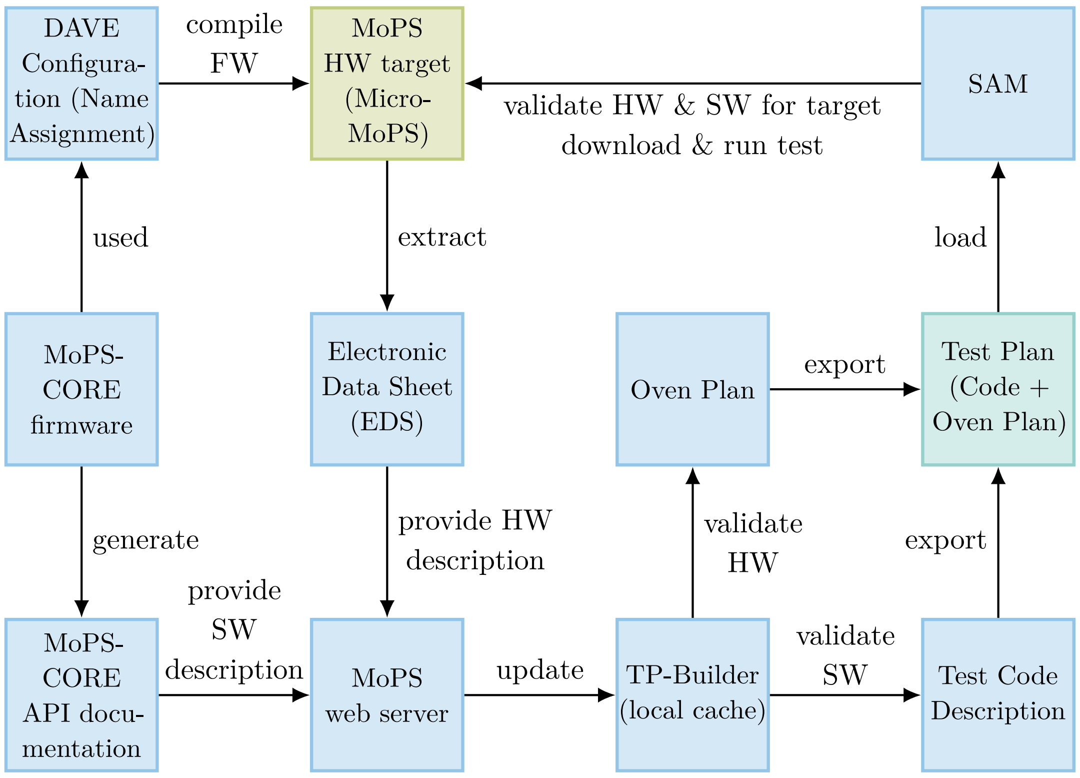


Figure 5.1: MoPS test procedure work flow

In order to run a test procedure on the MoPS system, a few steps are required. For documentation purposes, a test plan file needs to be stored on a network file share or in a database. The test plan is created by the TP-Builder, which requires information about the available software and hardware.

As can be seen in Figure 5.1, the source code of the MoPS-CORE firmware is used together with the hardware configuration settings (the DAVE handles and the custom name assignment as described in Section 4.1.3) to create the MoPS hardware target firmware. In this thesis, only one specific instance, the *MicroMoPS* hardware target is presented. However, multiple software versions and hardware revisions exist. Therefore, the MoPS-CORE project includes a script to generate the API documentation. The features describing the specific hardware target are obtained when the firmware is flashed onto the microcontroller. SAM or a special flash tool can be used to extract the on-chip generated EDS. Both the hardware description and the software description are put on the MoPS web server. Users can then access this documentation on-line.

The work flow described up to here is static and requires only very few updates, for example when a new hardware target is created. The following procedure may be carried out without updating the documentation or description files on the web server.

The lab engineers use the TP-Builder to create the test plan for their test application. The TP-Builder retrieves the documentation and uses it to provide features like Lua code auto-completion. The user specifies the oven plan and the test code in the test plan. The oven plan is used to assign a microcontroller target to the individual DUT and test carrier board (see Section 3.5). When the user decides to export the test plan, the TP-Builder validates both software and hardware descriptions and provides a valid test plan on success. Then, the test plan can be loaded into the SAM application running on the test system instance.

The exported test plan is stored in a JSON text format where SAM performs another validation of the loaded file using the TP-Builder (Section 4.3.2). Finally, the test code is distributed to the microcontroller targets and the test can be started.

5.2 Test Execution

Lua is used to execute script code within the LabVIEW environment. The similar configuration of both microcontroller and host application allows the user to create a hierarchical FSM layout (see Section 4.3.1). The main test FSM can monitor and control multiple child FSMs. This feature is required later for complex test setups, e.g. the intermittent load test of the PFC converter test.

5.2.1 Lua Test Code

In the first version of SAM, the use of Lua for LabVIEW (see Section 2.3.2) was investigated. When the stand-alone Lua interpreter library is used within C-based projects, circular calling is possible. Both C and Lua support interfaces to be called from the other language:

- the Lua script code can call C-based functions which operate with the Lua stack interface
- the C-based code can execute a Lua script which may call C-based library functions

Unfortunately, *Lua for LabVIEW* has a major drawback that originates from a missing language interface. LabVIEW does not provide an interface for C-based code to call VI functions directly. Thus, it is not possible for *Lua for LabVIEW* to have this feature either.

The proposed solution by the *Lua for LabVIEW* developers is to use a special *Execute Iteration* VI. There, the provided Lua script is executed until a function call to a LabVIEW provided function occurs. In such case, the Lua VM yields and returns a function index to the LabVIEW code, where the function call can be made. Afterwards, the Lua VM must be resumed by calling the same *Execute Iteration* VI again.

In order to run the Lua code from the test description (i.e. the FSM functions) in the Lua VM, a separate wrapper loop needs to be created in the Lua script (see Listing 5.1). First, the basic Lua library functions are defined (lines 1 to 3). The second block (lines 5 to 9) lists all the FSM states and the Lua code given by the test plan. Finally, the main loop (lines 11 to 17) is given. This script is compiled in the *Lua for LabVIEW* VM before starting the test execution. Compiling a script in Lua means to translate the textual representation into byte code.

When the test procedure is started, the iterative execution process starts. Then, the functions are created in the VM and the main loop is run. Within the main loop, the LabVIEW function `FSM()` (line 13) is executed. The function provided by LabVIEW evaluates the events from the test FSM and returns the name of the state the FSM is currently in, which is stored in the variable `func`. Next, one of the corresponding functions (line 6 to 9) will be called (line 15). When the FSM is finished, the LabVIEW function will return the Lua value nil, so that the evaluation on line 17 terminates the loop.

Listing 5.1: Lua main loop for SAM

```lua
-- SAM Lua library functions
function pairs() return nil end
function list_iter() return nil end

-- Test FSM functions
function IDLE() end
function SEND_START() sendEvent('start') end
function RUNNING() end
function SEND_STOP() sendEvent('stop') end

-- Main Loop
repeat
    func = FSM() -- get current state (=function name)
    if func then
        _G[func]() -- run FSM function
    end
until not func
```

Several limitations have been investigated for this rather complicated setup:

- It would be possible to run the FSM evaluation in Lua, however the Lua VM would then be in control continuously. In this case, the main loop must interrupt periodically to poll for received messages, since the Lua VM is run in a separate actor. However, adding another LabVIEW side call into the Lua wrapper loop does not simplify the setup. Further, each API must be implemented twice, in both the main loop script and the LabVIEW SAM software.

- When there is an error in the Lua script, the complete *Lua for LabVIEW* VM is cleaned up and the test crashes. There is an error message describing the problem, but the VM cannot be resumed. Since the test code is provided by lab users, an error must be handled more gracefully than terminating the VM.

- Executing each FSM state code in a separate Lua VM is not desired, because sharing variables between the different Lua VMs would not be possible. In addition, creating and cleaning up an independent Lua VM state for each state is a high performance hit.

5.2.2 MoPS Lua RPC Library

In order to overcome these limitations of calling LabVIEW functions from C, the concept of a Lua Remote Procedure Call (RPC) library has been presented [89]. The Lua script interpreter is built into a shared library for use with LabVIEW. By careful software design, the library can be compiled for both Windows and Linux platforms. The reason therefore is to be able to run the same Lua test code on the host level and on Linux-based embedded targets. This concept will be implemented in a master thesis project [25]. The library interface is very simple:

Open library When loading the library, a local TCP port has to be specified, where the library will connect to send RPC requests. Further, the Lua VM will be initialized and the base libraries will be loaded.

Register function A Lua function name can be registered to a numeric ID. Whenever the registered function is called, the library will perform an RPC to the previously specified TCP port. All parameters from the Lua stack are sent in addition to the numeric ID. After the RPC call, the results are placed back on the Lua stack.

Run code The provided string will be executed in the Lua VM running in the library. This library function is first used to load all the test code to the library and later to run the function code for each FSM state.

Close library When the test procedure has concluded, the library can be unloaded from memory.

By using TCP sockets and a generic library interface that is not tied to dedicated API functions, the C-based Lua VM can now call pre-defined LabVIEW functions. The working example is shown in Figure 5.2. The host application first has to open a TCP socket and to provide the port number. Then, the library can be opened and the API functions as well as the FSM state functions are loaded. Afterwards, the host application waits for the library to connect to the provided socket. Now, three parallel threads can be used to handle all communication without blocking each other:

Receive RPC This thread will wait for incoming RPC calls and forward them as actor message, so that the RPC call can interact with the private data of the test actor object. This data includes the FSM state of the microcontroller, test time and programmatic references to interact with the test FSM as well as displaying and storing test information.

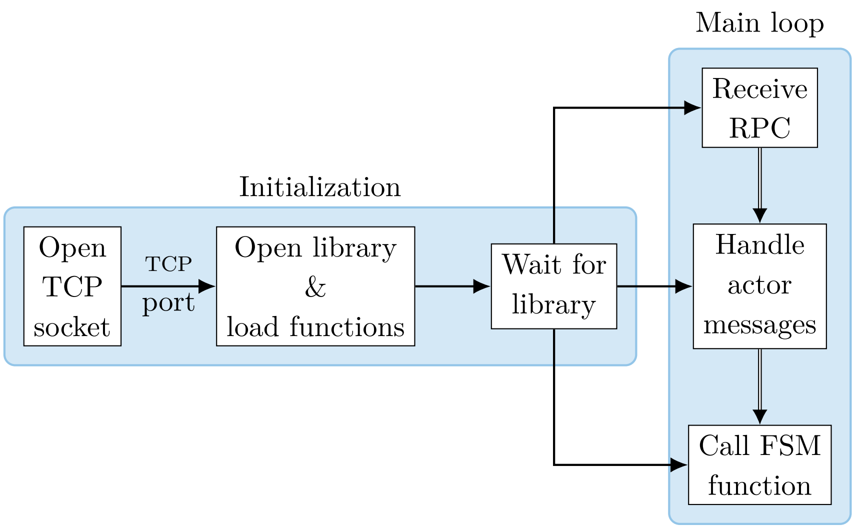


Figure 5.2: Interaction with the Lua RPC library

Handle actor messages This thread can receive actor messages and handle them without blocking the RPC reception or being blocked by calling an FSM state function.

Call FSM function The call of FSM functions is also split into a separate thread to be able to receive RPC calls and actor messages in the meantime. The actor message handling loop will send the function to be called after evaluating the FSM transitions.

5.2.3 FSM Visualization

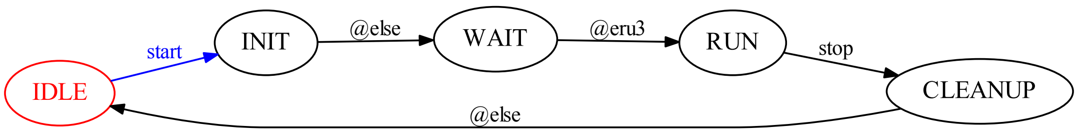


Figure 5.3: Graphviz rendered image

When running a test application, an often requested feature from the users is to visually see the FSM graph with the current state highlighted. The GUI of LabVIEW based applications cannot easily render custom drawings directly. However, LabVIEW has the capability to run external commands and display static images. Therefore, we use Graphviz (see Section 2.3.3) to provide a real-time update of the FSM status. The transition table described in Section 4.2.1 is converted into the *dot* graph description language (see Listing 2.2). Color markup can be added for the current FSM state and the currently set events. These scripts describing directed graphs are further processed with either the *dot* or the *neato* program, depending on the user preference. The programs return image data (Figure 5.3), which can be directly displayed on a LabVIEW GUI front panel.

5.2.4 Measurement Data Acquisition

Another very important feature for a modular test system is the possibility to acquire, display and store analog measurement signals. The values sampled by the ADC are quantized in a certain range (e.g. 12 bit describes a range between 0 and 4095) and represent analog values in a specific range (e.g. 0 V to 3 V). Hence, a scaling function is required to convert the digital values to voltages. Actual voltages are simpler to read for the users. Furthermore, it is significantly easier to perform a comparison as shown in Listing 5.2.

Listing 5.2: Analog measurement comparison

```lua
msr = adc:reads(1)     --    read scaled channel 1
if msr > 1.5 then
    setEvent("over_voltage")
end
```

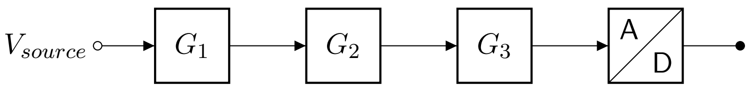


Figure 5.4: Analog signal path

Scaling of analog signals

However, typical voltages present at power test systems range up to several hundreds of volts. Thus, it is very common to have a voltage attenuation between the source of the signal (i.e. the input voltage of the converter) and the sink of the signal, the ADC. Furthermore, the scaling operation may not be linear. Therefore, the gain function can be described by an arbitrary function, allowing common arithmetic operations.

In the presented MoPS test system, each module (see Section 3.5) may add another attenuation to the overall signal path. Figure 5.4 displays the path of such analog signal. In this example, the source is attenuated three times, by the gain functions G_1, G_2 and G_3. Since the neighbouring stages may influence the gain function of a stage, the overall gain function is the result of the convolution of the individual functions.

A simplified model, however, takes only the input and output impedances of the gain stages into consideration (see Figure 5.5). Then, an additional resistive divider exists between two succeeding stages. Its mathematical description can be implemented for use on the microcontroller very easily.

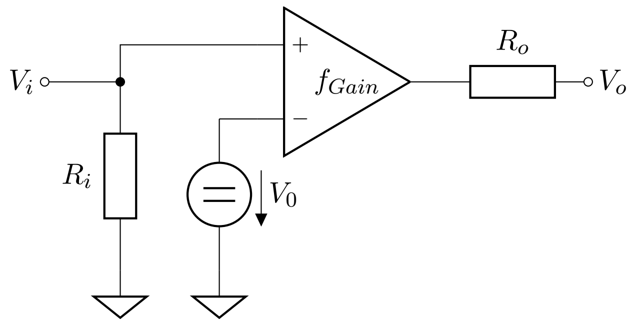


Figure 5.5: Analog gain model for a MoPS module

Transferring measured data

In order to transfer measurement data from the control and measurement nodes (i.e. the microcontroller hardware targets), a publish-subscribe scheme has been implemented. The microcontroller sends a publish message to the host application to inform that is has allocated a particular memory area for acquisition of a specific analog or digital signal. The publish message consists of the following elements as specified with the Lua test script:

- the label of the analog module
- the address of the memory block on the DRAM
- the size of the memory block on the DRAM
- the sampling frequency (used for proper display of the acquired signal)
- a list of enabled channels for this module

In the current implementation, the SAM host software automatically subscribes to all published signals and is able to request the measured data. In the future, the subscription may be performed upon request by the test procedure to reduce the network communication and the computational effort on the host.

The measurement data is requested by calling the API function `fetchMsr()` in the test FSM. The function accepts an optional table argument, where the desired hardware targets IP addresses may be specified to fetch data only from selected nodes.

In the current implementation, all subscribed signals will be fetched at the same time. Therefore, SAM will send a *READMEAS* message with the desired offset and length to the hardware target (see Section 3.2.3). The microcontroller node returns the data of the requested memory region to the host software. SAM receives this packet and inserts the read data into its published variables storage. Since a signal may consume more memory than is possible to transfer in a single UDP Ethernet frame, SAM will request further memory regions. When all the data for the signal is available, the binary data will be scaled according to the previously mentioned signal model.

After scaling, the analog signal is displayed on the GUI interface and may be stored for offline evaluation. In the near future, the data will also be available for online evaluation within the host software.

5.2.5 Instrument Control

As described in Section 3.1.1, the test actor reserves the instruments actors required for the operation of the loaded test plan. When an API function for instrument control is called, the call is redirected to the concerning instrument actor controlling the desired external instrument. Since these instruments are complex, they often report status and errors when communicating with them. For that reason, it is important to react to these reported conditions.

Typically, actor based systems are loosely coupled. Therefore, a path for returning messages, which is a synchronous call, is usually not desired. Thus, reading data from the instruments is not trivial. Measurement data obtained from the instruments can be periodically reported to the assigned test actor. However, there is typically a delay until the data is available for evaluation. In addition, when a command is sent to an instrument, the possible error is not immediately known in the test script.

As a result, it was decided to implement the instrument control API functions synchronously with a timeout to reduce the impact of blocking. Before the command is sent to the instrument actor, a return queue is created. Then, the message along with the queue reference is sent. The instrument actor carries out the requested command and the result values including a possible error information are sent back to the waiting test actor using the provided queue reference. The timeout is used in case the instrument command takes either too long to complete, or the instrument communication fails. In this case, this error is reported to the Lua engine. It is the responsibility of the test script to either ignore the error (in case of an soft-error condition) or perform the required actions to shut down the test procedure.

5.3 Power Factor Correction Boost Converter

The Power Factor Correction (PFC) boost converter test is a high voltage supply application stress test. The boost converter is a special kind of a power converter that converts lower voltage into higher voltage by using a PWM controlled half bridge and passive components. Such converters are usually present in switched mode power supplies and photovoltaic converters e.g. in the smart grid [90].

The modular load board (refer to Section 3.5) holds the *MicroMoPS* hardware target (Section 3.3.4). Further, a DUT board that features the actual half bridge application including the power semiconductor drivers and measurement circuits can be connected.

This application stress test has been proven to be a suitable evaluation for our distributed test concept. The requirements for this application test are described in the following:

- controlling the gate drivers of the power transistors
- measuring input and output voltages and currents
- performing a fast closed-loop PI control (100 kHz) to regulate the output current and voltage
- monitoring the DUTs' temperatures, voltages and currents
- identify connected load and DUT modules to provide individual calibrated analog scaling information

- operating multiple carrier boards in parallel at the same time
- controlling the external power supply and electronic load units for start-up procedures
- performing an intermittent load test through individual control of multiple hardware targets

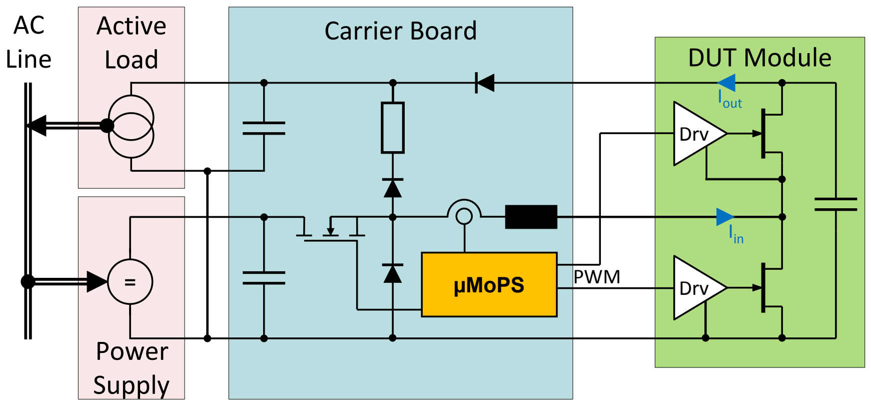


Figure 5.6: Simplified PFC boost converter application circuit

Therefore, as indicated in Figure 5.6, the DUT module is connected to a carrier board. The carrier board holds the application circuit components and connects the external instruments. A *MicroMoPS* hardware target (Section 3.3.4) is plugged onto each carrier board to perform the previously mentioned control and measurement tasks.

5.3.1 Static Load Test

In order to learn about possible degradation mechanisms in the DUTs, static load tests above the nominal operating condition are performed. Therefore, in situ monitoring as well as protection mechanisms must be individually configured for each test application.

Depending on the actual test application, varying sequences may be carried out to guarantee a specific run-time behavior during the test. In the case of the PFC application, a sample start-up routine is described in Figure 5.7. Parts of the procedure concern actions on the host layer (i.e. querying and configuring instruments as well as observing the FSM state of the microcontroller nodes) and the remaining actions (periphery modules interfacing the DUT and their measurements) are related to the hardware target close to the DUT.

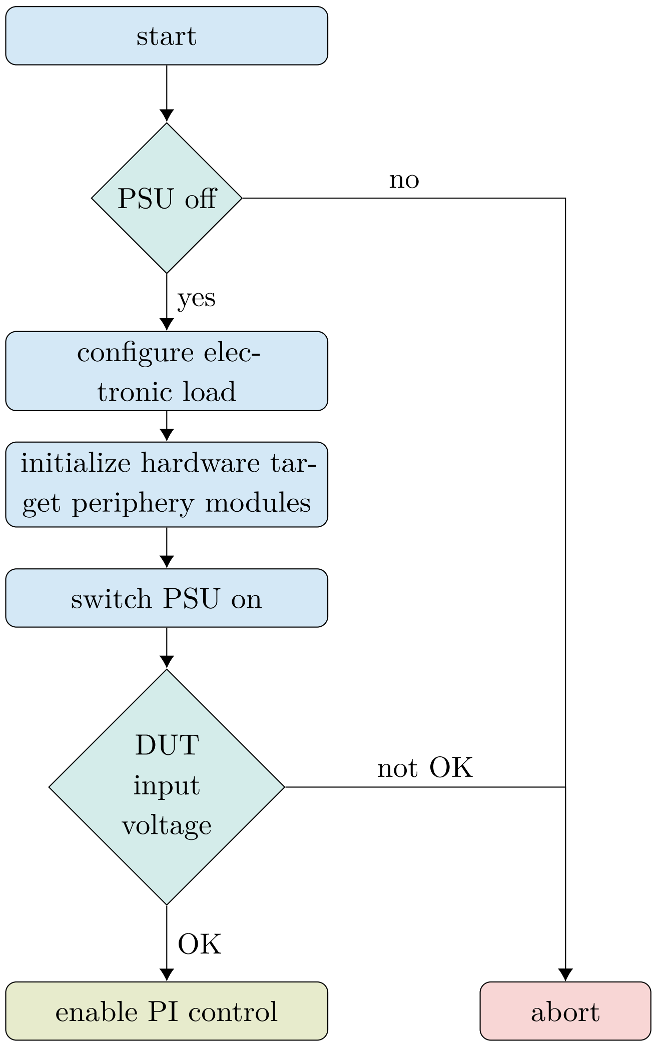


Figure 5.7: PFC boost converter sample start-up routine

As the start-up and the run-time sequences differ between different stress tests, it is important that these can be created and modified in a simple way. Therefore, the possibility to create the test program using the TP-Builder (Section 4.3.2) is very important. The actual test plan description for the PFC test can be seen in Figure 5.8.

In the left hand side, the host level test FSM is displayed. It controls the external instruments and monitors the microcontroller. In the right hand side, the microcontroller FSM can be seen. There, the start-up routine, the continuous measurement task (**MSR**) and the periodic report task (**REPORT**) can be recognized in the center. Various monitoring signals (over-voltage and over-current detection) are connected to

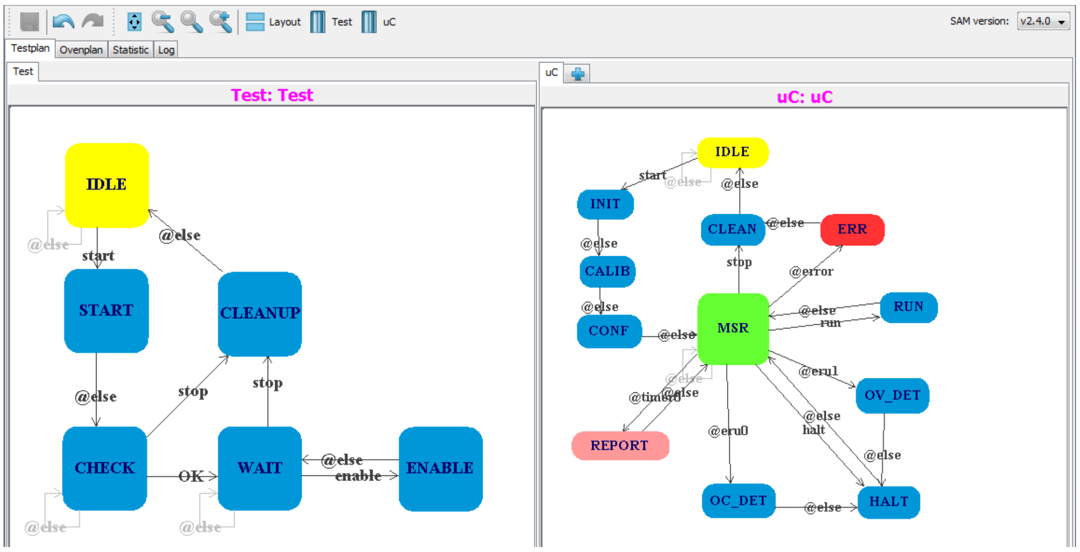


Figure 5.8: PFC test plan FSM created in TP-Builder

the event request inputs of the microcontroller allowing them to use the events *eru0* and *eru1* in the FSM for custom actions.

The Lua script and the sequence of states in the FSM can be changed very easily by adding and removing connections between the states. Successively, the updated test plan can be transferred to the microcontroller hardware target. Since the *MoPS-CORE* firmware of the microcontroller allows changing the FSM during run-time, the C-programmer can focus on providing a functional API. The test engineer who understands the needs of the test application can focus on the test procedure.

5.3.2 Intermittent Load Test

Another scenario in the PFC test application is the intermittent load test. In this test procedure, the individual test application is pushed to its thermal limit. Therefore, the DUTs are turned on for a few minutes to heat up and then turned off to cool down again (Figure 5.9). To save the number of required instruments, multiple tests can be run interleaved using the same physical connection to the external PSU and electronic load.

Therefore, a more sophisticated test plan compared to Figure 5.8 is required (see Listing 5.3 and Figure 5.10). Within the **MAIN** state, the host observes the attached microcontroller nodes' FSM state (Line 6). When its active time has elapsed (indicated

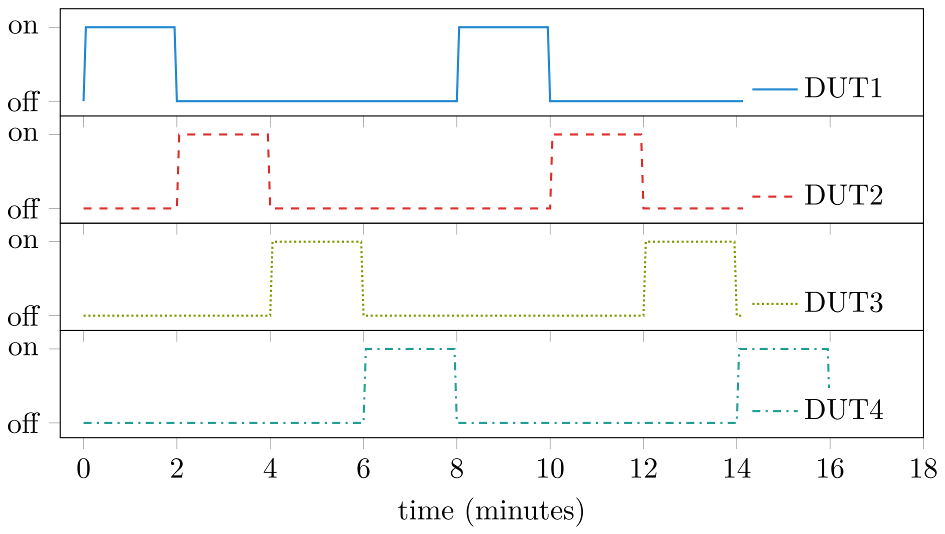


Figure 5.9: PFC intermittent load test

Listing 5.3: Intermittent load test (partial host script)

```lua
-- FSM INIT state:
duts = getNodes()
key, currentDut = next(duts) -- first DUT

-- FSM MAIN state:
state = getChildState(currentDut)
if (state == "ENABLE" or state == "ACTIVE" or state == "DISABLE") then
    -- currently running
elseif (state == "WAIT") then
    -- current DUT has finished, switch to next
    key = next(duts, currentDut) or next(duts)
    currentDut = duts[key]
    if (getChildState(currentDut) == "WAIT") then
        sendEvent("enable", {currentDut}) -- turn on waiting
    end
else
    -- error: switch to next
    print_log("DUT state "..state)
    key = next(duts, currentDut) or next(duts)
    currentDut = duts[key]
end
```

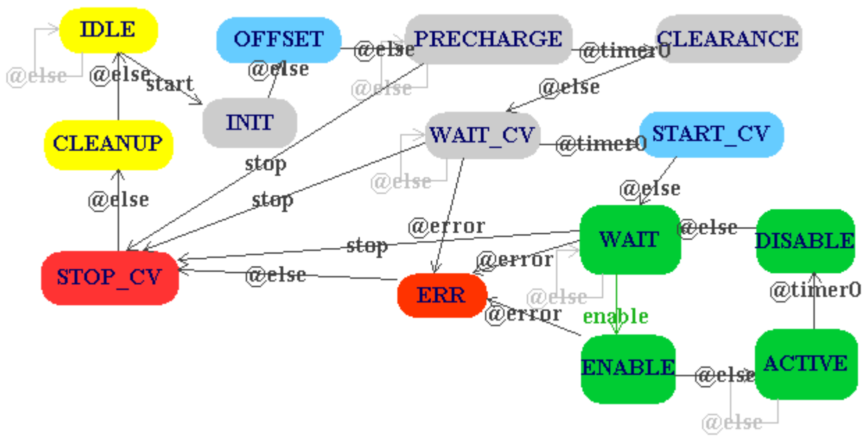


Figure 5.10: PFC intermittent microcontroller test plan

by the occurrence of the *@timer0* event in Figure 5.10) and it has returned to the **WAIT** state (Line 9), the next DUT is selected and enabled (Lines 11 to 15). Using such a procedure, a non-overlapping power utilization from the supply is guaranteed.

5.4 Point-of-Load Converter

The PoL converter is a DC-DC buck converter application. The aim is to provide the low-voltage (1.2 V) supply for notebook, desktop and server CPUs. The PoL test application [82] is similar to the previously described PFC application. However, the DUT already contains both half-bridge power transistors. The voltage control of the DUT is performed using a dedicated analog control chip. Therefore, the requirements are as follow:

- interfacing the analog controller to provide set values
- communicating with the DUT using analog and digital interfaces
- DUT board identification for individual analog scaling and tracking
- measuring DUT voltage, current and temperature
- monitoring of DUT status, detecting its failure and controlling the guard module
- operating multiple DUTs in parallel

- performing an intermittent load test through individual control of the external loads input channels

The test application circuit is given in Figure 3.10. The task of the control module is to observe the DUT and report measurements to the host application. The host application controls the power supply and the external instruments.

5.4.1 Static Load Test

The PoL application requires a very similar start-up procedure. However, a guard module is placed in the supply path before the actual application to prevent the DUT from destruction on failure. This guard measures the current flow, compares it to a pre-set value and switches off in case of an over-current event. Therefore, the hardware target microcontroller also needs to check and configure this guard module. In addition, the host software needs to check the external instruments.

During the static test, parametric values are obtained by performing analog measurements using the built-in ADCs on the microcontroller node. These values are stored in DRAM and transferred to the host application for visualization and storage for offline evaluation. Possible derivations during the run-time of the test can be monitored in situ – i.e. the devices need not be removed from the test application. Thus, the test can be run without interruptions. The reproducibility of the stress test is improved and the test time decreased.

5.4.2 Dynamic Load Test

The main test application for the PoL converter is the dynamic load test. Within the typical application, a notebook or desktop CPU usually switches between idle and full load in several steps. Therefore, the test procedure has to apply specific load profiles to the DUTs.

Such an example profile is given in Figure 5.11. During one load cycle, the load reaches 25 %, 50 %, 75 % and 100 % of the maximum load for 10 %, 10 %, 1 % and 5 % of the cycle time respectively. The remaining time, the load is at 5 % of the maximum load. The load cycles are then repeated during the overall test time.

One can imagine that this arbitrary load profile is changed very often. Through the MoPS system, the test engineer receives the full flexibility of creating the test plan – both FSM and Lua script. In the example above, a list of values is defined, where each value is sent to the external instrument. Therefore, the host software and the microcontroller firmware remain fixed.

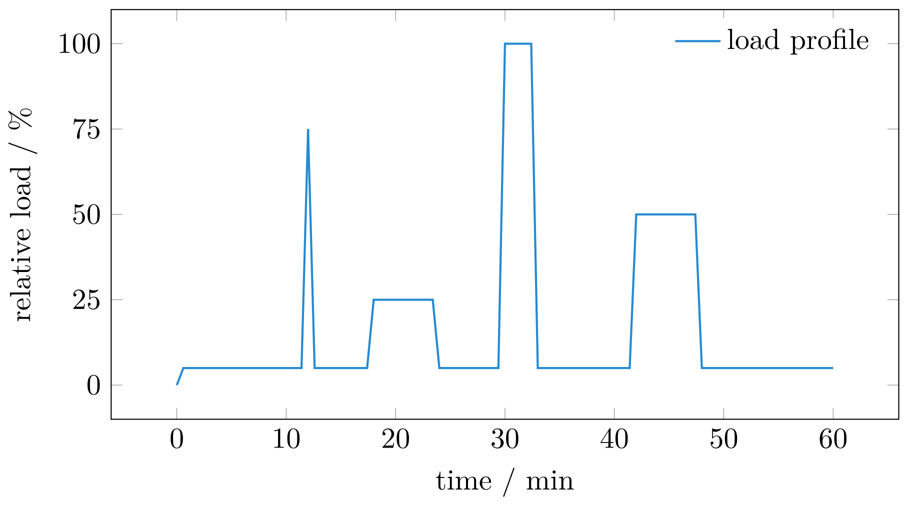


Figure 5.11: PoL test intermittent load profile

5.5 MicroMoPS Test

In contrast to the two previous application tests, the *MicroMoPS test* has been created to evaluate the functionality of said hardware target. The MicroMoPS (Section 3.3.4) consists of many components soldered onto its PCB. Parametric variations or wrong components may have a serious impact on utilizing such component in a stress test. Therefore, it is important to have functional control and measurement nodes available. Up to now, more than 100 pieces of the MicroMoPS have already been manufactured. Testing and verifying them manually is very tedious. The flexibility of the MoPS system architecture allows the creation of a test procedure for this task.

A special carrier board, the *MicroMoPS test board*, has been designed. It features connectors to interface each electrical signal from the MicroMoPS hardware target, some analog filters and DSD modulators as well as a power supply connection. The board can also be used in the lab for prototyping a test application before the actual carrier board is designed.

5.5.1 Test Procedure

In order to test each individual peripheral module of the tested MicroMoPS, a set of three test boards is required due to the large number of available functions. As displayed in Figure 5.12, two auxiliary boards – so-called *golden samples* – are used to apply stimuli and read responses from the MicroMoPS located between them. Some of

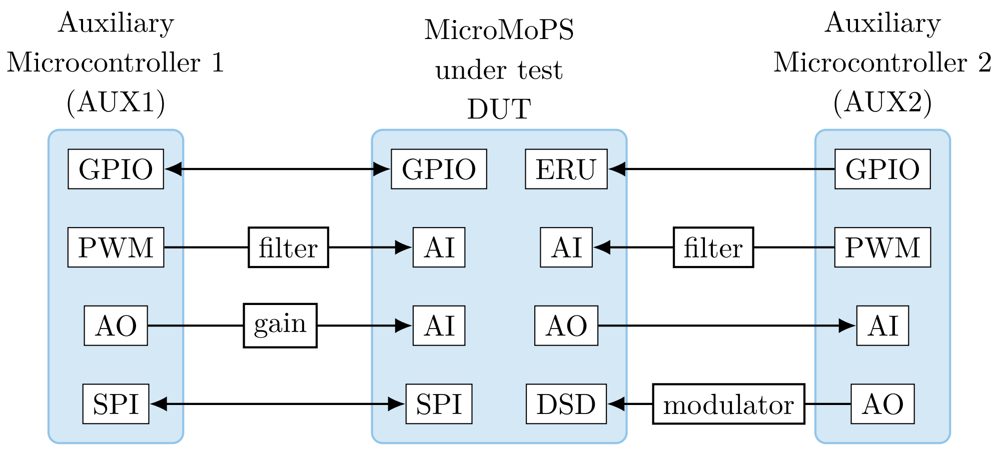


Figure 5.12: MicroMoPS test setup

the module interfaces require a transformation of the applied signal. Passive low-pass filters are used to flatten the digital PWM output and sigma-delta analog converters are provided to sample the analog voltages.

Since multiple microcontroller nodes are used within the same test, the SAM host software can be used to synchronize the individual hardware targets. Using the FSM approach presented in the previous chapter, such a test sequence can easily be modelled and created:

The host application tells the three nodes to initialize their modules. Upon completion of the initialization, the nodes change their FSM state and SAM can successively send an event to the AUX1 node, commanding it to apply the first pattern. The DUT node then reads in the pattern and the host application compares the results. In such way, all modules can be tested one after each other. While executing the test, the microcontroller nodes and the host test FSM can print the results into log files. Thus, a summary can be quickly obtained.

5.6 Chapter Summary

To run a test procedure using the MoPS system, many functional components need to work together. The overall test plan work flow is as follows: The MoPS-CORE firmware is compiled together with the configuration of the periphery modules. Both the software (custom Lua API) and the hardware documentation (the EDS) are provided on the intranet web server for viewing by the users and for the TP-Builder tool. Using the TP-Builder, the test plan can be created and verified for the specific hardware target to be used. After exporting the graphical test description, this test plan can be loaded in the SAM host software. It will download the test to the microcontroller target and run the test.

The utilization of the LuaVIEW library has been described and discussed. It is lacking some key features for a reliable use within the proposed SAM application. Therefore, a custom solution for the Lua library using a generic TCP interface (the Lua RPC library) has been described.

Users can follow the execution of the FSM in a graphical way. For that reason, the Graphviz tools are used to render images of the current system state.

Analog signals measured in the test application are described using a multi-stage signal model. The individual stages are defined with a gain function and junction impedances. Thus, the microcontroller implementation performs reasonably well. Waveforms of analog signals can be acquired and stored in the DRAM of the Micro-MoPS hardware target. The host is notified using a publish-subscribe mechanism. The measurement data are requested by the SAM host application. After the transfer, they are scaled and displayed on the GUI as well as stored for offline evaluation.

External instruments are also controlled using Lua API functions within the Test Actor of the SAM software. In order to decouple the actors and still retain a link for reading measurement data, a separate return queue is used. When the instrument fails to return the data after a timeout, the error is reported and the Lua test script continues its execution.

Three different test applications are described. Two of them – being power application stress tests – are similar. For that reason, the use of the MoPS system provides a solid platform. Visual code representations like flow charts can be easily mapped to the graphical test plan using the TP-Builder application. Therefore, creating, editing and documenting test procedures becomes simple and intuitive. Furthermore, an updated test plan can be immediately loaded and used without the need to change neither the microcontroller firmware nor the SAM host application. The introduction of communicating FSMs enables the triggering of transitions on the microcontroller. Thus, multiple hardware targets can be synchronized and interleaved to e.g. not exceed the possible peak power consumption.

The MicroMoPS test demonstrates the capabilities of the FSM communication. It has distinct test plans for the microcontroller targets. The host application sends an event to one of the auxiliary controllers in order to signalize that it must apply a specific pattern. The tested MicroMoPS board is then requested to read the provided signals and provide the results to the host application, where they can be verified.

6 Results & Discussion

If your experiment needs statistics, you ought to have done a better experiment.

(Ernest Rutherford)

THIS thesis introduces a new, modular concept for power electronic stress testing. Previous systems modularity mainly focused on the availability of COTS hardware for PXI systems. However, the software for these legacy systems has been created for a single purpose with little re-usability for other projects.

6.1 Results

The presented MoPS test concept has been successfully implemented on the host level by the SAM application and on the embedded target level by the microcontroller firmware MoPS-CORE. The host acts as a server for distributing the test procedures as well as for data collection from the hardware targets. In addition, it manages the loaded test procedures by running Lua script code to synchronize multiple microcontroller hardware targets and by controlling external instruments.

The MoPS test system is the framework for implementing the stress test applications. The use of a small number of fixed components allows a small software development team to maintain the test systems and to incrementally implement new features. The required flexibility for the test procedures is provided by the embedded Lua scripting language. By allowing the user to write his/her own test plans, custom FSM descriptions and Lua scripts can be loaded and executed. Therefore, new physical test system instances can be set up with little to no additional effort with respect to the software design since the test procedures are not implemented as plug-ins, but loaded as Lua script code.

The actions to be carried out during the test procedures have been modelled with the FSM model. The benefit of using the FSM description is the possible inclusion of external events (like interrupts from the microcontroller periphery) into the test procedure description. In this thesis, a generic test software has been presented that enables the user to modify the FSM structure.

In order to simplify the creation of these test plans, the TP-Builder tool has been created. It gives direct access to the API documentation of the host and hardware target software. Furthermore, the TP-Builder is capable of presenting code creation hints like auto-completion. In addition, the FSM model and the Lua script syntax are checked. The syntax check takes the selected microcontroller hardware target and the host system configuration into account. Therefore, the TP-Builder is also used to check the validity of a test plan before it is loaded onto the microcontroller hardware target.

6.1.1 Prototype Test Applications Summary

In Chapter 5, the flexibility of the MoPS system architecture is demonstrated. The same host software and microcontroller software are used to run the PFC test (Section 5.3), the PoL test (Section 5.4), and the MicroMoPS test (Section 5.5).

During the evaluation of the PFC test application, the distributed architecture proved to be capable of handling the requirements. Having multiple independent agents that control smaller tasks are easier to configure, test and observe. The introduction of the communicating FSMs enables sending messages also to the hardware targets. This describes the extension of the actor framework to the microcontroller units.

During the execution of the PFC test procedure, several analog quantities are measured (see Figure 6.1). The tested DUTs convert 1 kW each. The input voltage is set to 200 V. The input current of the power converter and the temperature of the converters DUTs are plotted. The values for the current are acquired with the internal ADC module of the XMC4500 on the MicroMoPS hardware target. The DUT temperatures are measured using delta-sigma modulators and acquired using the DSD module of the microcontroller. We can observe that individual devices are shut down by the guard module (see Section 3.5) as soon as an over-current event is detected (usually because the semiconductor devices fail in a short circuit). Figure 6.1 shows such events: DUT5 fails after about 12 h test time. Before this event, a rise of the DUT temperature can also be observed. The users need this kind of measurement data to investigate the performance of the DUTs under real application conditions without having to stop the test for off-line DUT characterizations. Feedback from the MoPS system users has been very positive and confirms the benefits provided by the software architecture presented in this thesis.

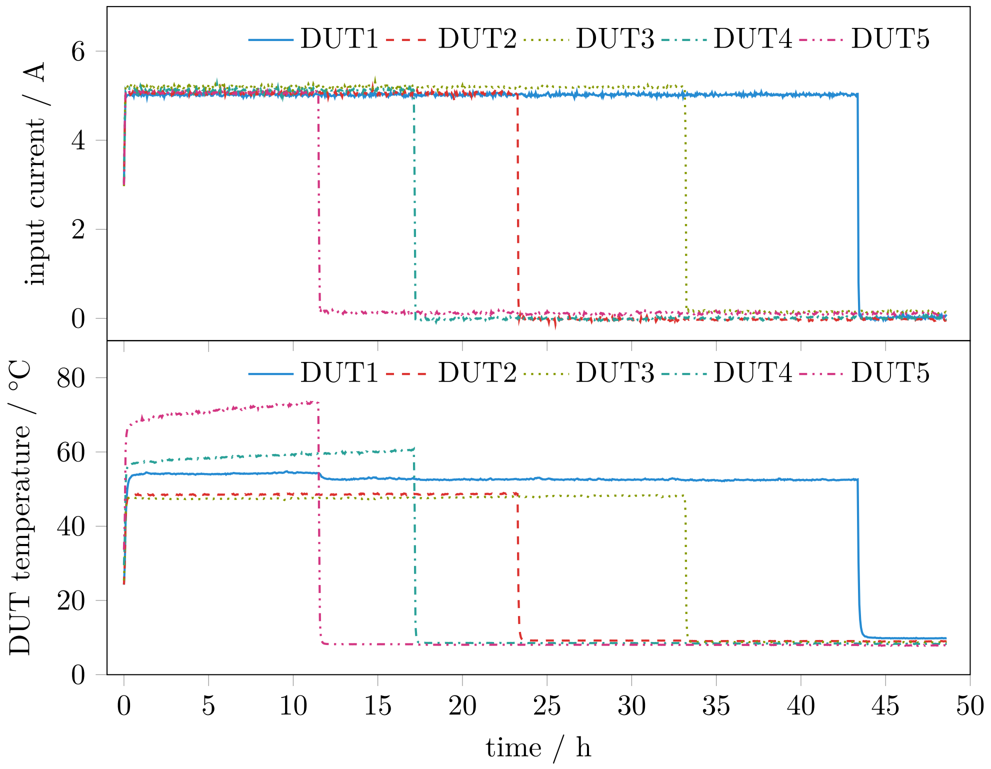


Figure 6.1: PFC test measurements

6.2 Discussion

6.2.1 Usability

The users appreciate various features of the MoPS system architecture. First of all, reading and storing measurements was mentioned as one of the key features of the presented test system. The lab engineers appreciate using the system and creating the test procedures using the Lua script language, which makes them not directly dependent on the software engineers anymore. There is more effort required to create the FSM structure than filling in a plain spreadsheet with test parameters for a legacy test system (e.g. ACUTE). However, the users value the added functionality of non-linear test procedures. The described Micro-MoPS hardware target is comparable to a digital storage oscilloscope, albeit with higher resolution (12 bit) and lower sample rate (100 kHz). However, the signal quality largely depends on the underlying application boards. Furthermore, loading the test plan works properly and the speed of prototyping test applications earns a very high satisfaction with our users. On the other hand, we received feedback that downloading test plans to the hardware targets could be faster. In addition, the GUI sometimes is considered to be not responsive enough.

6.2.2 Host Performance

While executing the PFC stress test (see Section 5.3), the performance of the SAM host application was evaluated. Rendering and displaying the FSM graphs as images within the SAM host software (Section 5.2.3) turned out to be very slow. When running a test plan with a few connected hardware targets, this behavior did not impact the test execution. On the other hand, having more than 8 hardware targets connected to one host caused severe performance issues. The FSM status images were updated synchronously with the reception of the status messages from the hardware targets. However, the LabVIEW application consumed the majority of the available CPU time for rendering these images and meanwhile blocked the processing of further messages to the respective Node Actor.

Therefore, a configuration option to disable the real-time update of the current nodes' FSM states is provided. In this case, a static image is displayed – the FSM graph is drawn only once during loading of the test plan and is not updated during run-time. The current FSM state is indicated in a text field on the GUI, thus the user can still track the execution of the test procedure. Currently, we do not observe any more scalability problems running 24 microcontroller targets with one single host application.

6.2.3 Microcontroller Performance

On the XMC microcontroller, the closed loop PI control required for the test application cannot be done in the Lua VM due to performance reasons. To achieve the required 100 kHz update rate for the PI control, the algorithms are implemented in C and are, therefore, part of the firmware image. The flexibility during the run-time and test execution is provided by setting the PI-controller's parameters through the Lua API of the PI-controller instance.

In addition, the measurement data acquisition is limited by missing features of the DMA controller within the used XMC microcontroller. The DMA controller is used to collect the measurements from the result registers and transfer them to the DRAM. Unfortunately, it is not possible to write to a ring-buffer structure because the address wrapping cannot be configured. The DMA controller is missing a compare register to check if the end address of the destination block has been reached in order to reset to the start address of the destination block. Therefore, an ISR function has to be called after each DMA transfer to perform this check using the main CPU impacting the microcontroller application. As a result, the analog sampling rate is limited to about 200 kHz.

6.2.4 Lua Host Library

The PoL test heavily relies on instrument control. However, using the LuaVIEW library (Section 2.3.2), the Lua interpreter was not reliable. As described in Section 5.2.1, an error raised in the Lua script caused LuaVIEW to clean the complete Lua stack without the possibility for a proper shutdown of the test. This triggered the investigations of creating a custom Lua-based library for integration in LabVIEW (see Section 5.2.2). According to our first tests, the proprietary library outperforms LuaVIEW by a factor of 10 to 20.

6.2.5 Large Test Plans

The MicroMoPS test plan is very large with many states in the FSM and a lot of Lua script code. Due to the limited microcontroller memory, it was not possible to fit the total procedure onto the microcontroller. Therefore, the test program had to be split into three separate programs: The first one tests the digital interfaces and the remaining two test the analog interfaces. Meanwhile, a new chip of the XMC series has been released: the XMC4700 features up to 352 KiB RAM – twice the amount of the currently used XMC4500. However, a more complex program might still not fit on these large microcontrollers. On the other hand, typical application tests do not require all these functions at the same time. Furthermore, the available memory will increase with future microcontroller devices.

6.2.6 Measurement Data Acquisition

Through the provided Lua API functions, the users can select the analog channels they would like to use in their test application. The sampling frequency can easily be configured and is only limited by the overall CPU time of the microcontroller ISR handling the DMA transfers.

By switching from the CAN based communication channel to Ethernet, the effective measurement data transmission rate has been improved by two orders of magnitude. These results are attributed to the faster physical link speed (100 Mbit/s instead of 1 Mbit/s). Furthermore, the increased payload size of the transmitted packets results in fewer packets required to send the measurement data to the host.

The modeling of the analog signal path (see Section 5.2.4) provides the possibility to specify the individual gain functions of the modular components. When the raw data is received at the SAM host software, it is automatically scaled and displayed on the GUI (see Figure 6.2) and stored for offline evaluation.

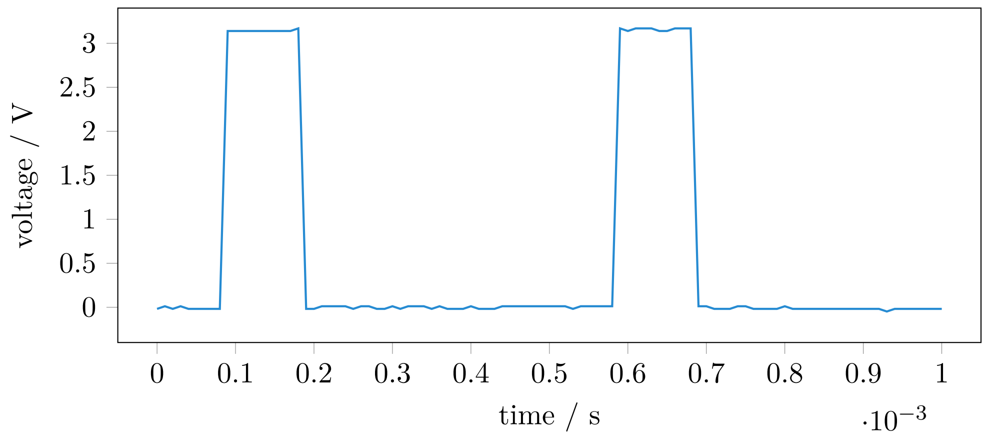


Figure 6.2: Analog data acquisition

7 Conclusion & Outlook

Science is the acceptance of what works and the rejection of what does not. That needs more courage than we might think.

(Jacob Bronowski)

THE task of this research project has been to integrate the components of a stress test system (see Figure 1.1) into a framework that is capable of controlling individual devices while collecting, displaying and storing measurement data. The configuration of these actions (i.e. the system programming) must be easy and flexible.

7.1 Conclusion

Throughout this thesis, numerous problems have been investigated and solved in order to provide a flexible way to create and update test software. A distributed architecture – that allows integration of individual distributed control and measurement nodes (i.e. the hardware targets) – has been created and its application has been demonstrated. The flexibility of the MoPS system architecture is twofold.

- Configuration files are provided to set dependencies for external instruments. Further, utilized tools can be set or removed depending on the users requirements. Therefore, it is possible to change the test system in a simple way.

- The test procedures executed on the test system are designed according to the FSM model. Within the FSM states, custom Lua script code can be defined. Therefore, basically all kind of test procedures are possible. To provide even more functionality, the Lua based API can easily be extended.

By combining the configuration possibilities and the Lua scripting support with hardware interaction on the embedded level, the MoPS test system provides a solid development platform for power stress test applications. The configuration of such a modular test system is straight forward. By using the JSON file format instead of XML, the configuration data can be expressed in a clearer way. Reading this data is simpler for both humans and machines.

The microcontroller firmware is created in a flexible way, so that it is possible to use the Lua script and FSM functionality on any hardware target based on the XMC4000 microcontroller family. By using C configuration files, the pin to name assignment can be easily performed without the need to have a knowledge of the DAVE development environment. The firmware image is currently flashed manually to the target. However, through the means of Lua scripting and dynamically changeable FSM structures, the firmware only needs to be updated when fixing a bug or when adding new features.

While writing the test plan, the user can focus on the physical implementation of the test application. High level Lua script commands are used instead of dealing with the low-level microcontroller registers. Thus, the development of such test applications has become much faster and easier while maintaining maximum flexibility.

7.2 Outlook

Even though the proposed architecture is working and being used for actual stress test control, there are still several topics left for improvement and future investigations. Since the main goal was to provide a functional system architecture, the real-time requirements for the communication channel and the embedded microcontroller were not in the key focus of this PhD project. To investigate possible improvements of real-time capabilities on the hardware targets and synchronization between those microcontrollers, a further research project has been recently started.

Furthermore, the usability of some tools leaves room for improvement. Especially the configuration of a hardware target currently requires expert knowledge about the internal structure of the used microcontroller. A future version of the MoPS-CORE DAVE-App (see Section 4.1.4) that uses the DAVE scripting engine for the hardware solver is already planned.

Due to performance reasons, the state change of the microcontroller FSM is only polled by the host software. Unfortunately, this behavior results in larger delays for detecting specific states of the microcontroller and sending messages to switch to the next state. However, we see the possibility of sending events from the microcontrollers to the host application, thus eliminating the polling requirement.

Porting the firmware to any ARM Cortex-M based microcontroller or CPU is possible. Switching to the latest XMC4700 microcontrollers to immediately gain an increased memory and a faster CPU is a simple task. In addition, we will evaluate the migration of our custom Lua modules to the Xilinx Zync processor embedded in the NI System on Module[1].

A topic that has not been dealt with in this thesis is the automated deployment of firmware updates of the microcontroller. A boot loader for use with the Ethernet communication channel will be implemented in the MoPS-CORE firmware in a soon-to-start master thesis project.

The embedded measurement nodes (i.e. the microcontrollers) communicate over standard Ethernet. Currently, the communication is not encrypted due to performance and debugging reasons. If desired, encryption can be enabled on to the channel e.g. by using DTLS. On the other hand, DoS are always possible on these microcontrollers. Therefore, it is currently still mandatory to have separated networks between the office LAN and the test system LAN to mitigate attack scenarios.

[1] `http://www.ni.com/som`

Bibliography

[1] *AEC-Q100 Failure Mechanism Based Stress Test Qualification for Integrated Circuits*, Automotive Electronics Council Std., Rev. H, Sep. 2014. [Online]. Available: http://www.aecouncil.com/AECDocuments.html 1, 2, 4, 54

[2] *JESD22 Series - Reliability Test Methods for Packaged Devices*, Accessed 2016-02-29, JEDEC Solid State Technology Association Std. [Online]. Available: http://www.jedec.org 1, 2

[3] A. Steininger, "Testing and Built-in Self-Test – A Survey," *Journal of Systems Architecture*, vol. 46, no. 9, pp. 721–747, Jul. 2000. 1

[4] O. Bluder, M. Glavanovics, and J. Pilz, "Applying Bayesian mixtures-of-experts models to statistical description of smart power semiconductor reliability," *Microelectronics Reliability*, vol. 51, no. 9-11, pp. 1464–1468, 2011. 3

[5] O. Bluder, "Prediction of Smart Power Device Lifetime based on Bayesian Modeling," Ph.D. dissertation, Alpen-Adria-Universität Klagenfurt, 2011. 3

[6] K. Plankensteiner, O. Bluder, and J. Pilz, "Bayesian Network Model with Application to Smart Power Semiconductor Lifetime Data," *Risk Analysis*, 2015. 3

[7] K. Plankensteiner, "Predicting Censored Semiconductor Lifetimes with Bayesian Regression Models Using Mixtures of Experts," in *Young Statisticians' Meeting*, 2011. 3

[8] *Temperature, Bias, and Operating Life*, JEDEC Solid State Technology Association Std., Rev. D, Nov. 2010. 4

[9] K. T. Feng, L. Rushing, P. Canfield, and L. Flores, "Determination of reliability on MOCVD grown InGaP/GaAs HBT's under both thermal and current acceleration stresses," in *Proceedings of the 2001 GaAs Reliability Workshop*. IEEE, 2001, pp. 159–180. 5

[10] S. Singhal, T. Li, A. Chaudhari, A. Hanson, R. Therrien, J. Johnson, W. Nagy, J. Marquart, P. Rajagopal, J. Roberts, and et al., "Reliability of large periphery GaN-on-Si HFETs," *Microelectronics Reliability*, vol. 46, no. 8, pp. 1247–1253, Aug. 2006. 5

[11] H. Eder, “Development of a repetitive clamping test system hardware for smart power switches,” Master’s thesis, Carinthia University of Applied Sciences, Villach, Austria, 2006. 5

[12] M. Glavanovics, H. Köck, V. Košel, and T. Smorodin, “Flexible active cycle stress testing of Smart Power switches,” in *European Symposium on Reliability of Electron Devices, Failure Physics and Analysis.* Elsevier B.V., Radarweg 29, 1043 NX Amsterdam, The Netherlands, 2007. 5

[13] M. Glavanovics, H.-P. Kreuter, R. Sleik, and C. Schreiber, “Cycle Stress Test Equipment for Automated Short Circuit Testing of Smart Power Switches According to the AEC Q100-012 Standard,” in *Proceedings of the 13th European Conference on Power Electronics and Applications*, 2009. 5

[14] R. Sleik, “Investigation of Integrated Protection Functions in Smart Power Switches based on the Development of an Advanced Control and Measurement Interface,” Master’s thesis, Carinthia University of Applied Sciences, Villach, 2010. 5

[15] B. Steinwender, “In-situ characterization of smart power switches during cycle stress testing,” Master’s thesis, Carinthia University of Applied Sciences, Villach, 2010. 5

[16] M. Bertocco, F. Ferraris, C. Offelli, and M. Parvis, “A Client-Server Architecture for Distributed Measurement Systems,” *IEEE Transactions on Instrumentation and Measurement*, vol. 47, no. 5, pp. 1143–1148, 1998. 6

[17] A. Pirker-Frühauf and M. Kunze, “A novel methodology to combine and speed-up the verification process of simulation and measurement of integrated circuits,” in *Proceedings of the 2008 IEEE AUTOTESTCON Conference*, 2008. 6

[18] A. Pirker-Frühauf, W. Gallent, M. Kunze, and G. Pelz, “Acceleration of IC verification process using advanced flexible modular measurement systems and software architectures,” in *Proceedings of the 2008 IEEE Instrumentation and Measurement Technology Conference.* IEEE, 2008, pp. 1845–1847. 6

[19] B. Steinwender, S. Einspieler, M. Glavanovics, and W. Elmenreich, “Distributed power semiconductor stress test & measurement architecture,” in *Proceedings of the 11th IEEE International Conference on Industrial Informatics*, Jul. 2013, pp. 129–134. 6, 30, 31, 37, 41

[20] S. Einspieler, "Distributed Microcontroller Network for Smart Power Device Life Testing and In-Situ Monitoring," Master's thesis, Alpen-Adria Universität, Klagenfurt, 2013. 8, 38, 49

[21] G. Palatin, "Entwicklung eines intelligenten In-situ Testsystems zur Lebensdaueruntersuchung von Power-MOSFETs," Bachelor's thesis, Carinthia University of Applied Sciences, 2013. 8

[22] ——, "Firmware eines intelligenten In-situ Testsystems zur Lebensdaueruntersuchung von Power-MOSFETs," Bachelor's thesis, Carinthia University of Applied Sciences, 2013. 8, 48

[23] K. Plankensteiner, "Test Plan Generation and Verification for a Modular Power Stress Test System," Master's thesis, TU Graz, 2015. 8, 79

[24] G. Palatin, "User-Configurable Firmware Generation for a local Microcontroller Node in a Modular Semiconductor Stress Test System," Master's thesis, Alpen-Adria-Universität Klagenfurt, 2015. 8, 71

[25] S. Bauer, "Implementation of a Real-Time Environment for HTOL Qualification Systems," Master's thesis, Alpen-Adria-Universität Klagenfurt, 2016, in writing. 8, 90

[26] J. Weigmann and G. Kilian, *Decentralisation With Profibus DP/DPv1*, 2nd ed., S. AG, Ed. Publicis Corporate, 2003. 11

[27] M. Felser, *PROFIBUS Manual*, 2012. 11

[28] *Industrial communication networks - Fieldbus specifications*, International Electrotechnical Commission Std. 11, 17

[29] J. Ferreira and J. A. Fonseca, *The Industrial EElectronic Handbook, second edition – Industrial Communication Systems.* CRC, 2011, ch. Controller Area Network, p. 31. 11

[30] *ISO 11898-2:2003: Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit*, International Standards Organisation Std., 2003. 11

[31] Robert Bosch GmbH, "CAN Specification 2.0," 1991. 11, 12

[32] F. Hartwich, "CAN with flexible Data-Rate," in *Internation CAN Conference*, 2012. 12

[33] K. Tindell and A. Burns, "Guaranteeing message latencies on control area network (CAN)," in *Proceedings of the 1st International CAN Conference*, 1994. 12

[34] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, Jan. 2007. 12

[35] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*, 5th ed., H. Michael, Ed. Prentice Hall, 2010. 13, 14, 15

[36] *ISO 15765-2:2011: Road vehicles – Diagnostic communication over Controller Area Network (DoCAN) – Part 2: Transport protocol and network layer services*, International Standards Organisation Std., 2004. 13

[37] O. Pfeiffer, A. Ayre, and C. Keydel, *Embedded networking with CAN and CANopen*. Copperhill Media, 2008. 14

[38] S. Biegacki and D. VanGompel, "The application of DeviceNet in process control," *ISA Transactions*, vol. 35, no. 2, pp. 169–176, Jan. 1996. 14

[39] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM*, vol. 19, no. 7, pp. 395–404, Jul. 1976. 14, 16

[40] G. W. Brock, *The Second Information Revolution*, G. W. Brock, Ed. Harvard University Press, 2003. 14

[41] S. Krywult and C. Steiner, "Survey on Present Real-Time Ethernet Solutions," 2004. [Online]. Available: http://www.vmars.tuwien.ac.at/documents/intern/2010/rt-ethernet.pdf 16

[42] G. Marsal and D. Trognon, *The Industrial EElectronic Handbook, second edition – Industrial Communication Systems*. CRC, 2011, ch. Industrial Ethernet, p. 31. 16

[43] EtherCAT Technology Group, "EtherCAT - the Ethernet fieldbus," Accessed 2016-02-12. [Online]. Available: http://ethercat.org 17

[44] G. Cena, A. Valenzano, and C. Zunino, *The Industrial EElectronic Handbook, second edition – Industrial Communication Systems.* CRC, 2011, ch. EtherCAT, p. 38. 17

[45] S. Augarten, *State of the Art.* Ticknor & Fields, 1983, ch. The Most Widely Used Computer on a Chip - The TMS 1000, p. 38. 18

[46] R. Obermaisser, P. Peti, W. Elmenreich, T. Losert, I. Wen, H. Kopetz, and Clifford D. Fung, "Monitoring and configuration in a smart transducer network," in *Proceedings of the IEEE Workshop on Real-Time Embedded Systems*, 2001, pp. 1–7. [Online]. Available: https://mobile.aau.at/~welmenre/papers/2001/rr-11-2001.pdf 18

[47] W. Elmenreich and S. Pitzek, "Smart Transducers - Principles, Communications, and Configuration," in *Proceedings of the 7th IEEE International Conference on Intelligent Engineering Systems*, 2003, pp. 510–515. [Online]. Available: https://mobile.aau.at/~welmenre/papers/2003/rr-10-2003.pdf 18

[48] J. J. P. Tsai, K.-Y. Fang, and H.-Y. Chen, "A Noninvasive Architecture to Monitor Real-Time Distributed Systems," *Computer*, vol. 23, no. 3, pp. 11–23, Mar. 1990. 18

[49] W. Elmenreich, "Time-Triggered Smart Transducer Networks," *IEEE Transactions on Industrial Informatics*, vol. 2, no. 3, pp. 192–199, Aug. 2006. 18

[50] E. Armengaud, A. Steininger, and M. Horauer, "Towards a Systematic Test for Embedded Automotive Communication Systems," *IEEE Transactions on Industrial Informatics*, vol. 4, no. 3, pp. 146–155, Aug. 2008. 18

[51] W. Elmenreich, "Configuration and Management of Networked Embedded Devices," in *Networked Embedded Systems.* Boca Raton, FL 33431, USA: CRC Press, 2009, pp. 21–22. 18

[52] E. Schlunder. (2010) High-Speed Serial Bootloader for PIC16 and PIC18 Devices. AN1310. Accessed 2015-02-08. Microchip. [Online]. Available: http://ww1.microchip.com/downloads/en/appnotes/01310a.pdf 19, 28

[53] J. Garcia-Zubia, I. Angulo, U. Hernandez, M. Castro, E. Sancristobal, P. Orduña, J. Irurzun, and J. de Garibay, "Easily Integrable platform for the deployment of a Remote Laboratory for microcontrollers," in *Proceedings of the IEEE EDUCON 2010 Conference.* IEEE, Apr. 2010, pp. 327–334. 19

[54] eLua. Accessed 2015-02. [Online]. Available: http://www.eluaproject.net 19

[55] p14p - python-on-a-chip. Accessed 2015-02-08. [Online]. Available: https://code.google.com/p/python-on-a-chip 19

[56] T. W. Barr, "Microcontroller Programming for the Modern World," Ph.D. dissertation, Rice University, 2014. 19, 20

[57] D. George. (2015, Feb.) MicroPython. Accessed 2015-02-08. [Online]. Available: http://micropython.org/ 19, 20

[58] G. van Rossum, *Python Reference Manual*, May 1995. [Online]. Available: http://www.python.org 19

[59] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho, "The Implementation of Lua 5.0," *Journal of Universal Computer Science*, vol. 11, no. 7, pp. 1159–1176, 2005. 20

[60] R. Ierusalimschy, "Integers in Lua 5.3," in *Lua Workshop 2014*, Moscow, Russia, Sep. 2014. 20

[61] ——, "Programming with Multiple Paradigms in Lua," in *Functional and Constraint Logic Programming*, ser. Lecture Notes in Computer Science, S. Escobar, Ed. Springer Berlin Heidelberg, 2010, vol. 5979, pp. 1–12. 20

[62] ——, "Small is Beautiful: the Design of Lua," in *PPL Seminar*, Stanford, CA, Mar. 2012. 20, 21

[63] *Standard Digital Interface for Programmable Instrumentation - Part 2: Codes, Formats, Protocols and Common Commands (Adoption of (IEEE Std 488.2-1992)*, IEEE Std. 22, 67

[64] National Instruments, "Actor Framework," Nov. 2012. [Online]. Available: http://ni.com/actorframework 22

[65] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. 22

[66] LuaVIEW. Accessed 2015-11-16. [Online]. Available: http://luaview.tm-solutions.eu 23

[67] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software: Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, 2000. 23

[68] A. Møller and M. Schwartzbach, *An Introduction to XML And Web Technologies.* Pearson Education, 2006. 24, 25, 26

[69] *The JSON Data Interchange Format*, ECMA International Std. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf 26

[70] F. Galiegue and K. Zyp, "JSON Schema: core definitions and terminology," Working Draft, IETF Secretariat, Internet-Draft draft-zyp-json-schema-04, Jan. 2013. 27

[71] I. Yabanova, S. Taskin, H. Ekiz, and H. Cimen, "Bootloader design application for embedded systems by using controller area network," *Global Journal on Technology*, vol. 1, 2012. 28

[72] B. Steinwender, M. Glavanovics, and W. Elmenreich, "Executable Test Definition for a State Machine Driven Embedded Test Controller Module," in *Proceedings of the 13th IEEE International Conference on Industrial Informatics*, 2015. 33, 64

[73] S. Bauer, "Log-File Management with Database Structures Using Object Oriented Programming in LabVIEW," Alpen-Adria-Universität Klagenfurt, Tech. Rep., Feb. 2016. 35

[74] P. Kruczkowski, "Controlling Shared Resources in Actor Oriented Systems," in *European CLA Summit*, 2016. 36

[75] H. Kopetz, "The three interfaces of a smart transducer," *Proceedings of FeT '2001-4th IFAC International*, vol. 2, 2001. 38

[76] W. Elmenreich, W. Haidinger, P. Peti, and L. Schneider, "New Node Integration in TTP / A Networks," Vienna University of Technology, Tech. Rep., 2001. [Online]. Available: https://mobile.aau.at/~welmenre/papers/2002/rr-05-2001.pdf 39

[77] N. Modadugu and E. Rescorla, "The Design and Implementation of Datagram TLS," in *Proceedings of the Network and Distributed System Security Symposium*, 2004. 42

[78] C. Stuart and K. Marc, "Special-Use Domain Names," Internet Requests for Comments, Internet Engineering Task Force, RFC 6761, Feb. 2013. [Online]. Available: http://tools.ietf.org/html/rfc6761 42

[79] A. Dunkels, "Design and Implementation of the LWIP TCP/IP Stack," Swedish Institute of Computer Science, Tech. Rep., 2001. 43, 125

[80] *Internet Protocol*, Request for Comments, Accessed 2015-11-04., University of Southern California Std., Sep. 1981. [Online]. Available: https://tools.ietf.org/html/rfc791 43

[81] M. Nelhiebel, M. Glavanovics, B. Steinwender, R. Sleik, G. Glatte, and G. Palatin, "Active Cycling Test of Smart Power Devices," in *ECPE Workshop on Intelligent Reliability Testing*, Dec. 2014. 48

[82] R. Sleik, M. Glavanovics, S. Einspieler, A. Muetze, and K. Krischan, "Modular Test System Architecture for Device, Circuit and System Level Reliability Testing," in *Proceedings of the 31st annual IEEE Applied Power Electronics Conference and Exposition*. IEEE, Apr. 2016, pp. 759 – 765. 51, 53, 60, 100

[83] R. Ierusalimschy, *Programming in Lua*, 3rd ed., R. Ierusalimschy, Ed. Ierusalimschy, Roberto, 2013. 52, 57

[84] S. Pitzek and W. Elmenreich, "Configuration and management of a real-time smart transducer network," in *Proceedings of the 2003 IEEE Conference on Emerging Technologies and Factory Automation*, vol. 1. IEEE, 2003, pp. 407–414. 59

[85] A. E. Council, "AEC-Q100-012: Short Circuit Reliability Characterization of Smart Power Devices for 12 V Systems," Component Technical Committee, Tech. Rep., 2006. [Online]. Available: http://www.aecouncil.com/AECDocuments.html 61

[86] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, Jun. 1987. 72

[87] J. Van Gurp and J. Bosch, "On the Implementation of Finite State Machines," in *in Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications, IASTED/Acta*, 1999, pp. 172–178. 72

[88] D. E. Knuth, *Literate Programming*, 1st ed., D. E. Knuth, Ed. The Center for the Study of Language and Information Publications, 1992. 82

[89] S. Bauer, Y. Nikitin, and B. Steinwender, “Comparing the sbRIO-9651 to the XMC4500 for a Real-Time Environment in HTOL Testing Systems,” in *NIDays 2016*. National Instruments, Mar. 2016. 90

[90] D. Egarter, A. Monacchi, M. Pöchacker, K. Schweiger, and B. Steinwender, “Smart Grid: Vision & Herausforderungen,” in *Energie – Interdisziplinäre Perspektiven auf eine knappe Ressource*, ser. Klagenfurter Interdisziplinäres Kolleg, G. Getzinger and H. P. Groß, Eds. Profil Verlag, 2014, no. 4. 95

Glossary

DAVE Digital Application Virtual Engineer – An Eclipse-based integrated development environment for configuring, compiling and flashing software for XMC micro-controllers. See `http://dave.infineon.com`

Git A version control system created by Linus Torvalds to maintain the Linux kernel development. See `http://git-scm.org`

LabVIEW LabVIEW (Laboratory Virtual Instrument Engineering Workbench) is a development environment used for instrument control and data acquisition using a graphical programming language. See `http://ni.com/labview`

lwIP A lightweight IP stack for embedded systems [79]. See `http://lwip.wikia.com/wiki/LwIP_Wiki`

National Instruments A United States company producing test and measurement equipment and developing the LabVIEW software platform. See `http://ni.com`

Acronyms

ACUTE Active Cycle Universal Test Equipment

ADC Analog-to-Digital Conversion

API Application Programming Interface

ARCTIS Advanced Repetitive Clamping Test Integrated System

ARM Advanced RISC Machines

ATE Automated Test Equipment

CAN Controller Area Network

COTS Commercial off-the-shelf

CPU Central Processing Unit

CRC Cyclic Redundancy Check

CSMA Carrier Sense Multiple Access

CSMA/CD Carrier Sense Multiple Access / Collision Detection

DAQ Data AcQuisition

DHCP Dynamic Host Configuration Protocol

DMA Direct Memory Access

DoS Denial-of-Service

DRAM Dynamic RAM

DSD Delta-Sigma Demodulator

DTD Document Type Declaration

DTLS Datagram Transport Layer Security

DUT Device Under Test

EDS Electronic Data Sheet

EtherCAT Ethernet for Control Automation Technology

FMS Fieldbus Message Specification

FPGA Field Programmable Gate Array

FSM Finite State Machine

GPIB General Purpose Input Bus

GPIO General Purpose Input Output

GUI Graphical User Interface

HTOL High Temperature Operating Life

ID Identification

IO Input & Output

IP Internet Protocol

ISR Interrupt Service Routine

JSON JavaScript Object Notation

KAI Kompetenzzentrum Automobil- und Industrie-Elektronik

LAN Local Area Network

LED Light Emitting Diode

LXI	LAN eXtensions for Instrumentation
MAC	Medium Access Control
MoPS	Modular Power Stress
NIC	Network Interface Card
OSI	Open Systems Interconnect
PCB	Printed Circuit Board
PDO	Process Data Object
PDU	Protocol Data Unit
PFC	Power Factor Correction
PI	Proportional-Integral
PLC	Programmable Logic Controller
PoL	Point-of-Load
PROFIBUS	PROcess FIeld BUS
PROFIBUS DP	PROFIBUS Decentralised Peripherals
PROFIBUS PA	PROFIBUS Process Automation
PSU	Power Supply Unit
PWM	Pulse Width Modulation
PXI	PCI eXtensions for Instrumentation
RAM	Random-Access Memory
RPC	Remote Procedure Call

SAM	Software Architecture for MoPS
SDO	Service Data Object
SPI	Serial Peripheral Interface
SPS	Smart Power Switches
TCP	Transmission Control Protocol
TP-Builder	Test Plan Builder
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VI	Virtual Instrument
VISA	Virtual Instrument Software Architecture
VM	Virtual Machine
XMC	cross-market Microcontroller
XML	eXtensible Markup Language
XVP	eXecutable Verification Plan

RRR031

TB61H5G

business contact

private contact

Git version: 460ca75. Last change: Wed, 8 Jun 2016 18:31:48 +0200.