Embedded Software Testing in a Power Electronics Context

Janne Paalijärvi

School of Science

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

Espoo 2017-11-27

Supervisor

Prof. Stavros Tripakis

Advisor

M.Sc. Pasi Lauronen

Aalto University
School of Science

Aalto University
School of Science

Aalto University, PO BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Janne Paalijärvi

Title Embedded Software Testing in a Power Electronics Context

Degree program Master's Programme in Computer, Communication and Information Sciences

Major Computer Science **Code of major** SCI3042

Supervisor Prof. Stavros Tripakis

Advisor M.Sc. Pasi Lauronen

Date 2017-11-27 **Number of pages** 68+2 **Language** English

Abstract

Usage of microcontrollers and embedded software is a continually rising trend. The use of established processor technologies in the power electronics industry combined with the customer- and vendor-based analysis demands create a need for running more embedded software on products. At the same time, the code complexity warrants better testing and verification of software.

This thesis researches a solution to enable embedded software testing in a power electronics context. Regular microcontroller-based solutions are emphasized. Care is taken to consider the reuse and automatization of testing, as well as to facilitate the early development of an electronics product.

Keywords Embedded software, power electronics, testing, verification

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Janne Paalijärvi

Työn nimi Sulautettujen ohjelmistojen testaus tehoelektroniikkaympäristössä

Koulutusohjelma Master's Programme in Computer, Communication and Information Sciences

Pääaine Computer Science **Pääaineen koodi** SCI3042

Työn valvoja Prof. Stavros Tripakis

Työn ohjaaja DI Pasi Lauronen

Päivämäärä 2017-11-27 **Sivumäärä** 68+2 **Kieli** Englanti

Tiivistelmä

Sulautettua ohjelmakoodia ajavien mikrokontrollerien suosio kasvaa jatkuvasti. Vakiintuneiden prosessoriteknologioiden käyttö vastaamaan asiakkaiden ja valmistajien vaatimuksiin mm. data-analyysissä merkitsee sulautetun koodin määrän kasvamista tuotteissa tulevaisuudessa. Tuotteita tulee myös enemmän ja ne ovat entistä kompleksisempia. Vaatimukset tuotteiden koodin parempaan laadunvarmistukseen ja testaukseen kasvavat myös.

Tässä diplomityössä tutkitaan ratkaisuja sulautetun ohjelmakoodin testaukseen mikrokontrolleripohjaisissa tehoelektroniikkaympäristöissä. Työssä on kiinnitetty huomiota testauksen uudelleenkäyttömahdollisuuksiin ja testauksen automatisointiin sekä elektroniikkatuotteiden varhaiskehitykseen.

Avainsanat Sulautetut ohjelmistot, tehoelektroniikka, testaus, ohjelmistotestaus, laadunvarmistus

Preface

I would like to thank my family and friends for their continued support and trust vested in me throught the thesis work. I would also like to specifically thank Lauri Kääriäinen for his continued support of my academic research. My thanks go to Lasse Kärkkäinen for teaching me the fundamentals of efficient software development and how to unlock the full potential of the human mind. I would like to finally thank Sampo Syreeni for his brutally honest and accurate criticism of the earlier versions of the thesis. Sampo's comments helped me the most in finalizing the thesis.

Mexico City, November 25, 2017

Janne Paalijärvi

Contents

Abbreviations and Acronyms

AC	Alternating current; an electrical current which alternates its direction periodically
AD	Analog-to-digital; the conversion of a signal from analog form to digital form
ADC	Analog-to-digital converter; a microprocessor peripheral device that converts an analog signal to digital format for the processor
ARM	Advanced RISC Machine; a processor family
ASIC	Application-specific integrated circuit; a method of implementing the application directly on silicon
C	A popular low-level program language
CI	Continuous integration; a method of development and testing for automatic integration of individual works to the common codebase
CRM	Customer relationship management; software and practices of managing relationships and business cases with customers
DAC	Digital-to-analog converter; a microprocessor peripheral device which converts a digital signal from the processor to voltage signal needed outside the microprocessor
DC	Direct current; an electrical current with non-alternating direction
DSP	Digital signal processor; a microprocessor with added accelerators for some dedicated functionalities
DUT	Device under test; shorthand for the device or product currently undergoing tests
ERP	Enterprise resource planning; a business suite of software for managing company resources and actions
FIFO	First in, first out; a method for processing data in a way that the first segment is processed and delivered first
FPGA	Field-programmable gate array; an integrated circuit with the ability to do logic-gate level programming after manufacturing
FPU	Floating-point unit: a dedicated block inside a processor to effectively operate with floating-point numbers in calculations
GPIO	General-purpose input/output; a digital microcontroller pin that can be configured to operate as input or output
HIL	Hardware-in-the-Loop; a methodology in embedded systems development where hardware-based control is run in an environment which is simulated to some degree
HTTP	Hypertext transfer protocol; a protocol to transfer and display information for web browsers

I2C	Inter-Integrated Circuit; a communications protocol for embedded system bus
IDE	Integrated development environment; an assisted system for writing programming code for applications
JTAG	A debugging interface for microcontrollers developed by the Joint Test Action Group
LED	Light-emitting diode; a semiconductor light source component
MAC	Multiplier–accumulator; a block on processor conducting fast multiply–accumulate operations
MCR	Match Control Register; a register that controls timer behavior in when a match event occurs
MCU	Micro-controller unit; a small computer on a single chip, including at least one processor, memory, and different kinds of peripherals
MISRA	Motor Industry Software Reliability Association; an organization that promotes best practices and safety in software development for cars
MR0	Match Register 0; a register holding a value that is the comparison point in timer operations in an MCU
PCB	Printed circuit board; in electronic products the board housing processor(s), components and interconnecting electrical pathways
PHP	Recursively PHP: Hypertext Preprocessor; a scripting language initially intended for server-side Web programming
PID	Proportional-integral-derivative; a generic controller type
PMBus	Power Management Bus; a SMBus-based communication bus for power electronic devices management and measurement
PNP	A bipolar junction transistor variant with a segment of N-type semiconductor between P-type semiconductor segments
PWM	Pulse-width modulator; a popular way to control actuators and other devices wired externally to microcontroller legs
RAM	Random-access memory; a memory type in computing
REST	Representational state transfer; an architecture model for web services
RJ45	A telecommunications cable jack
SNMP	Simple network management protocol; a protocol for collecting information from managed devices on IP networks
SPI	Serial Peripheral Interface bus; a synchronous serial communication bus device
TAP	Test Access Port; a functionality implemented in microcontrollers to allow debugging and testing the software on the target

UART	Universal Asynchronous Receiver/Transmitter; a device used for serial communication in an asynchronous manner
UI	User interface
USART	Universal Synchronous/Asynchronous Receiver/Transmitter; a device used for serial communication with the ability for mode selection between synchronous and asynchronous
USB	Universal Serial Bus; a standard for cables, communication and protocols for USB devices
WebUI	Web user interface; a user interface operable with a web browser

Units

s	second
ms	millisecond; 0.001 x second
us	microsecond; 0.000001 x second
Hz	hertz
kHz	kilohertz; 1000 x hertz
V	Volt
mV	millivolt; 0.001 x Volt

1 Introduction

This thesis researches solutions for the testing and verification of embedded systems software. Specifically, the researched solutions apply to power electronics products.

In commercial projects in the aforementioned industries, it is usually desirable to develop a feasible product while meeting an agreed upon deadline. The combination of embedded software and power electronics has some unique constraints regarding application development. Some of these constraints are not visible in normal application software development for example in established PC-style computers and operating systems or smartphones. With power electronics projects the actual final hardware prototype may not be initially available. Instead, the hardware is often available only at a reduced maturity level in the early stages of the project. Usually, the hardware is first completely non-existent and then reaches its final form and functionality at the very end of the actual product development cycle [6].

It is possible to gain momentum in software development for embedded systems development even before having access to the complete working hardware. The development process can usually be bootstrapped by using a subset of the final hardware. Typically the software developer can use the product-specific microcontroller(s) mounted to a debug board. This prototyping board is then connected with signal generators, switches, oscilloscopes and other instruments to emulate the environment of the final and complete electronics product. Figure 1 shows a typical setup for early-stage software development for embedded systems.

Generally, it is favorable to find bugs and defects early in a software project, as every new chronological step makes it more costly to fix the issues compared to the previous step. It should be noted, however, that the estimations of defect costs have varied depending on the source and the time in software engineering history [2][1]. Zhivich et al. [36] have argued that it is actually quite hard to evaluate the costs of discovering and fixing bugs in the operation/end user phase because a single error in an unfortunate place can have catastrophic fiscal consequences.

1.1 Objective of the Thesis

The objective of the thesis is to create a solution for facilitating embedded software testing and development. The emphasis is put on the testing and verification of regular microcontroller products in a power electronics context, but the methodologies can also be used outside of this scope. The presented solution is aimed at cutting down on the overall costs associated with the development and verification of the power electronics products running embedded software.

1.2 Overview of the Thesis

Section 2 of this thesis describes what power electronics and embedded software are. The use of embedded software in power electronics is also critically scrutinized and the future of the industry is assessed.

[1] Criticism has also been presented about the cost per defect as a metric [11].

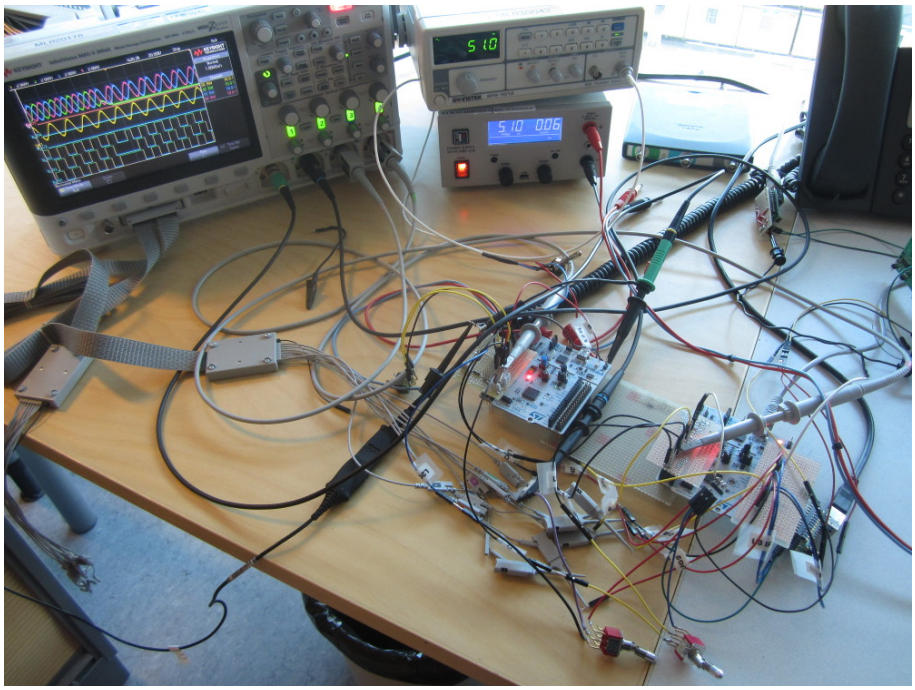

Figure 1: Embedded systems development setup for the early stages of a project. Prototyping board has been populated with the chosen microcontroller, and a plethora of output and input signals have been wired. These wires are connected to the oscilloscope, switches, power supplies, etc.

Section 3 addresses embedded software testing. Goals, constraints, and processes of the discipline are presented and analyzed. The testing methods are showcased and investigated.

In Section 4, a conceptual solution for testing embedded software in a power electronics context is presented. The solution takes into account the constraints introduced in Section 3 and emphasizes on reusability, configurability, and traceability in testing.

Section 5 shows how the system was implemented. The hardware and software implementation are discussed. Notable features and properties of the system are showcased.

The challenges faced during the implementation phase of the project are listed in Section 6.

An actual embedded systems power electronics prototype was tested as a case study. The results are listed in Section 7.

Section 8 concludes the thesis. This section evaluates the outcomes related to the goals set in Section 4. Enhancements for further development are also presented.

2 Power Electronics and Embedded Software

This section describes what power electronics and embedded software are and how they can be utilized in combination when creating new products and applications. Benefits and criticism of this cooperation are also discussed. Finally, predictions for the future for power electronics with embedded software is given.

2.1 Power Electronics

Power electronics are defined on a high level as devices that control electric energy flow in a way that makes it the most suitable for user loads [19]. The type of energy source for this electric conversion varies. Basically, the source can be of two different types of current, namely alternating current (AC) or direct current (DC). Alternating current typically has either one or three phases. The specific source of energy can also vary. It can originate for example from power distribution networks (domestic or industrial), local generator, batteries, solar panels or fuel cells [4]. Figure 2 shows a typical theoretical block diagram for a power electronics device on the left and a corresponding physical product on the right.

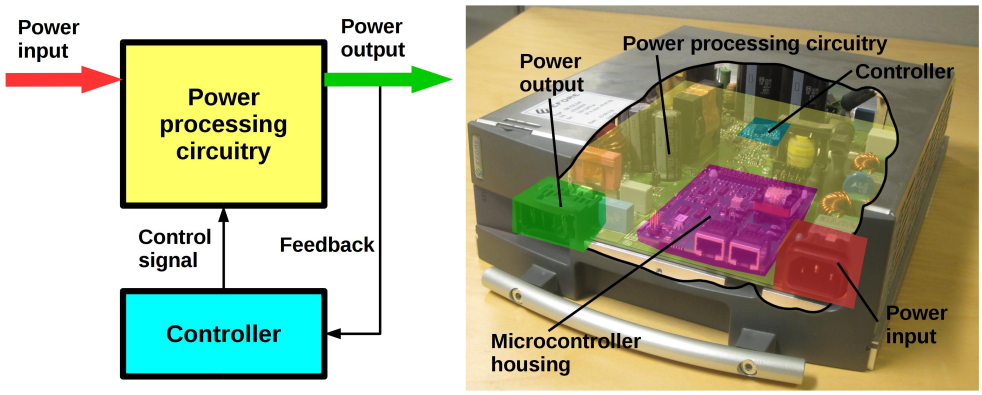

Figure 2: On the left is a theoretical block diagram of a power electronics device. On the right, there is a cutaway model of a physical power converter, with corresponding color highlights for input, output, power processing circuitry, microcontroller housing and controller chip (picture adapted from Erickson et al. [8]).

2.1.1 Control and Regulation

Regulation is an essential part of working power electronics devices [8]. Regulation typically means controlling the output characteristics of the power conversion according to the output needs[2]. These needs are dictated by the properties of the

[2]A definition for purely voltage-based regulation also exists. The definition presented here is more generic and applicable to all power conversion products.

equipment connected to the device output. The characteristics can be, for example, voltage, current, and phase-angle relationships [19]. For regulation and efficiency purposes, a dedicated controller chip is usually needed for power conversion products. For simplicity, we will examine next an elementary voltage controller (even though the same principles can be applied to other characteristics of power conversion).

The mode of operation for an elementary voltage controller is quite simple. In the controller, there is a reference voltage measurement, which is from a stable source. This stability is presumed over time and different operating conditions. For purely analog designs, the reference can be from a dedicated reference voltage circuit. For digital-based designs, this reference is a predetermined or adjusted analog-to-digital (AD) reading originating from microcontroller memory and fed to the dedicated controller chip. In both of these implementations, the controller continuously measures the voltage of the power processing circuitry output and compares it to the reference point. If the measurement is lower than the reference, the controller increases the voltage at the device output. If the measurement is higher than the reference, the output voltage is decreased [19]. If the reference and measurements match, no action is taken. It should be noted that the controller does not directly increase or decrease the voltage. Instead, it passes the new control signal to actual power processing circuitry of the device, which in turn changes the voltage. Conceptual picture of a simple controller can be seen in Figure 3.

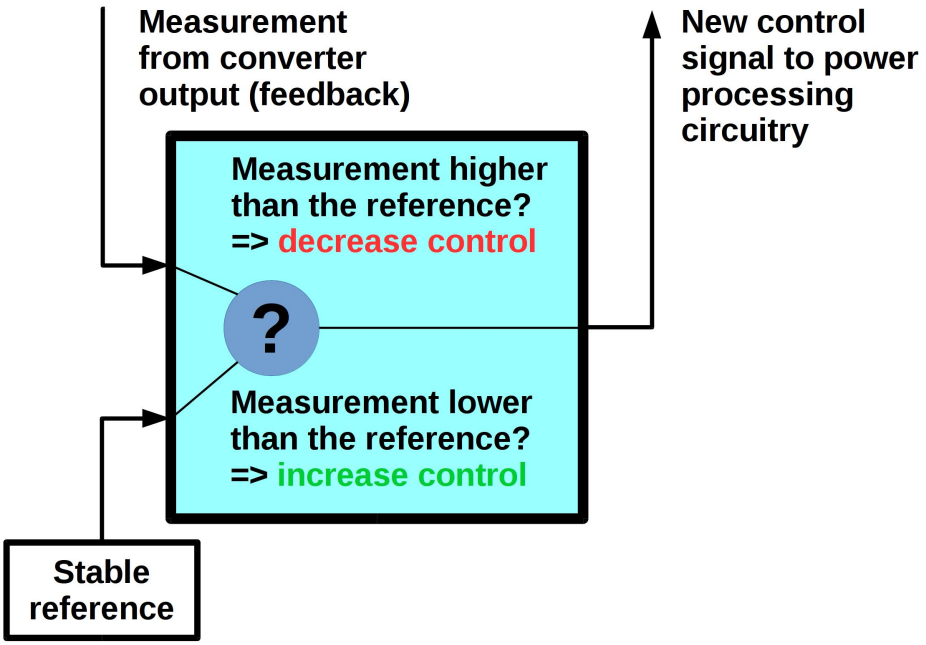

Figure 3: Simple controller for a power converter. It should be noted that controller outputs control value and does not solely alter the actual power conversion. Instead, power conversion circuitry uses the control value to make adjustments.

2.1.2 Efficiency

In power electronics it is usually favorable to have as much of the input power converted to output power as possible due to energy cost reasons. A property called "efficiency" and denoted by η is used to define the amount of conversion. In mathematical context, efficiency ranges from 0.00 to 1.00. An ideal converter transforms 100% of its input power to output power. In actuality, a 100% conversion rate is never achieved. The power not successfully converted to output is lost as heat. If the amount of heat is substantially high, a dedicated system for heat transfer must be incorporated in order to prevent the overheating of the converter. Putting it all together, maximizing the conversion efficiency benefits the product in 2 ways: 1. it reduces the cost of input energy by eliminating the waste energy, and 2. it reduces product costs due to the unnecessity of dedicated heat transfer infrastructure [19].

2.1.3 Linear versus Switching Mode

For power processing devices, in general, there are two prominent modes of operation available. These modes are linear and switching mode. Linear mode devices adjust the output voltage by using a transistor with a `constant control signal`. The total available voltage is split among transistor and device voltage. Essentially the transistor acts as an adjustable resistor, as the component conductivity state is defined by the control value from the dedicated controller [19].

For switched mode power processing devices the conversion happens essentially by switching the internal power processing circuitry on and off. More precisely, the controller gets feedback value from device output and based on a comparison with a reference, the control signal is set on or off. This control signal is fed to the power processing circuitry. A challenge of this approach is that the converted voltage waveform is essentially a square wave. Capacitive and inductive filtering is used to smooth the resulting waveform.

Linear mode and switched mode power processing technologies have different characteristics and application areas. As stated in the previous section, heat is one issue to be taken into account when constructing a power processing device. It has been established [25] that usage of switching mode technology yields much higher efficiency (and thus less waste energy as heat) than using linear mode. The downsides of the switching mode are increased design complexity and fluctuations in output voltage. Linear mode, on the other hand, offers fewer fluctuations and decreased complexity with the cost of increased amount waste energy and increased size of the device.

2.1.4 Power Electronics Device Types

Depending on the type of input and output voltages of power electronics devices, the device has a dedicated type. The types are described in the following as (partially) per [8]:

- AC-to-AC: Frequency converter, cycloconverter

- AC-to-DC: Rectifier
- DC-to-AC: Inverter
- DC-to-DC: Converter[3]

2.2 Embedded Software

Embedded software has no definition that all researchers completely agree on [29]. One popular view, however, seems to be that embedded software is run on resource-constrained devices and applications that directly interact with the real world [15]. Some authors talk about cyber-physical systems [16]. Some examples of devices running embedded software are dishwashers, refrigerators and printers. Embedded software can also be used to operate car braking systems, factory robots and keycard-based access control systems, just to name a few. A modern mobile phone can be considered as a collection of devices that run embedded software and interact with the physical world.

2.2.1 Microcontrollers

Embedded software naturally needs a device to run the program code. One type of these devices is called microcontrollers, or MCUs. These microcontrollers typically house an actual processing core, special memory for storing the program code and RAM memory for application purposes. Some microcontrollers also house non-volatile memory for application purposes or utilized memory chips connected externally.

Microcontrollers also usually incorporate peripheral devices. A peripheral device is a device that implements a defined functionality inside the microprocessor. The device may be accessible through the pins of the microprocessor in question and/or via processor internal hardware registers. Common peripheral devices found in microcontrollers are analog-to-digital converters (ADCs), digital-to-analog converters (DACs), communication transceivers (for example U(S)ART[4] or I2C[5]), timers, pulse-width modulators (PWMs), and general purpose input/output (GPIO) pins [35]. Figure 4 displays the typical microcontroller from Texas Instruments, whereas in Figure 5 there is a schematic diagram of the same microcontroller pin description and orientation. This diagram is taken from the manufacturer-supplied datasheet, which also lists the peripheral devices available for this specific microcontroller.

[3] Converter is also a generic name for a power conversion device.
[4] Universal (Synchronous/)Asynchronous Receiver/Transmitter.
[5] Inter-Integrated Circuit.

Figure 4: Picture of Texas Instruments MSP430-family microcontroller. (Copyright: Digi-Key Corporation)

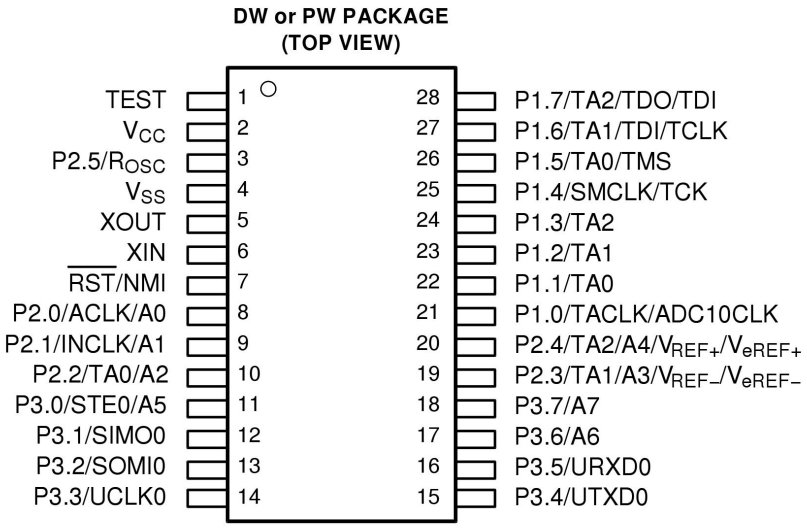

Figure 5: A diagram showing Texas Instruments MSP430-family microcontroller pin descriptions and orientation. (Copyright: Texas Instruments Inc.)

2.2.2 DSPs and FPGAs

There also exist also other kinds of devices for running embedded software than microcontrollers. One of those devices is called a digital signal processor (DSP). DSP is essentially a regular MCU, but with added functionality for accelerating some tasks, combined with instruction set supporting these accelerations [22]. One common accelerated feature is multiply-accumulate (or MAC) that is used extensively in matrix operations and filters [33]. DSP designs make it possible to carry out

these operations in only a few clock cycles. DSPs can be considered accurate and sophisticated, low-latency FIFO[6] devices. As new, top-level MCUs including DSP capabilities are entering the market, the distinction between DSPs and MCUs as processors is getting vague.

The field-programmable gate array, or FPGA, is another way of running embedded program code. In this implementation, the custom application code is synthetized on the logic-gate level. The FPGA approach allows tuning specific algorithms performance nearer the hardware-implementation level [22][7]. Different implementation techniques are tradeoffs between programming flexibility, performance, cost and power usage.

2.2.3 Development Tools

The application development differs somewhat for microcontrollers compared to the regular PC-style application development. The basic mode of operation remains as writing code, compiling it and running it on the specified microcontroller, and is often called the "target". There are some notable considerations, however. On a PC, the resources available for developing and debugging tools are usually readily available. For example, hooking up a debugger to a simple program is done completely in software. Special and often expensive hardware-based tools are needed for MCUs to achieve the same functionality. In addition, after the compilation of a simple program, it needs to be flashed to the MCU in the embedded systems context. For a PC, the program is readily available to be run.

Integrated development environments (IDEs) for microcontroller development are available. Manufacturers provide specifically preconfigured IDEs for their microcontrollers. Some manufacturer-independent IDEs like KEIL or IAR also exist. There is usually an added license cost to using these products, however. This cost naturally adds to the overall development costs for microcontroller projects. The actual compilation of the program code needs special compiler tools because the target microprocessor architecture is usually different than where the actual program development work is done. These tools are called cross-compiler tools. These tools produce application binary that can be run on the target microcontroller. For a project, it is important to have strictly specified compiler settings for all of the participating developers for the present and future so that the produced application binary will always be identical for the same set of input files.

After compilation, the application binary needs to be transferred to the target microcontroller. This is called programming the microcontroller. These programming devices are usually target-platform-specific. They also vary in speed and the amount of available slots for concurrent programming.

Features for debugging the application on target also exist. One of these solutions is the JTAG[8] interface. JTAG is probably the most widely used debugging solution

[6] First in, first out.

[7] Another technology called application-specific integrated circuit, or ASIC exists for embedded systems. ASICs are actually implementations of the full application completely on silicon.

[8] Joint Test Action Group.

for microcontrollers according to [28]. JTAG-based debuggers connect to the microcontroller test access port (TAP). Interacting through TAP via JTAG, the software developer can see the state of the application from a development computer. The developer is also able to place breakpoints, step in and out of functions, and inspect variables when combined with IDE. Another debugging method is to devise an UART-based proprietary binary protocol implemented in target application software. Another endpoint for this protocol can be the development computer. The same protocol can aldo naturally be used for test, verification, and calibration purposes.

Simulators also exist for embedded software. Simulating the program code without an actual hardware platform makes the development task easier by eliminating the need for having the target microprocessor available. Simulating eliminates the time needed to program the code to target[9]. Simulator selection is very far from being available for all microcontroller families, models, and variants. Developing and verifying such a simulator is very costly.

A proper simulator can be a great asset in developing embedded software. A simulator eliminates the need to program the application binary to the microprocessor. In addition to saving time and relaxing the need of the specified programming devices discussed earlier, the simulation approach also reduces the wear of the microprocessor as every write to the programmable memory deteriorates it. Typically, the microprocessor vendors guarantee a number of successful memory writes, after which write operations are allowed to fail. This policy applies to both the programmable memory of the microprocessor and other non-volatile type memories. Because simulators are implemented in software running on the development platform (typically a PC), directly inspecting the microprocessor and application state including register and memory values is easy. Because the simulators are software-operated, interfacing them with software-based debug and verification infrastructures is possible.

A general lack of simulators is considered one of the main reasons for researching for an alternative solution for the testing of embedded software in a power electronics context.

2.2.4 Programming Embedded Software

Now that some elementary concepts about embedded software have been declared, it is time to take a short look at how a very simple embedded systems program can be organized. In this section, we describe with C language a program for a microcontroller. The program lights up a light-emitting diode (LED) for 3 seconds when a button is pressed. Both the LED and the button are represented by GPIO pins.

[9]Sometimes, however, the simulator initialization startup time negates this effect.

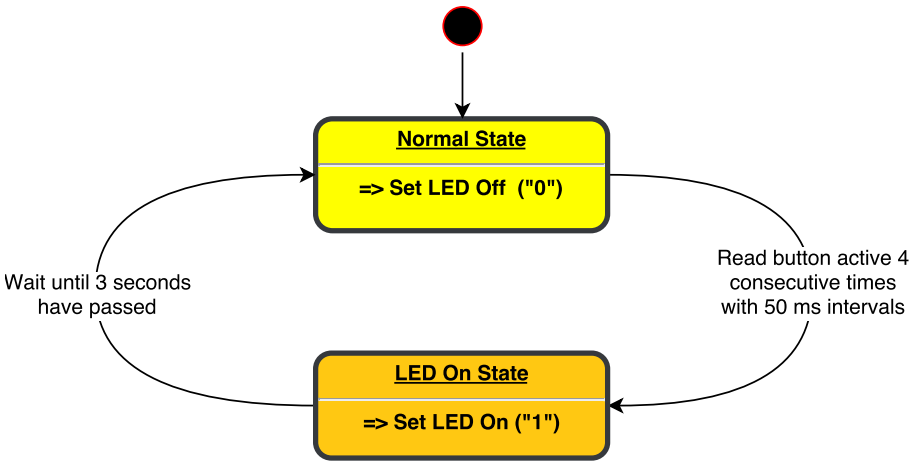

Figure 6: The LED timer program state machine diagram.

In embedded systems programming it is helpful to initially have some kind of blueprints of the desired logical operation of the system under development. In Figure 6 these blueprints are presented as a simple state machine diagram. This diagram is followed during the development of the actual program code.

```
int main() {

  // Initialize and start system
  SystemInit();
  setup_led1();
  setup_button1();
  setup_50ms_timer();

  while(1)
  {
    // Sleep the CPU until we hit the next 50ms mark:
    __WFI();
    // Run the actual 50ms task:
    run_50ms_task();
  }

  return 0;
}
```

Listing 1: Main function of a simple embedded systems program.

In Listing 1, the `main()` function of the program is shown. Execution of the code starts from this function. In the beginning, the MCU runs initialization code provided by the microcontroller vendor. Then, it configures LED and button pins for operation. Finally, it sets up the internal timing peripheral device to operate on 50 ms time units, called "ticks" and start the main loop. In the main loop ("`while (1)`"), two things happen. First, the MCU goes into sleep mode via the `__WFI()` function. In sleep mode, the MCU core is mostly idling. It returns to full power after the timer interrupt is received. The single 50 ms task function is run after this, and then a new sleep period is started. In other words, the MCU wakes up every 50 ms tick period to run the task function. If there was an additional need to run

tasks, for example, every 500 ms, simple arithmetic could be used to determine when enough multiples of 50 ms were elapsed to trigger the 500 ms task. However, here only single task is needed, simplifying the main loop somewhat.

The aforementioned sleep/interrupt behavior was chosen to avoid the so-called "busy-waiting", where the MCU runs at full speed, continuously polling a condition. This mode of operation would waste an extremely high number of clock cycles. There are cases, however, where such behavior would be well suited for the task at hand.

```
#include "LPC17xx.h"


#define LED1_PIN_SELECTOR (LPC_PINCON->PINSEL3)
#define LED1_PIN_SELECTOR_MASK (((uint32_t)1 << 5) | ((uint32_t)1 << 4))
#define LED1_GPIO_MASK ((uint32_t)1 << 2)
#define LED1_GPIO_DIRECTOR (LPC_GPIO1->FIODIR2)
#define LED1_GPIO_SETTER (LPC_GPIO1->FIOSET2)
#define LED1_GPIO_CLEARER (LPC_GPIO1->FIOCLR2)

#define BUTTON1_PIN_SELECTOR (LPC_PINCON->PINSEL0)
#define BUTTON1_PIN_SELECTOR_MASK (((uint32_t)1 << 1) | ((uint32_t)1 << 0))
#define BUTTON1_PIN_MODE (LPC_PINCON->PINMODE0)
#define BUTTON1_GPIO_MASK ((uint32_t)1 << 0)
#define BUTTON1_GPIO_DIRECTOR (LPC_GPIO0->FIODIR0)
#define BUTTON1_GPIO_READ_DATA (LPC_GPIO0->FIOPIN)
```

Listing 2: Pin-related redefinitions in MCU program code.

In order to more easily understand the organization of further functions, let's take a look at one core practice of programming embedded systems. Listing 2 represents the beginning of the program code file with the inclusion of vendor-provided, MCU-specific system header. After that, there are various rows of `#define` statements. These statements are used here to make the readability of the code better. For example, usage of the symbol `BUTTON1_PIN_SELECTOR_MASK` is much more intuitive and focused versus the awkward bit pattern `(((uint32_t)1 « 5) | ((uint32_t)1 « 4))`. This kind of redefining of obscure register values and various bit patterns is a common practice in embedded systems programming.[10]

[10]Some definitions have been omitted here to retain brevity.

```
void setup_led1(void)
{
  // Configure LED pin functionality to GPIO:
  LED1_PIN_SELECTOR &= ~LED1_PIN_SELECTOR_MASK;
  // Configure LED GPIO direction to write:
  LED1_GPIO_DIRECTOR |= LED1_GPIO_MASK;
  // Dim the LED:
  LED1_GPIO_CLEARER |= LED1_GPIO_MASK;
}

void setup_button1(void)
{
  // Configure button pin functionality to GPIO:
  BUTTON1_PIN_SELECTOR &= ~BUTTON1_PIN_SELECTOR_MASK;
  // Configure button GPIO direction to read:
  BUTTON1_GPIO_DIRECTOR &= ~BUTTON1_GPIO_MASK;
  // Make button read logical level high by default:
  BUTTON1_PIN_MODE &= PIN_RESISTORS_MASK;
}
```

Listing 3: Setup of 2 GPIO pins for LED and button functionalities, respectively.

Listing 3 shows how the processor pins are configured to operate both as a led and a button. For LED operation, the pin function is chosen as GPIO for both. This is important, as each pin can act as many different peripheral functionalities. The same selection is needed for the button pin. The next step is different between the pins, as the LED pin is configured to be in write mode, and the button pin is configured to be in read mode. In bit-level, this is about assigning a `1` or `0` bit to a dedicated director position, respectively. Assigning a `1` without touching any other bits can be done via straightforward OR operation. `0` bit is slightly trickier to set. It is accomplished by taking the inversion of the `1` bit pattern, and then ANDing the result. As a final step for LED, the GPIO value is set to logical 0 V, dimming the light. For button functionality, the last step is to make the pin "normal" state 3.3 V via the internal "pull-up resistor". Combined with software, the active state of the pin/button can be more easily made 0.0 V, or shorted to ground.

```
void TIMER0_IRQHandler(void)
{
  // Dummy interrupt handler to wake the system from sleep
  LPC_TIM0->IR = LPC_TIM0->IR;
}

void setup_50ms_timer(void)
{
  // Clear counter and generate interrupt when timer matches:
  LPC_TIM0->MCR |= (TIMER0_MR0_INT_MASK | TIMER0_MR0_RESET_MASK);
  // With prescaler set the counting step to 1 ms:
  LPC_TIM0->PR = (SystemCoreClock /
                  (CONV_1MS_TO_HERTZ * TIMER0_CLOCK_DIVISOR)) - 1;
  // Make counter count to 50 counting steps, equaling 50 ms:
  LPC_TIM0->MR0 = TIME_50MS;
  // Connect the interrupt handler:
  NVIC_EnableIRQ(TIMER0_IRQn);
  // Finally, enable the timer:
  LPC_TIM0->TCR |= TIMER0_ENA_MASK;
}
```

Listing 4: Dummy interrupt handler and 50 ms timer configuration.

In Listing 4, the timer functionalities are defined. There is a dummy interrupt handler, whose only task is to clear the interrupt condition. This elementary handler is needed to make it possible to wake the MCU from sleep mode periodically.

An actual timer configuration function is the most complicated part of the whole program. The first instruction manipulates the `MCR`, or match control register. It makes the timer execute the interrupt handler and reset the timer value once the counter value equals the `MR0`, or match register 0 value. The `PR`, or prescaler register, controls how many clock cycles need to pass before the counter is actually incremented by one unit. Here, the adjusted `PR` value was made to correspond to 1 milliseconds. `MR0` value was set to cause a match when 50 times the 1 millisecond period has passed. Finally, in the function, the interrupt handler is connected and the timer set to run.

Table 426. TIMER/COUNTER0-3 register map

Generic Name	Description	Access	Reset Value[1]	TIMERn Register/ Name & Address
IR	Interrupt Register. The IR can be written to clear interrupts. The IR can be read to identify which of eight possible interrupt sources are pending.	R/W	0	T0IR - 0x4000 4000 T1IR - 0x4000 8000 T2IR - 0x4009 0000 T3IR - 0x4009 4000
TCR	Timer Control Register. The TCR is used to control the Timer Counter functions. The Timer Counter can be disabled or reset through the TCR.	R/W	0	T0TCR - 0x4000 4004 T1TCR - 0x4000 8004 T2TCR - 0x4009 0004 T3TCR - 0x4009 4004
TC	Timer Counter. The 32-bit TC is incremented every PR+1 cycles of PCLK. The TC is controlled through the TCR.	R/W	0	T0TC - 0x4000 4008 T1TC - 0x4000 8008 T2TC - 0x4009 0008 T3TC - 0x4009 4008
PR	Prescale Register. When the Prescale Counter (below) is equal to this value, the next clock increments the TC and clears the PC.	R/W	0	T0PR - 0x4000 400C T1PR - 0x4000 800C T2PR - 0x4009 000C T3PR - 0x4009 400C
PC	Prescale Counter. The 32-bit PC is a counter which is incremented to the value stored in PR. When the value in PR is reached, the TC is incremented and the PC is cleared. The PC is observable and controllable through the bus interface.	R/W	0	T0PC - 0x4000 4010 T1PC - 0x4000 8010 T2PC - 0x4009 0010 T3PC - 0x4009 4010
MCR	Match Control Register. The MCR is used to control if an interrupt is generated and if the TC is reset when a Match occurs.	R/W	0	T0MCR - 0x4000 4014 T1MCR - 0x4000 8014 T2MCR - 0x4009 0014 T3MCR - 0x4009 4014
MR0	Match Register 0. MR0 can be enabled through the MCR to reset	R/W	0	T0MR0 - 0x4000 4018

Figure 7: An excerpt from NXP Cortex-M3 LPC176 datasheet. (Copyright: NXP Semiconductors N.V.)

The timer-based definitions presented in the program code are, of course, well-defined. The definitions can be found in microcontroller-specific documents called datasheets. These documents are provided by MCU-vendors in order to make the development of software in the specific microcontrollers possible. Figure 7 shows an excerpt of the LPC1768 datasheet. Only a small part of the full functionality of the MCU is presented here as the full document spans almost 850 pages.

```
void run_50ms_task(void)
{
  static volatile uint8_t u8_state = STATE_NORMAL;
  static volatile uint8_t u8_button_samples = 0;
  static volatile uint8_t u8_led_wait_samples = 0;

  if(u8_state == STATE_NORMAL)
  {
    if((BUTTON1_GPIO_READ_DATA & BUTTON1_GPIO_MASK) == 0) // Button read ok
    {
      if((u8_button_samples++) >= TIME_200MS_IN_TICKS)
      {
        LED1_GPIO_SETTER |= LED1_GPIO_MASK; // Enough read samples,
        u8_button_samples = 0;              // light up the LED
        u8_state = STATE_LED_ON;            // and change state.
      }
    }
    else
    {
      u8_button_samples = 0;
    }
  } // Check pass of 3 seconds in state STATE_LED_ON:
  else if((u8_led_wait_samples++) >= TIME_3S_IN_TICKS)
  {
    LED1_GPIO_CLEARER |= LED1_GPIO_MASK; // Enough wait samples,
    u8_led_wait_samples = 0;             // dim the LED and
    u8_state = STATE_NORMAL;             // change the state.
  }
}
```

Listing 5: The 50 ms task function of the LED controlling program.

The most important part of the program code for the system is the 50 ms task function shown in Listing 5. As shown earlier in the state machine diagram, the system has 2 states, namely “Normal” and “LED On”. In the normal state, the input of the control button/pin is read. In Normal state, the read happens once every 50 ms. The button is determined to be pressed down when the corresponding pin is connected to ground. When 4 reads have occurred at 50 ms intervals, the LED is lit, button sample counter reset, and the state changed to LED On state[11].

The LED On mode counts how many 50 ms samples have passed. When there has been enough samples for 3 seconds, the LED is dimmed, the local sample counter is reset, and the state is declared again as Normal. A complete set of program codes is available on GitHub [23]. A video demonstration of the working system is available on YouTube [24].

After having introduced both concepts individually, we can then take a look at how embedded software can be used in power electronics.

[11]It should be noted that in Normal mode if an inactive button state (1) is read after an active state (0), the sample counter is also reset. The behavior is called filtering. It is undesired to trigger a functionality based on a single sample, which may even be a mishap or temporary electrical glitch. Therefore more samples are investigated.

2.3 Embedded Software in Power Electronics

Figure 2 shows a block diagram of a power electronics device on the left and an actual device (rectifier) on the right. The block defined as a microcontroller housing seats the microprocessor that in turn runs the program code. A detailed view of the housing is shown in Figure 8.

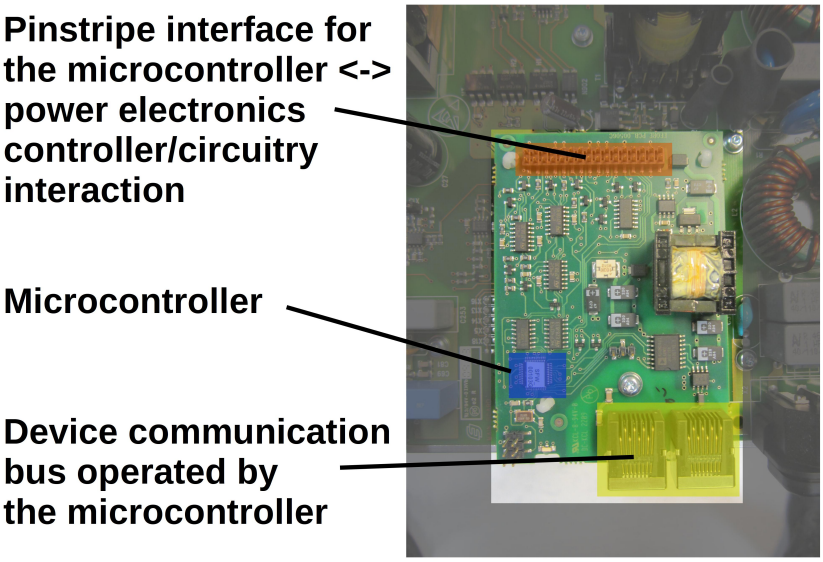

Figure 8: A picture shows the microcontroller housing of a power electronics device. On top, there is the conductive pathway that is used as an interface to interact with the power processing circuitry. The microcontroller is shown on the left and communication sockets on the bottom right.

In this design, the microcontroller housing printed circuit board (PCB) interfaces with power processing circuitry via conducting pins. These electrical pathways can be used to read state information from power processing circuitry (for example voltage and current). The microcontroller can also alter the state of the power conversion by, for example, changing the reference voltages for power processing circuitry controller or enabling or disabling the state of the device output. These alterations can be performed by using microprocessor PWM and digital output peripherals, for example. In the picture, there are also two RJ45 connectors visible. The microcontroller uses these connectors in forming a messaging bus to communicate with other products. A power electronics design can also include multiple different processors that perform dedicated tasks.

2.3.1 Benefits

Having embedded controllers in power electronic products has many benefits. As discussed in the previous section, a microcontroller can read measurements and other

information from the power conversion circuitry. The microcontroller can analyze and store this data as well as determine the error conditions in operation (either internal or when physical conditions reach alarm levels).

As hinted in the previous section, the controller can also participate in the communication bus with other devices. Multiple devices can act together in cooperation to fulfill dedicated tasks or report measurement and diagnostics data to upstream devices. Some devices allow the reconfiguration of their parameters via the communication bus (for example for setting voltage limits and configuring alarms). It is also possible for devices to collect statistics about their operation and display this to the user when requested[12]. Some power electronics communication buses can be used during production testing at the factory for calibrations and verification measurements.

Many proprietary and standardized communication protocols for power electronics exist. One of these protocols is PMBus [32][13]. Some of the other protocols used in power electronics running embedded software are SNMP[14] and even plain HTTP[15]. If communication happens only between devices, a lightweight binary protocol is preferred.

2.3.2 Future

There is active research being conducted using FPGA-based solutions for power electronics devices [14]. As the unit prices of the technology become lower, FPGA chips can be more interesting candidates for future designs.

Another trend of power electronics is the need for cloud-connected services. In these scenarios, a lightweight protocol is used to transfer device data to the cloud. This data can be accessed via multiple different user interfaces. A rudimentary web browser-based interface can be implemented, but market demand is getting focused on mobile phone applications for cloud-connected power electronics products.

The last emerging trend in power electronics running embedded software is self-diagnosing devices. Devices can determine when they have a fault condition and report this event to upstream devices or the cloud. If the amount of data sent to the cloud is sufficiently large, machine learning algorithms can be used to predict upcoming fault conditions based on variations of measurement values. In this case, automated systems like enterprise resource planning and customer relationship management (ERP and CRM, respectively) can proactively propose the ordering of spare parts. An overview of this kind of system is shown in Figure 9.

[12] Rudimentary information about the device can also be displayed for example via LED lights operated by the microcontroller.

[13] Power Management Bus.

[14] Simple network management protocol.

[15] Hypertext transfer protocol

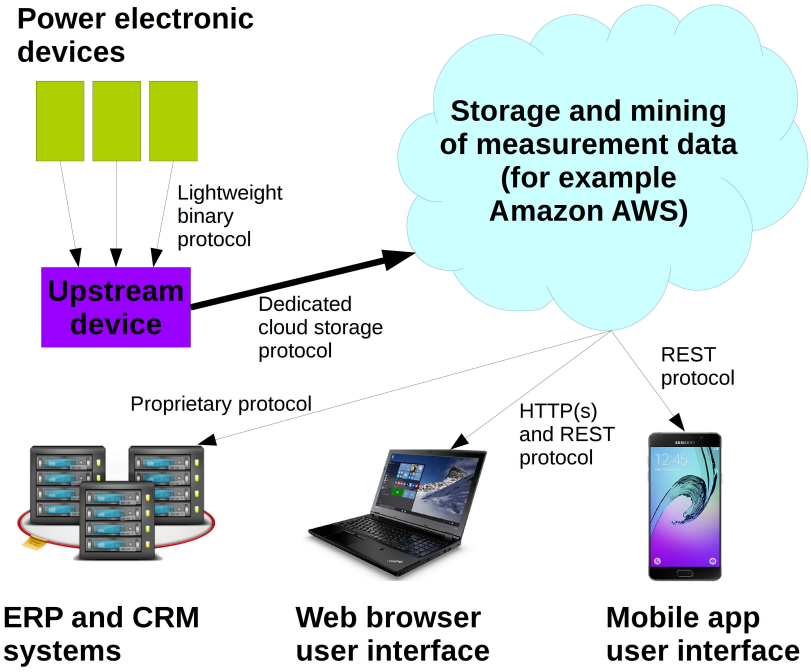

Figure 9: The image shows the measurement data flow from power electronic devices to the cloud. Using data mining, the predictions of spare part needs can be made and this data used when interacting with the customer via ERP and/or CRM. Measurement results and status information can also be displayed in a web browser user interface or dedicated smartphone applications (Sub-picture copyrights: Samsung Electronics Co. Ltd., Lenovo Group Ltd., Flickr.com/flow2u).

In this section, we discovered what power electronics and embedded software are. Different kinds of processors were also introduced, as well as some common development tools. We also saw how embedded software can be used in power electronics and took a look at the upcoming trends in the industry.

3 Embedded Software Testing

In this section, we take a look at how embedded software testing and verification is carried out. Testing goals and constraints are presented and analyzed. Different testing and verification methods are also showcased and investigated.

3.1 Benefits of Testing in Industrial Power Electronics

Most publications and papers do not explicitly state why software is tested. It is more common that publications list many ways, methods, and strategies for testing but don not directly ask or answer the question: Why? Burnstein et al. state that testing "provides strong support for the development of high-quality software" [7], which is indeed the direct goal of testing. The natural question thereafter after this point is: Why do we need high quality software? High quality software is easier to maintain, develop, and operate. The operation of this kind of software is also less error-prone. With good-quality software, there is a smaller risk of facing heavy financial and public relations consequences that may happen on some software failures as briefly discussed in Section 1[16].

Power Electronics industry differs somewhat from the "regular" software industry. In PC and mobile handset applications, it is more of a norm than an exception to have regular updates to the software. These updates are usually applied by the end users. The case for power electronics products can be quite different. For starters, such a product might not have a field-operable mechanism for installing updates. In order to cut mass producing costs, it is usually carefully investigated if implementing this kind of mechanism is justified regarding the development time and additional hardware requirements to facilitate the functionality. Another issue is that, even if such update functionality existed, there is no guarantee of a feasible delivery method, as opposed to PCs and handsets where usually some kind of Internet connection is always available.

It is usual that in mass-produced power electronics the software needs to be working on an acceptable level beginning from the first production batch after proto- and acceptance series for the full intended lifetime of the product. Otherwise, the manufacturer risks a costly recall process if a warranting error is found from the units after they have been shipped to end users in various locations of the world. In other words, the manufacturer usually needs to get the software "right" straight from the beginning.

As summarization: Testing aims to make software development easier and reduce the overall costs by preventing failures after the product launch. Testing is especially important for products that do not have field-operable mechanisms for software upgrades.

[16]Overall, healthy criticism should be applied when dealing with software defect costs. For example, the "classic" 1* design/6.5* implementation/15* testing/100* operation defect cost rule has been attributed to an institution called "IBM Systems Sciences Institute". The problem about this attribution and thereby the whole rule itself is that such an institution seems to have never existed.

3.2 What to Consider when Testing in the Power Electronics Context

Many factors contribute to embedded software testing. There are some additional ones when considering the power electronics context. From a generic point of view, the things to consider in testing fall into two types, namely constraints and resources. Some constraints are explicit and some are derived, for example the unavailability of a resource is a constraint.

Two basic project resources are time and money [18]. In an industrial power electronics project, a customer[17] usually determines how much time is available in the realization of the project. The customer also determines the scope and quality level for the project. Depending on the timing and other constraints, the amount of of funding required is also budgeted for a project. Other natural points to be taken into consideration while testing are the number and the skill set of the people involved in the project.

Some industrial power electronics projects are special in a sense that they need specialized hardware for testing. For example, if the product under development needs to operate under a variety of temperatures, a climate chamber may need to be used. Especially the communication of an embedded systems product via I2C or other similar buses may be severely affected in temperatures of -40 degrees centigrade. It is hard to figure out the complete effects of such environments and their change gradients with a simple desktop simulation. Therefore, climate chambers can be essential in test and verification work regarding embedded systems power electronics with the communication bus, especially when the projected operation temperature differs significantly from room temperature or changes over time.

The control of some power electronics devices[18] is often driven by PWM signals. For some devices from this field, the PWM is used to carry out syncing operations between MCUs and other components. Quality PWM signal generators and analyzers are essential when testing embedded software running on power electronics devices in the development phase.

Simple simulations cannot ascertain the correct functionality of the final power electronics product. They may, of course, give implications about how the software running inside these applications operates and conforms to specifications, but for the final product needs to have additional testing conducted for verification. For products designed to convert or sink large amounts of electrical energy, high-capacity power sources and loads may be used. One such device is pictured in Figure 10.

One notable constraint specific to industrial power electronics testing are standards and regulations. One such standard is the IEC 60950-1 [9] that encompasses the safety requirements of information technology equipment. Even though this specific standard is not software-related, safety considerations should also be kept in mind in software development, as implications of high-energy power electronics products

[17] A customer may be an entity external or internal to the company overtaking the project. Sometimes companies develop their own products for sale and sometimes they contractually develop products for other companies.

[18] Especially in inverter technology.

Figure 10: Programmable 40V/375A/15kW power supply. (Copyright: Chroma Systems Solutions, Inc.)

can be considerable. There are usually many standards and regulations involved in a project. If the product is sold in many market areas, more regulations may apply. The customer may also oblige the developing party to adhere to other specifications at will.

3.3 Testing and Verification Methods

Multiple approaches for the testing and verification of embedded software and software in general exist. Some of these are more general and others more specific to embedded systems. A partial list of the methods is presented here for investigation.

3.3.1 Checklists

Checklists are a way of using predetermined listed knowledge to identify the possible risks of a project [3]. In addition to identifying the risks, checklists also contain information about how to deal with the risk. A typical software project checklist is shown in Figure 11.

3.3.2 Code Reviews

Reviews are an important part of the verification of a project. Reviews can concentrate on the availability and completeness of documents defined in process instructions. The aforementioned checklists are an example of such documents, other examples being plans, backup plans, schedules, purchase orders, authorizations, etc. Processes may dictate many different reviews during different stages of the project.

The program code can also be reviewed in code reviews. In these events, the author presents his/her program code to other participants, who usually include peers. Code organization and architecture are scrutinized and the possible errors and dangers are pointed out. A follow-up review may be organized later on after giving time to programmers to incorporate changes in the code.

3.3.3 Test-driven Development

Test-driven development is a practice where program code tests are written prior to actual application code. In this practice the functionalities of the application are

TABLE 1.
TOP 10 SOFTWARE RISK ITEMS.

Risk item	Risk-management technique
Personnel shortfalls	Staffing with top talent, job matching, team building, key personnel agreements, cross training.
Unrealistic schedules and budgets	Detailed multisource cost and schedule estimation, design to cost, incremental development, software reuse, requirements scrubbing.
Developing the wrong functions and properties	Organization analysis, mission analysis, operations-concept formulation, user surveys and user participation, prototyping, early users' manuals, off-nominal performance analysis, quality-factor analysis.
Developing the wrong user interface	Prototyping, scenarios, task analysis, user participation.
Gold-plating	Requirements scrubbing, prototyping, cost-benefit analysis, designing to cost.
Continuing stream of requirements changes	High change threshold, information hiding, incremental development (deferring changes to later increments).
Shortfalls in externally furnished components	Benchmarking, inspections, reference checking, compatibility analysis.
Shortfalls in externally performed tasks	Reference checking, preaward audits, award-fee contracts, competitive design or prototyping, team-building.
Real-time performance shortfalls	Simulation, benchmarking, modeling, prototyping, instrumentation, tuning.
Straining computer-science capabilities	Technical analysis, cost-benefit analysis, prototyping, reference checking.

Figure 11: Software project checklist. (Copyright: Boehm/Software risk management: principles and practices)

identified first, and then the test code is written for that specific functionality [12]. After this, the actual application code is written and tested with the pre-written test code until the functionality works. The method is iterated for all the functionalities of the application.

In the embedded systems context, this verification method needs to take into account the hardware-dependent nature of the devices under test. The test platform may need to expose the microcontroller peripherals, pins, and other functionalities to the test framework in an emulated way. These emulated components can be used as direct input for testing or as defining the environment for testing.

3.3.4 Simulation

Simulation can be utilized in the testing and verification of some embedded software. In some cases, the program code for MCUs can also be compiled for the PC environment. The binary can then be run on a PC and the results evaluated. Another form of simulation is the use of scientific calculation tool, such as MATLAB, in the development and verification of control algorithms. One such algorithm is the proportional–integral–derivative, or PID, and its derivatives. In some cases, simple spreadsheet programs can be used to at least partially verify the correctness of an algorithm.

3.3.5 Unit Testing

According to Sen et al., unit testing means splitting the application into small units and functions that are tested individually [26]. Input for the functions can be generated and the generation can be automated. For a successful software project, unit tests are not by themselves a sufficient way to ensure the quality of the product under development [6]. Instead, other types of testing and verification are also needed.

3.3.6 Regression Testing

Regression testing is the practice of establishing confidence to already tested parts of the code after changes have been introduced [17]. It is carried out by specifying and executing regression tests. After a change in the codebase, one of the following happens: Either all of the tests are repeated or only a subset of tests are repeated. The prior is called the retest-all approach and the latter the selective retest [30].

In the embedded software context, special care should be taken when running automated regression testing. As discussed in section 2.2.3, the microcontrollers have a limited number of cycles that can be used in rewriting with new program code. If regression tests are run frequently and especially with the retest-all method in a big embedded project, there is a risk of wearing out the internal programming memory. With microcontrollers nowadays having a minimum reflash count of 10,000-100,000, this risk can be quite small, but should still be kept in mind, especially when implementing automated test systems.[19]

3.3.7 Static Code Analysis

Static code analysis is performed with special analysis tools, that check that the structure of the code conforms to standards set in the project. Such an analyzer tool can enforce the programmers to have a well-defined code organization with enough comments while avoiding confusing or dangerous structures and practices [6]. Static code analyzers can be configured to enforce agreed style guides and also actual standards, such as MISRA[20] C [20]. An example of a static code analyzer is the the Lint software [34][21].

[19] It should be emphasized that the actual maximum allowed reflash counts can be considerably higher. Once, an MCU with 100,000 guaranteed reflash-cycles was tested for wear. The test setup ran for weeks, constantly erasing and flashing the MCU. It turned out that the testing was unable to wear down the MCU during the test time. The testing was stopped when 3,000,000 reflash-cycles had been induced.

[20] Motor Industry Software Reliability Association.

[21] Lint was originally the name of UNIX command-line software that was used to spot irregularities in C code. Since the introduction of this specific tool the word “lint” has become a synonym for all linting tools. Lint-like functionalities or lint modules exist in many modern software development products.

3.3.8 Dynamic Code Analysis

Dynamic code analysis is the practice of analyzing the execution of a running program. In this practice, the inputs of the testable functional units are manipulated and the result and runtime information are recorded [1].

Dynamic code analysis tools also exist. One such tool tailored for embedded software is called VectorCAST [31]. This tool can analyze the existing codebase and generate dynamic unit tests without the user actually needing to write any code at all. Compiled test code can be run in a simulator/emulator or also on the actual hardware.

3.3.9 Hardware-in-the-Loop

Hardware-in-the-loop (HIL) is a design, implementation, and verification methodology where the developed embedded system's control part is run on actual hardware, but its environment is simulated externally to some extent due to the unavailability of the actual environment [10]. This unavailability may be due to cost or complexity issues. Bouscayrol [5] has defined different types of HIL simulations, including the "signal level HIL simulation", which is presented in Figure 12. The VectorCAST tool described in previous section also utilizes HIL principles.

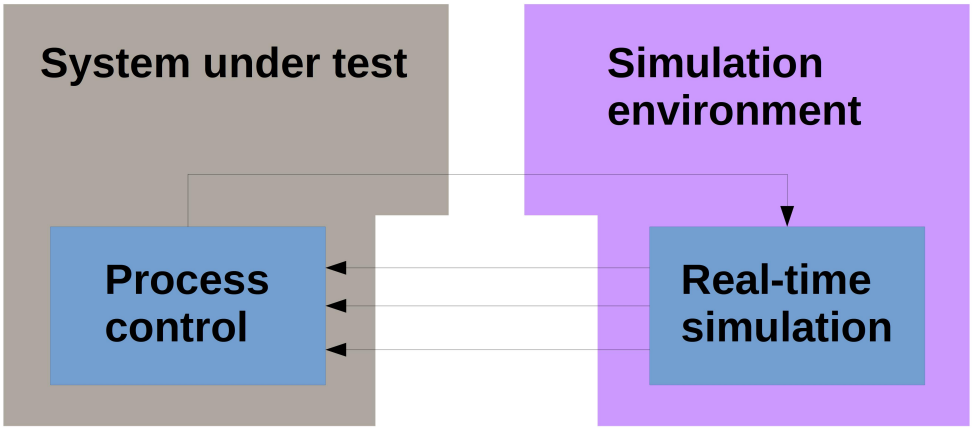

Figure 12: Schematic of signal-level Hardware-in-the-Loop simulation (Copyright: Bouscayrol/Different types of hardware-in-the-loop simulation for electric drives)

3.3.10 Black Box Testing

Black box testing means testing a system without prior knowledge of the internal structure of the system or its source code [13]. Black box testing is also called functional testing. One benefit of black box testing is the separation of the roles of programmer and tester. Due to the invisibility of the internals of the system, the test coverage is limited.

3.3.11 White Box Testing

White box testing is the practice of testing a system while having intrinsic knowledge about its internal workings, including architecture and source code [13]. White box testing enables testing the control flows, branching, and code path, among other things. An advantage of this test method is that it provides better test coverage. As a downside, the method is quite costly to implement to attain perfect or high test coverage. A test methodology combining both black box testing and white box testing is called gray box testing.

3.3.12 Fuzzing

Fuzzing is an automated test technique that means flooding inputs of the tested system with invalid, partial, and random data [21]. The methodology is cost-effective in relation to manual boundary testing. Manually devising the "correct" test points can be a resource-intensive task especially for a human, whereas automatically and randomly flooding the input and logging pass and fail data is easy and more easily automatable. A total random nature of the flooded signals is not an absolute necessity. Analysis of the program code of the tested system can be conducted, yielding efficient adjustments to the flood data.

3.3.13 Continuous Integration

In practice, continuous integration (CI) means a framework that tracks the changes in a shared source code repository and then executes predetermined actions based on these changes. A compilation step is executed for languages needing compilation. After passing compilation tests, unit tests can be run. And after unit tests, the whole application could be deployed to test environment and further tests run there. Finally, report of the full process would be automatically compiled for the developers and necessary build artifacts archived. CI is the cornerstone of agile software development [27].

In this section, we established the benefits of software testing with power electronics. We also investigated special testing considerations in the power electronics context and showcased some of the test and verification methods available.

4 Our Approach

This section describes an approach for testing embedded software in a power electronics context. The solution described here can be used to test embedded software in general also, but the emphasis is on traditional microcontrollers and power electronics.

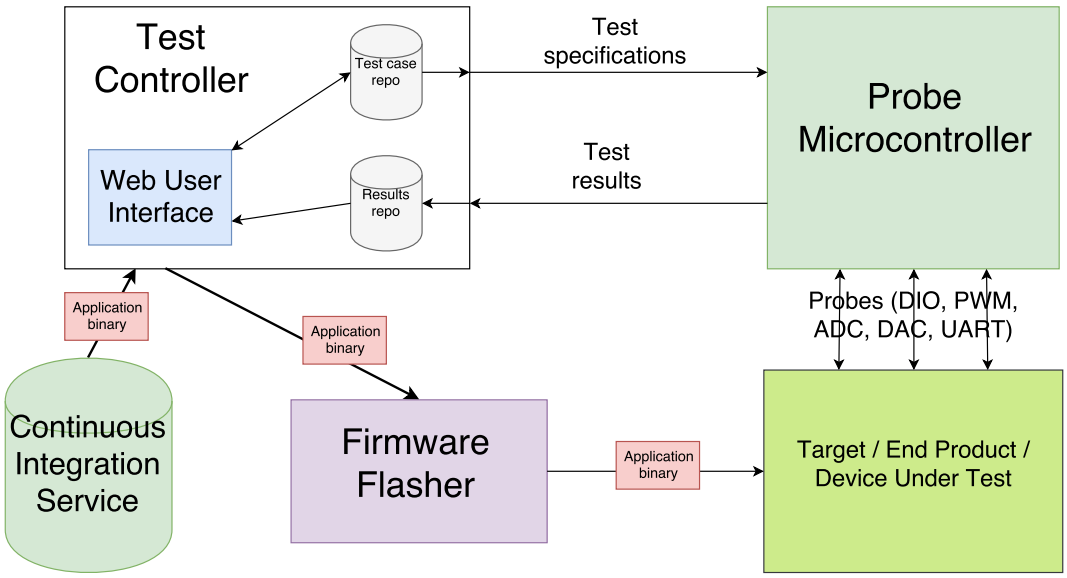

Figure 13: Overview of the devised system for testing embedded software in power electronics context.

4.1 Requirements

Primary requirement for the solution was to build a system that could be used to test embedded software running on microcontrollers in power electronic applications. The solution should be fast enough to facilitate observing the necessary signal changes, and it should also be able to generate such signals itself. It needed also be possible to use the system in both "manual mode" and as part of automated test infrastructure. Other necessities were the ability to store test specifications and results permanently and to have the system as easy to use as possible via good user interface (UI) design. Low hardware cost of the system was also one design goal.

4.2 High Level Architecture

To fulfill the requirements, a 3-part component core system was planned. It utilizes the concepts of Hardware-in-the-Loop, Continuous Integration, White Box Testing, and Black Box Testing from Section 3.3. The components of the system were called Test Controller, Firmware Flasher, and Probe Microcontroller. Interfacing with automated test infrastructure needs an actional component called Continuous Integration, or CI Service. CI is presented in order to define its relationship with

the planned core system. Figure 13 shows the overview of the plans regarding the components and their relations.

4.3 Individual Components

In the following sections, we briefly discuss each component of the devised system. It should be noted that when integrating the components and performing actual testing, great care should be taken to ensure that grounding and isolation of individual components are as expected in order to prevent electrical disturbances and erroneous test results. If the developed system were to be used to test actual high-energy power conversion products running embedded software directly, the safety and isolation considerations would be even more severe.

4.3.1 Test Controller

The Test Controller is the component that directly communicates with all of the other components of the system. It has an interface toward the CI service for receiving built application binaries as well as another interface for delivering the application binary to the Firmware Flasher for flashing to the Device Under Test (DUT). The Test Controller has a Web User Interface (WebUI) for performing manual tests or defining new test cases. A partial sketch of the WebUI is shown in Figure 14.

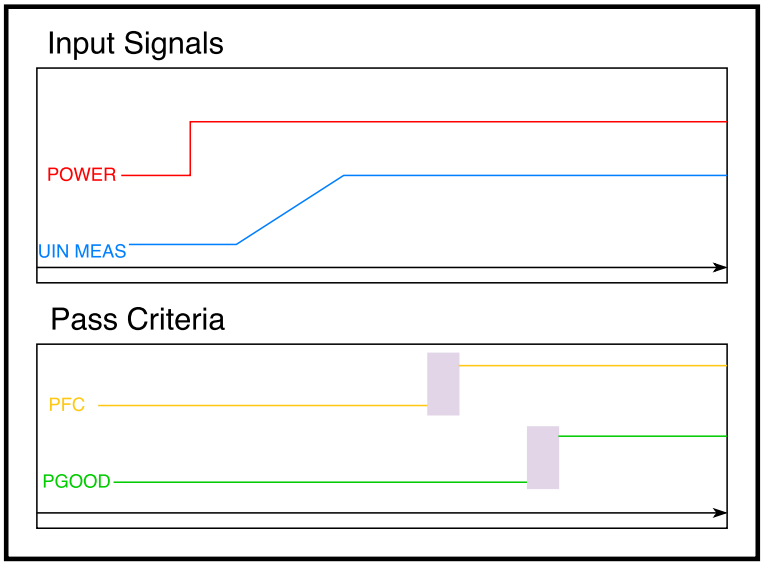

Figure 14: A partial Web User Interface of the Test Controller.

The Test Controller also has a local storage for test cases and a repository of test results. An off-device backup mechanism should be used in order not to lose test specifications or result data. The Test Controller instructs the Probe Microcontroller to test the DUT and to receive test results afterward. Communication between these two components is performed via a UART bus.

4.3.2 Firmware Flasher

The Firmware Flasher is a specific tool to flash the necessary application binary to the DUT when needed. This component should be made easily replaceable, as different DUTs require different tools for flashing. In an ideal case, one generic tool would cover all of the possible types of microcontrollers for flashing. As firmware flashing wears down the DUT MCU, a counter should be implemented in the Test Controller to act as a safeguard in testing when approaching the specified rewrite count limits of the MCU.

4.3.3 Probe Microcontroller

The Probe Microcontroller is the component that directly interfaces with the end product currently being tested. It has a dedicated protocol for receiving test specifications as scripts from the Test Controller. The results are stored locally for retrieval later on. The Probe Microcontroller includes different instruments, such as GPIOs, PWM devices for setting and analyzing signals, DACs and ADCs for setting and analyzing voltages, and communication buses for sending and capturing information (I2C, SPI[22]). The Probe Microcontroller includes a test sequencer with a specified 1 microsecond timestamping.

4.3.4 Continuous Integration Service

The Continuous Integration Service is used when the system is connected to automated test infrastructures. In the service, a new code commit to the repository codebase could trigger a build of the application binary. This binary would then be transferred to the Test Controller for relaying for flashing. After flashing, the related specified tests could be run automatically, and the results for test runs stored.

In this section, we devised a plan and architecture for building a system for testing embedded software in a power electronics context with the concepts introduced in Section 3.3. Each component of the system was defined separately, and their interconnected relationships were also defined.

[22]Serial Peripheral Interface bus.

5 Implementation

The implementation of the solution described in Section 4 is described in this section. Both the hardware and the software parts are discussed. Notable features and functionalities are also showcased.

Due to timing constraints and the amount of effort needed in implementing the complete design as a working system, it was decided that only the Probe Microcontroller part would be implemented during the project. This section reflects the decision.

5.1 Hardware Implementation

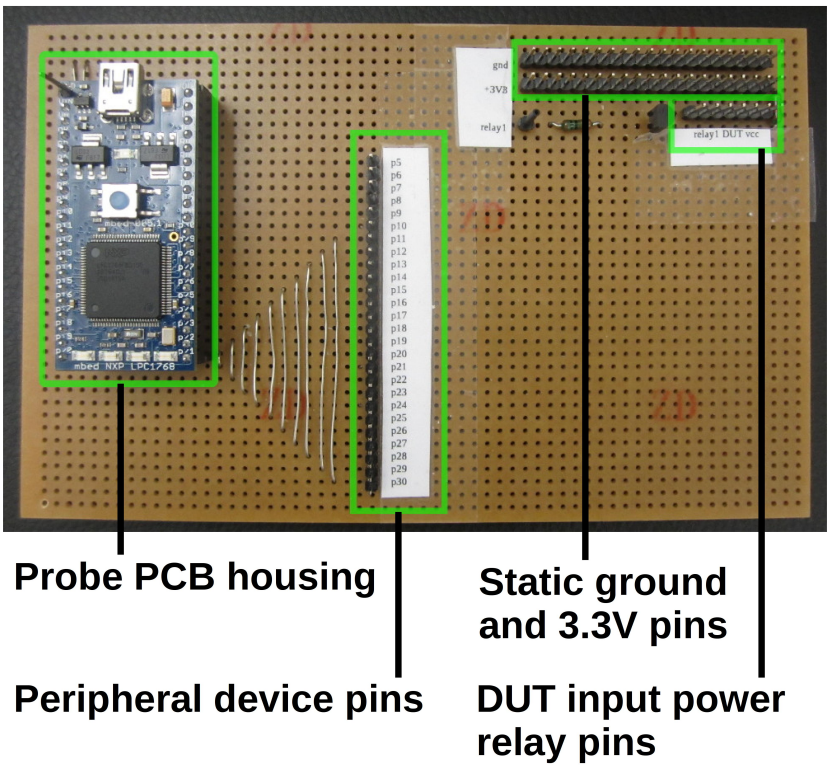

Figure 15: Probe for microcontroller testing constructed on a prototyping board.

As part of the development process, a housing for the Probe Microcontroller was constructed on prototyping stripboard. Operating software was also developed. The MCU type was selected to be ARM[23] LPC1768 with the Mbed platform PCB because of relatively cheap price and previous knowledge of the microcontroller.

[23] Advanced RISC Machine.

The complete physical construction of the implemented system is presented in Figure 15. The board has on the upper left corner a housing for the Mbed LPC1768 PCB. The PCB is pictured pressed into the socket. The approach of using a PCB/microcontroller socket in prototyping allows for soldering the necessary components to the board without the risk of damaging the processor. Furthermore, in case the PCB/microcontroller breaks down during operation, it can be replaced easily without the need for major modifications to the rest of the prototyping stripboard.

Oriented vertically in the middle there is an array of pins. These pins are routed from the LPC1768 processor. They are peripheral device pins that can be used for testing input signals sent to the device under test (DUT). The pins can also capture signals originating from the DUT. With this captured signal information compared with recorded timestamps, the verification of microcontroller software is made possible. The pins are labeled with symbolic names to help the test operator to wire the correct signals to the microcontroller under testing.

In the upper right part of the stripboard are two static pinstripes. The upper pins are at ground potential, whereas the lower pins are at the 3.3 V operating voltage constantly. Below these two pinstripes there is an additional set of pins for relay functionality. In the construction, a PNP-type transistor is controlling the input power of the microcontroller under testing. A dedicated probe pin `p8` GPIO signal is used to control the conductance of the transistor from emitter pin to collector pin. The DUT input power is regulated 3.3 V voltage from the LPC1768 PCB. This organization makes it possible to control DUT input voltage with weaker control-voltage. A dedicated circuit could be built to draw power directly from stripboard power input without going through the LPC1768 PCB. The relay control pin from the Probe Microcontroller, as well the transistor-control pin, are covered with black heat-shrink tubing in order to prevent the accidental connection of wires. The actual transistor control wiring is on the backside of the stripboard.

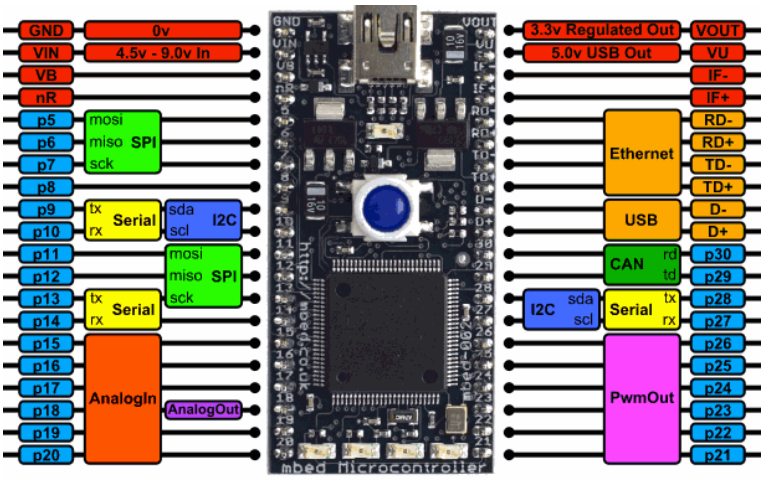

Figure 16: Mbed LPC1768 microcontroller PCB pinout. (Copyright: NXP Semiconductors N.V.)

For testing of the actual microcontroller, the system needs to be powered up,

which happens via the LPC1768 PCB micro-USB[24] connection with 5 V default voltage. The PCB circuitry has built-in converters to scale the voltage down to 3.3 V for both the LPC1768 and DUT to use. Another external connection that is needed is the system UART communication port on pins `p13` and `p14`. Figure 16 shows the pin names and peripheral devices mapping of the LPC1768 microcontroller. As seen in the figure, the system UART communication port was chosen this way to retain the possibility to use I2C communication peripherals on pins `p9`/`p10` and `p27`/`p28` while sacrificing the usability of the SPI peripheral on pins `p11` - `p13`.

5.2 Software Implementation

Software for the developed system was implemented as C code compiled with an ARM GCC[25] compiler. An empty Mbed platform project was used as a basis for the microcontroller probe codebase. The project was converted to regular C code project and readied for further development by removing unnecessary libraries, object files, and references. One benefit of the aforementioned approach was that the necessary processor-specific header and linker files were left intact. The produced minimal project was then continued further.

5.2.1 Overview of Design Decisions and High-Level Architecture

Even though the LPC1768 microcontroller has a floating-point-unit (FPU), it was decided that all of the needed calculations would be done as integer-based. The absence of FPUs has been the baseline in low-end microcontrollers; therefore, there was no reason to hinder portability by writing FPU code which would not work on other types of probe microcontrollers. The simplicity of the mathematical operations also supported the usage of integers. Depending on the situation, the length of the integers used varied between 8 and 32 bits. The granularity of the clock driving the test cases was chosen to be 1 microsecond. In reality, however, there is more variation in the speed the test steps are actualized. main() function of the Probe Microcontroller software is presented in Listing 6.

[24] Universal Serial Bus.
[25] GNU Compiler Collection.

```
int main() {

  uint32_t i = 0;

  // Initialize and start the processor first
  SystemInit();
  for(i = 0; i < SYSINIT_SETTLE_TIME; i++){ ; } // Give time for system to settle

  pin_reset_all(); // Reset all pins to gpio, direction in, weak pullup
  init_wall_clock(); // Initialize clock system to be used with testing
  init_device_list(); // Initialize list of all devices available for testing
  led_init(); // Start initializing individual devices
  pwm_init();
  dac_init();
  relay_init();
  gpio_init();
  reset_uart_flags(); // Start setting up UART communication
  uart_init(UART_NUM2);
  set_sys_uart(UART_NUM2);
  send_uart_poll(pt_sys_uart, (uint8_t*)"ready\r\n", 7); // Print acknowledge

  while(1)
  {
    pwm_handle_monitors(); // Handle PWM signal analysis...
    pwm_handle_monitor_logging(); // ... and logging too
    gpio_handle_monitors(); // GPIO signal monitoring
    handle_uarts(); // Check if UARTs have new input data to be processed
    check_wall_clock(); // Run the test sequencer when in run mode
  }

  return 0;
}
```

Listing 6: The `main()` function of the probe microcontroller software.

As can be seen from the listing, the function organization consists of two main parts. The first part is the collection of initialization routines and the second part is the continuously running handler loop. Another way of implementing the continuous functionalities would have been the usage of timer interrupts. Busy-looping was mainly chosen because of the ease of implementation. In this approach, test event accuracies of 4 us were not uncommon.

There was also a tiny concern about resource overhead. Every time the microcontroller jumps to the so-called interrupt context, a number of instruction cycles are used. As some of the functionalities in the probe system are highly time-dependent with interrupt-coupling, interrupts were ruled out as a driver for the main loop as a precaution. The main loop operates continuously in full-speed checking sequentially for tasks and advancing the test sequencer position when needed.

Some interrupts were still used in achieving the full functionality of the system. They were used in setting flags when, for example, a GPIO pin state is altered. In PWM monitoring mode, the interrupts help keep constant track of the last encountered up and down signal hold times. Actual logging and analysis of the events happening during testing are carried out in non-interrupt context through the main

loop, in which various functions analyze the flags set by interrupts. These functions record changes to the test log.

All notable implemented features of the system are presented in the list below. In the subsections after the list, we highlight some of these features.

- UART-based communication protocol between the Test Controller and the Probe Microcontroller
- Device list with corresponding pin identifiers
- Sequencer for test runs with microsecond granularity
- Sequencer binary parameter and result translation
- Probe type and version identification functionality
- Test sequence resetting, definition, and executing
- Post-test result gathering
- Relay functionality for DUT input voltage setting
- PWM signal setting and capture/analysis
- GPIO pin state setting and capture
- DAC analog voltage setting
- Built-in LED control functionality

5.2.2 UART-Based Communication Protocol

One of the first software-related tasks in the project was to build a communication protocol. UART-driven on platform pins `p13` and `p14` as pointed out in section 5.1 was the protocol selected. The bus speed was defined as the popular 115200 bits per second. Due to the design of the Mbed LPC1768 PCB, on-chip debugging was very hard to achieve and in the end was not actualized at all. Therefore, the development process of the UART communication relied on the usage of the PCB LEDs. They were primitive but essential tools in analyzing set variables and the code paths under scrutiny. Reception functionality of the UART device was implemented as interrupt-driven, but all communication originating from the Probe Microcontroller was chosen to use register polling for sending. After the basic UART functionality was ready, debugging of the system became much easier, with the notable exception of the PWM capture, which is described in Section 5.2.5.

The oscilloscope was essential in the verification of all the implemented functionalities. Figure 17 shows the connection of oscilloscope probes in the early stages of development, whereas Figure 18 shows captured ASCII debug data string “Hello”.

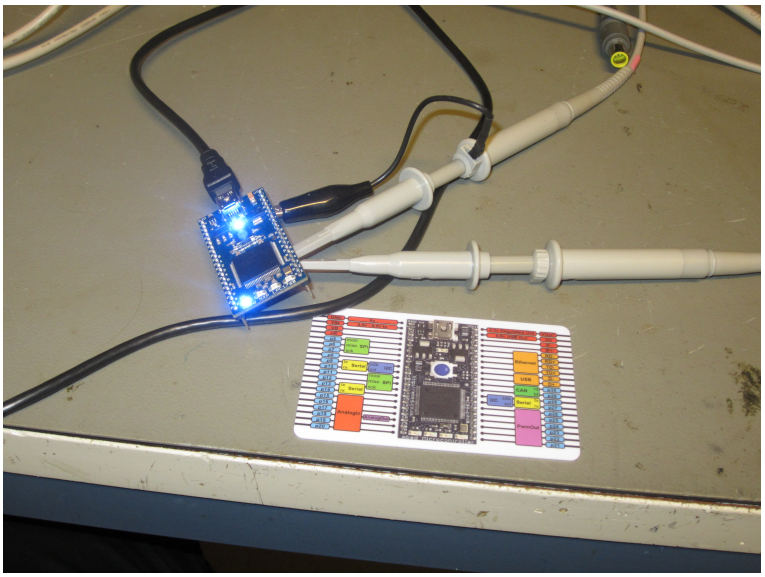

Figure 17: Oscilloscope connection during UART functionality debugging.

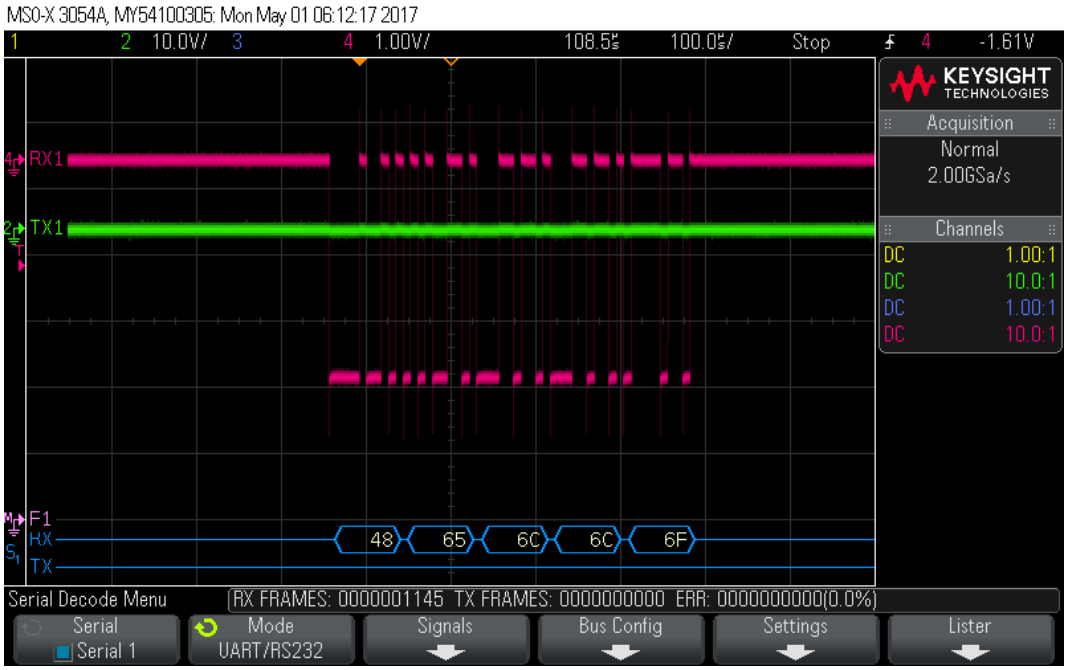

Figure 18: Oscilloscope in analysis mode, after having captured a debug string "Hello".

After the underlying UART device support was mature enough, the actual protocol was implemented. Protocol consists of 3 different commands, namely `identify`, `list devices`, and `test`.

The `identify` command is used to identify which microcontroller probe type

and version is connected. In the future, the Test Controller can use different probes for testing different microcontrollers. Identifying the probe first allows the selection of appropriate test cases. The `list devices` command and `test` command are described in more detail in subsections 5.2.3 and 5.2.4, respectively.

5.2.3 Device List

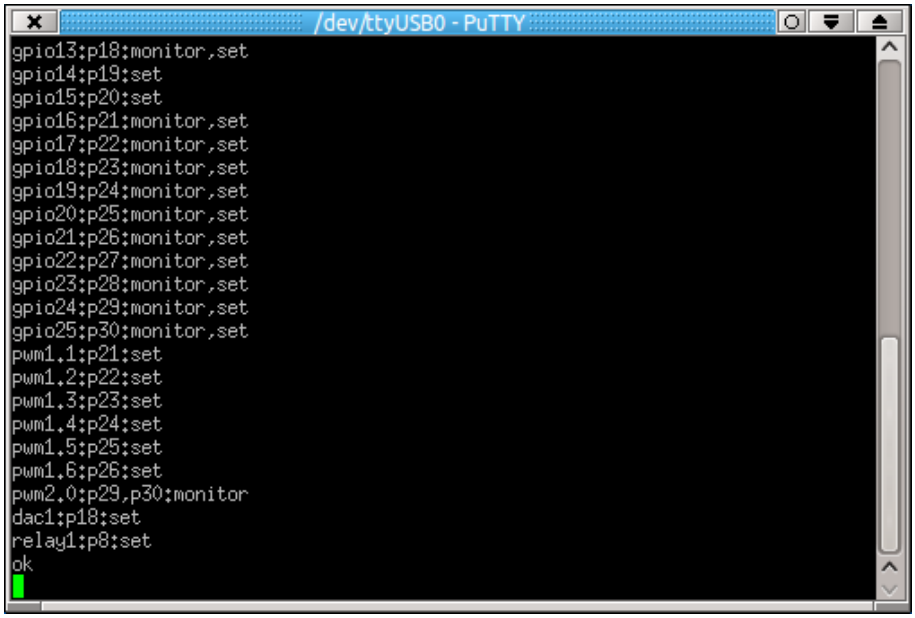

Figure 19: Device listing command response from the Probe Microcontroller.

The device listing command gives an exact listing of the available devices of the specific Probe Microcontroller. The command output is shown in Figure 19. This output tells what functional devices are available for testing and via which pins. The printout shows following types of devices: `gpio`, `pwm`, `dac`, and `relay`. The view conveys extensive important information. First of all, it tells the symbolic device names being used, for example, `gpio22`. For PWM devices, it should be noted that the names sharing the same main numeral share the same PWM clock internally. That means, for example, with device `pwm1.3`, you must use the same base frequency if, for example, `pwm1.5` is used at the same time. More about PWM functionalities is available in section 5.2.5.

Separated from the first field by the symbol `:` is the pin identifier. This identifier corresponds to the printed names on the prototyping board pins, as can be seen in Figure 15. Note the `pwm2.0` device that lists two pins, namely `p29` and `p30`. This kind of listing means that, in order to achieve the declared device functionality, both pins need to be used simultaneously. Reason for this syntax is presented in Section 6.3.

After the final separator, each line has a list of opcodes the specific device implements. For example, the `gpio16` lists both `monitor` and `set`, whereas the

`gpio15` can only do the `set` functionality. Other proposed but not implemented commands are the `read` command for reading instantaneous data and the `send` for sending data via UART, SPI, I2C, and similar devices.

```
typedef struct _dev_desc_t
{
  uint8_t au8_probe_dev_name[DEV_MAP_BUF_SIZE];
  dev_type e_dev_type;
  uint8_t u8_pin_count;
  uint8_t u8_pin_group;
  uint8_t u8_pin_id;
  uint8_t u8_cap_count;
  uint8_t au8_probe_pins[DEV_MAP_DEV_PINS_MAX][DEV_MAP_BUF_SIZE];
  uint8_t au8_processor_pins[DEV_MAP_DEV_PINS_MAX];
  step_e ae_capabilities[DEV_MAP_CAPS_MAX];

  uint8_t (*u8_make_step_params)(step_e e_param_step_type,
                                 uint8_t* pu8_param_command,
                                 uint8_t u8_command_len,
                                 uint8_t* au8_step_params,
                                 uint8_t* pu8_step_params_count);

  uint8_t (*u8_preconfigure_for_step)(struct _dev_desc_t* pt_param_device,
                                      step_e e_param_step_type,
                                      uint8_t* au8_param_command,
                                      uint8_t u8_param_command_len);

  void (*execute_step_params)(struct _dev_desc_t* pt_param_device,
                              step_e e_param_step_type,
                              uint8_t* au8_param_command,
                              uint8_t u8_param_command_len);

  uint8_t (*u8_transform_binary_log)(uint32_t* pu32_param_entry,
                                     uint8_t* au8_result_buffer,
                                     uint8_t* pu8_result_size);
} dev_desc_t;
```

Listing 7: C language definition of generic device type.

C language definition of generic device type is presented in Listing 7. Each device has a unique name, type, number of pins, and number of capabilities the device can perform. Also listed are the actual capabilities as well as MCU-specific group and pin id information. The `au8_probe_pins` holds the textual identifiers for associated device pins; these pins are used to guide the user to connect the correct wires to the microcontroller probe prototyping board. The `au8_probe_pins` is the presentation of the same pins, but for mapping them to actual processor pins instead of the Mbed platform/microcontroller probe board pins.

The rest of the definition consists of different callback function pointers. Whenever a device is initialized, these callback function pointers are connected to device-specific functions. For example, when initializing a gpio device, the functions `u8_gpio_make_step_params`, `u8_gpio_preconfigure_for_step`, `gpio_execute_step_params`, and `u8_gpio_transform_binary_log` are connected. More of these device functions are presented in the next section.

5.2.4 Test Commands and Test Sequencer

As mentioned earlier, the developed Probe Microcontroller includes a test sequencer that is operated by specific test commands. These commands are `reset`, `define`, `add`, `commit` and `run`, and they need to be prefixed with the keyword `test`. There is also the `results get` command for receiving test results after execution.

The `reset` command is used to reset the test program state and erase old test steps. To start defining a new test program, the command `define` needs to be issued. Internally, this command adds a special test step signifying the start of the test program and also increments the global step counter. The `add` command is used to add actual MCU peripherals to the test program. The add command takes 3 common parameters, namely timestamp, device name, and opcode. After opcode, follow the parameters specific to the device operation. For example, the 3rd LED can be lit with the command `test add 300ms led3 set 1` whereas 20% 110 kHz PWM generation can be achieved with the command `test add 100ms pwm1.3 set 1 20% 110kHz`. The specific parameters are run through the `u8_gpio_preconfigure_for_step` callback, which validates the parameters and also performs binary translation for them. This means transforming the textual parameters to binary form, which is much faster to use during the actual execution of the test script. More comprehensive example of a test script is presented in section 7.

The sequencer supports different kinds of definition suffixes. For timing the `us`, `ms`, and `s` are available for microseconds, milliseconds, and seconds, respectively. For frequencies, the `Hz`, and `kHz` suffixes are available. Voltage can be suffixed with `mV` and `V`. A definition of `3.3V` is possible here, because internally the voltages are recorded with millivolt precision.

There exists a special device-like keyword called `end` that is used in conjunction with timestamp to add ending to the text program. The `commit` command acts as a hook point for verification of the complete test script. Issuing the `run` command sets the sequencer to execute the script. In script execution, the Probe Microcontroller software first resets all of the devices and pins implemented in software. This includes setting corresponding flags, initializing filters and connecting interrupt handlers, if needed. After this, the test script is analyzed from the end to the beginning. It configures the chosen devices according to the test steps via the `u8_gpio_preconfigure_for_step` callback function on per-device basis. Finally, the test sequencer clock is initialized and the internal state is set to running.

During execution, whenever the sequencer encounters a new test step according to its defined timeslot, the related device callback code `gpio_execute_step_params` is run. It uses the fast binary parameters defined earlier to execute the desired test step. Logging of test data also happens in binary form. The system shares a binary array of 32-bit integers to store the log data. Whenever a test step is executed or interrupt-based new data received, new entry is made. The entry size is completely dynamic. After the test script has completed execution, the log data can be received via the `results get` command. Another binary translation is performed to transfer the test log data to human-readable form. The callback `u8_gpio_transform_binary_log` is utilized for this purpose.

5.2.5 PWM Signal Capture and Generation

One notable device from the implemented system is the PWM device. It is capable of both generating and capturing PWM signal, but the functionalities happen on different pins due to the Mbed LPC1768 PCB and MCU architecture. PWM capture is requested with the `monitor` keyword and for PWM generation the keyword `set` exists in the test scripts.

Scheduling a PWM capture in a test script is a straightforward thing. The example command `test add 400ms pwm2.0 monitor` sets the capture on for pwm2.0 device when 400 ms of time has elapsed since beginning of testing. All of the notable changes in signal are determined, including the start, end, change of frequency, and duty cycle.

The PWM capture functionality was implemented by using dedicated capture registers of two pins. Both pins have been configured to store amount of clock cycles for respective registers. In addition, an interrupt routine is always running during testing. This routine serves interrupts of the rising edge pin (other pin is detecting falling edges, but without actual interrupts). When an interrupt is received, the code performs the following operations:

1. Interrupt handler clears the pending interrupt.
2. The free-running PWM capture clock is zeroed.
3. The flag is set for the user context to indicate new sample.
4. User context polling finds the flag and runs the monitoring routine.
5. Monitoring routine stores the sample counts from 2 capture registers and adds the correction to facilitate the context switch delay.
6. Routine adds sample values to the software filter and performs sanity check analysis.
7. If there are irregularities in the sample train, filters are restarted.
8. The user context flag is cleared.

There is also another function that is run straight after performing the steps above. This function is for logging purposes. It also performs analysis on the samples of the PWM filter. If there are enough proper values and capture mode is on, the characteristics of the signal are logged. If the signal has not changed since the last analysis time, no new values are recorded. Extra care was taken to have as accurate timestamps in PWM as possible. Filters store the timestamps of the time that the first samples are stored. Therefore, even if the filter value analysis happened later than the actual PWM signal change, the actual change time would still be logged as a timestamp.

The PWM signal generation functionality is also implemented. As has been briefly noted earlier, all of the generating devices share a single clock. This means

that all participating PWM generators need to have the same base frequency. Duty cycle can be arbitrary, but within 1-100%. Base frequency can differ from 1 Hz to 400 kHz.

The Listing 11 from Section A shows a test log from a verification run of PWM capture functionality. In this verification test, the oscilloscope PWM signal generator was set to 200 kHz speed. Duty cycle was initially set to 20%. During testing the duty cycle was manually and continuously rotated to 80%. From the test log it can be seen that the implementation is able to detect duty cycle changes of 2% while operating at 200 kHz. The maximum deviation of frequency is 0.2%. It should be noted, that 200 kHz is the maximum design frequency of the capture. Therefore, the error percentages presented are worst-case errors. With lower frequencies, the errors become even smaller than listed herein above. Figure A1 shows the oscilloscope state at the end of the test.

5.2.6 DUT Input Power Relay

The implemented system also includes relay functionality to input power to the DUT. This relay is controlled by the dedicated Mbed LPC1768 PCB pin `p8`. Relay is implemented by using a PNP type transistor. When the control signal from `p8` is activated (in this case, with logical 0 V), 3.3 V voltage is connected to the `relay1 DUT vcc` pin array. The relay device automatically adjusts the control signal inversion. Therefore, programmatically setting the relay to `1` closes the relay and using the value `0` opens it. Dedicated input power sourced via adjustable relay is necessary to ensure that the DUT is powered up via the exact way wanted.

In this section, the implementation of the proposed test system was presented and analyzed. Hardware implementation was briefly demonstrated, as was software architecture. Important features and concepts, such as PWM generation and capture as well as the test sequencer were also introduced. PWM capture was also benchmarked successfully in a high-frequency environment.

6 Challenges

During the implementation of the developed system there were many challenges, as is sometimes the case with embedded systems projects. Some of the issues are listed in this chapter with the discovered and proposed solutions.

6.1 Error-Prone Pin Configuration

It was discovered in the early stages of development that the complexity of pin configurations and peripheral selections was very high. Thus, the development of pin-related functionalities was extremely error-prone.

The solution was to develop a PHP[26] script that autogenerated the necessary C code. The generator naturally included one set of the pin configuration as presented in the MCU datasheet. With this single-configuration approach generating the missing C code was comfortable and less error-prone than writing everything by hand every time a related functionality was needed. Part of the developed script is shown in Listing 8.

```
function print_gpio_monitor_conf($input_pin_num, $input_pin_data_array)
{
// NEED:
// case PIN_74:
//   LPC_GPIO2->FIODIR0 &= ~GPIO_74_MASK;
//   LPC_GPIOINT->IO2IntEnR |= GPIO_74_INT_MASK;
//   LPC_GPIOINT->IO2IntEnF |= GPIO_74_INT_MASK;
//   break;

  list($pincode, $pinselnum, $pinfunc_pos) = $input_pin_data_array;
  list($groupnum, $pinnum) = explode(".", $pincode);
  $groupnum = substr($groupnum, 1);
  $subnum = floor($pinnum / 8);

  if(($groupnum == 0) || ($groupnum == 2))
  {
    print("case PIN_" . $input_pin_num . ":\n");
    print("LPC_GPIO" . $groupnum . "->FIODIR" . $subnum);
    print(" &= ~GPIO_" . $input_pin_num . "_MASK;\n");
    print("LPC_GPIOINT->IO" . $groupnum . "IntEnR |= GPIO_");
    print($input_pin_num . "_INT_MASK;\n");
    print("LPC_GPIOINT->IO" . $groupnum . "IntEnF |= GPIO_");
    print($input_pin_num . "_INT_MASK;\n");
    print("break;\n");
  }
}
```

Listing 8: An excerpt from PHP script that generates C code for configuring the MCU.

[26] PHP: Hypertext Preprocessor.

6.2 Limited Number of DAC Devices

The absence of DAC devices on the Mbed LPC1768 PCB became a problem. As can be seen from Figure 16, only one such device was available for operations. A partial solution was to use GPIO signals instead. This solution, of course, does not provide smooth pseudo-linear control over the voltage, as only 0 V and 3.3 V voltages were available. Usage of dedicated bus-controlled DAC chips would have been the optimal solution. Another solution would have been the usage of PWM generation combined with hardware filter implemented by a capacitor and a resistor.

6.3 Missing PWM Capture Pins

Implementing the PWM functionalities caused many problems. One elementary problem was that there were no dedicated PWM capture pins available on the Mbed LPC1768 PCB. As can be seen in Figure 20, the necessary pins were not routed out of the PCB or were used for other purposes.

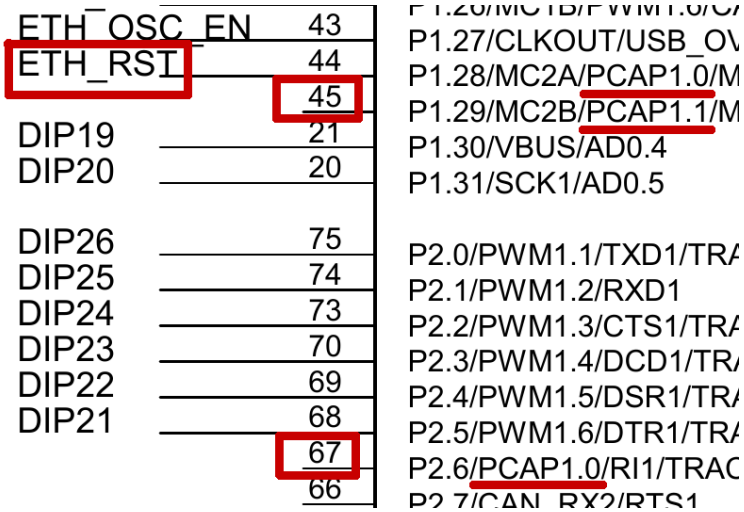

Figure 20: An excerpt from Mbed LPC1768 PCB schematics diagram. (Copyright: NXP Semiconductors N.V.)

The solution to this problem was to use two normal capture pins in combination. One of the capture pins was used to capture the up pulse cycle time of the PWM signal using falling edge. The second capture pin was used to capture the full cycle time of the signal using rising edge. A specific pin combiner was developed to be used with PWM capture. It is shown in Figure 21.

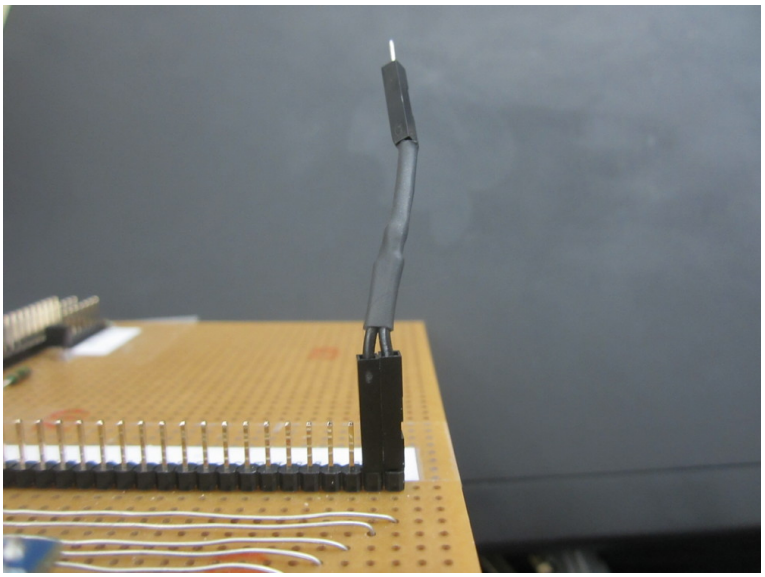

Figure 21: Pin combiner for PWM capture.

6.4 Misbehaving Signal Generator

The misbehaving signal generator of the Keysight MSOX3054A oscilloscope was one of the most significant challenges faced during development. It caused two separate problems. The first problem manifested when changing the generator PWM signal frequency. The captured signal filters started to exhibit serious errors whenever the frequency was changed. After two weeks of developing and debugging, dedicated embedded software was produced to issue GPIO signal when the error condition was encountered. The oscilloscope was set to trigger to the new signal.

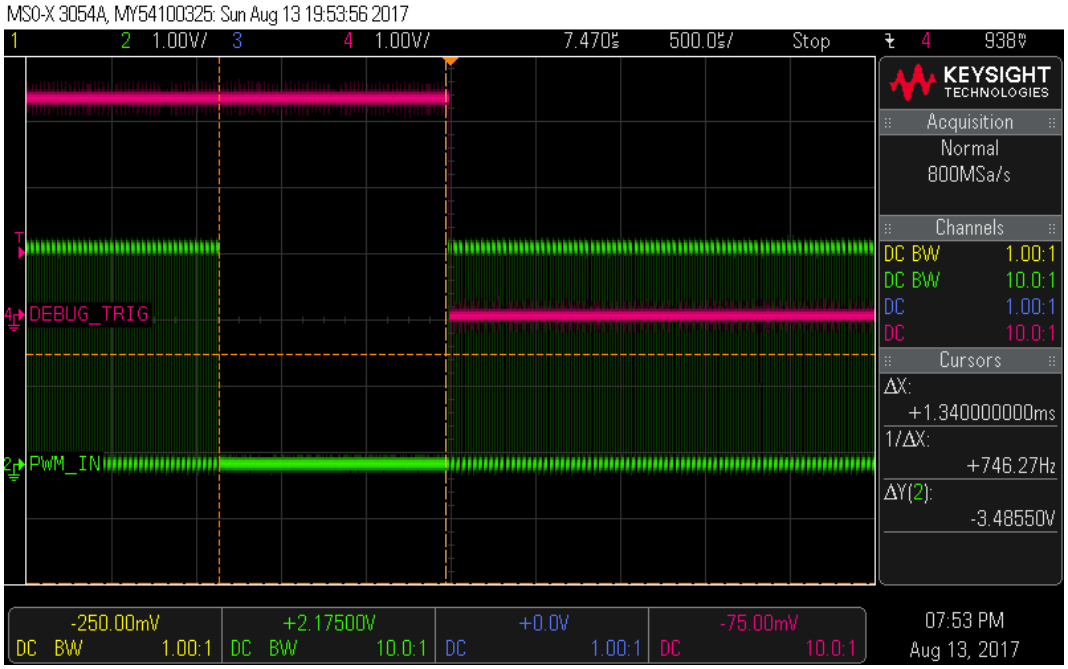

Figure 22: Oscilloscope PWM generator dropping signal upon frequency change.

Figure 22 shows the surprising behavior of the oscilloscope signal generator. Instead of smoothly transitioning to the new frequency, it decides to drop the signal to the ground level for 1.34 milliseconds. As a solution, the PWM filter was adjusted to account for the behavior. The Probe Microcontroller housing signal generator functionality was also proofed and developed into a state where the changing frequency or duty cycle does not cause signal drops like the one experienced in this case.

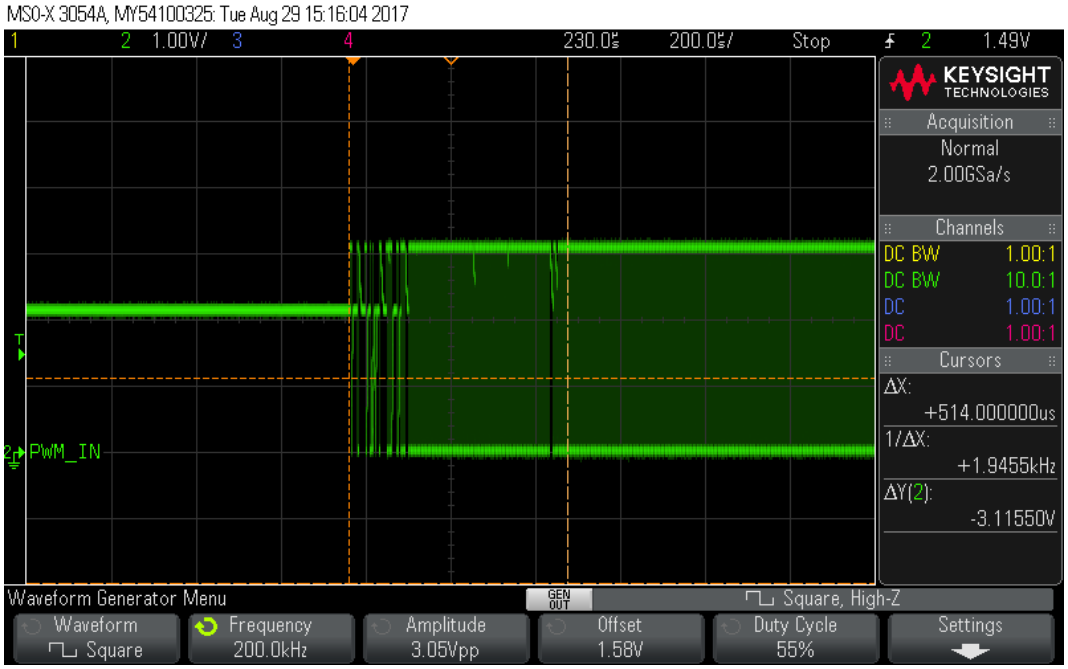

Figure 23: Oscilloscope PWM generator settling interference at 200 kHz.

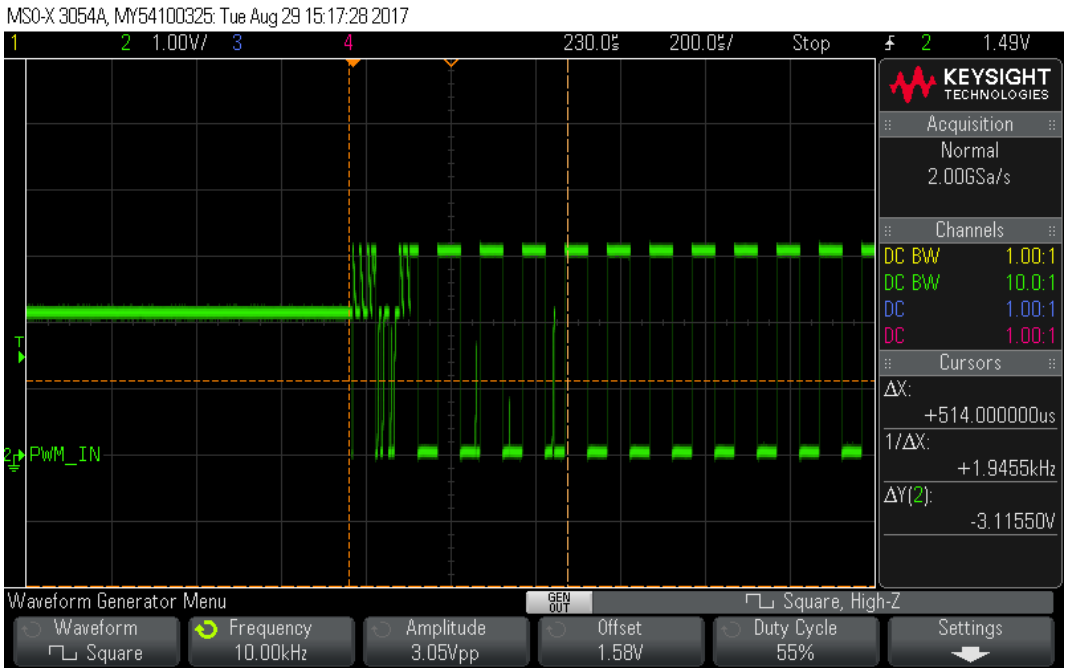

Figure 24: Oscilloscope PWM generator settling interference at 10 kHz.

Another oscilloscope signal generator problem was encountered later. It was discovered that whenever the PWM signal generation is started, there is a period

of instability. This instability has the same kind of form and duration regardless of the frequency, as can be seen by comparing Figure 23 to Figure 24. The solution for offsetting this settling interference was to tweak the PWM capture filters again. A delay from the encountered PWM sample change time was added first. In addition, a methodology for handling bad values was implemented. Bad values are determined by comparing momentary sample values to averaged sample values. Few bad values are allowed (and discarded), but when thresholds are reached, the filters are declared stalled and need to be repopulated from the start. The timestamp of the first acceptable sample is stored to more accurately communicate the starting time of the PWM signal.

6.5 Unstable PWM Sample Counter during UART Communication

The final PWM-related problem faced was encountered while debugging PWM filters. It happened that the signal sample counters were wandering up and down relatively slowly, causing filter stalls. One such filter stall phenomenon can be seen in Figure 25.

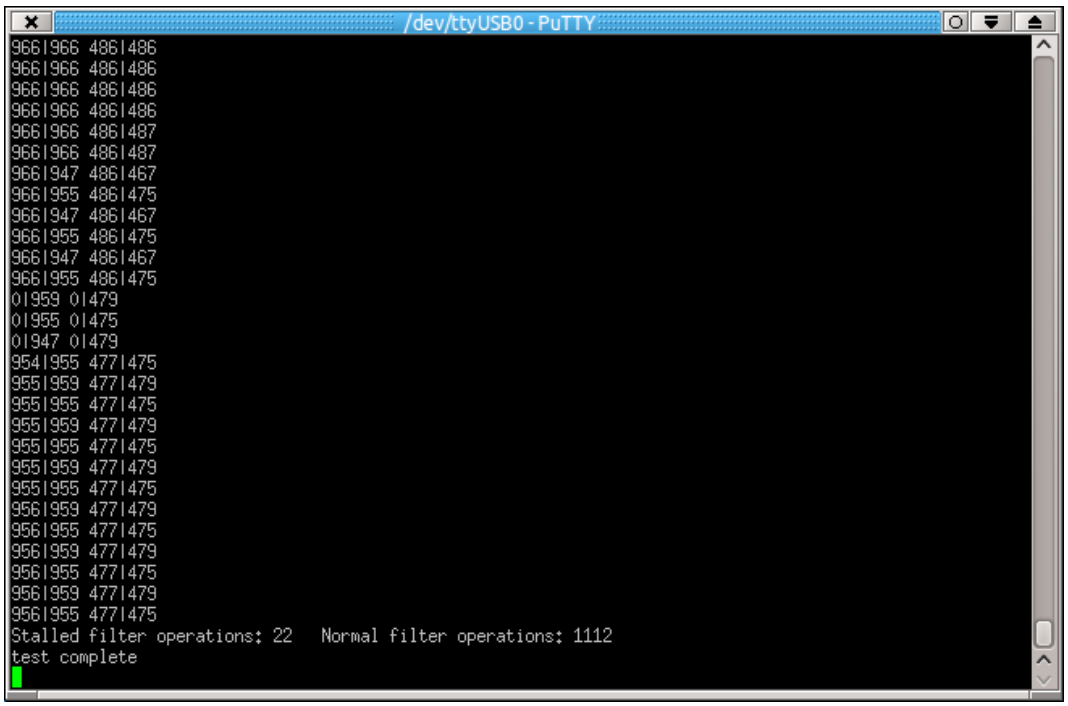

Figure 25: Debug printing during PWM capture at 100 kHz.

It turned out that having the UART-based debug printing activated was the cause of these problems. Without debug printing, the 22 filter stalls observed in Figure 25 dropped to 0 stalls in Figure 26. The actual root cause of the problem remains unknown. The possible reasons could be poor grounding or extra capacitance in pins.

Usage of 2 externally connected pins as single PWM capture device is also probably more error-prone than using a single dedicated capture pin.

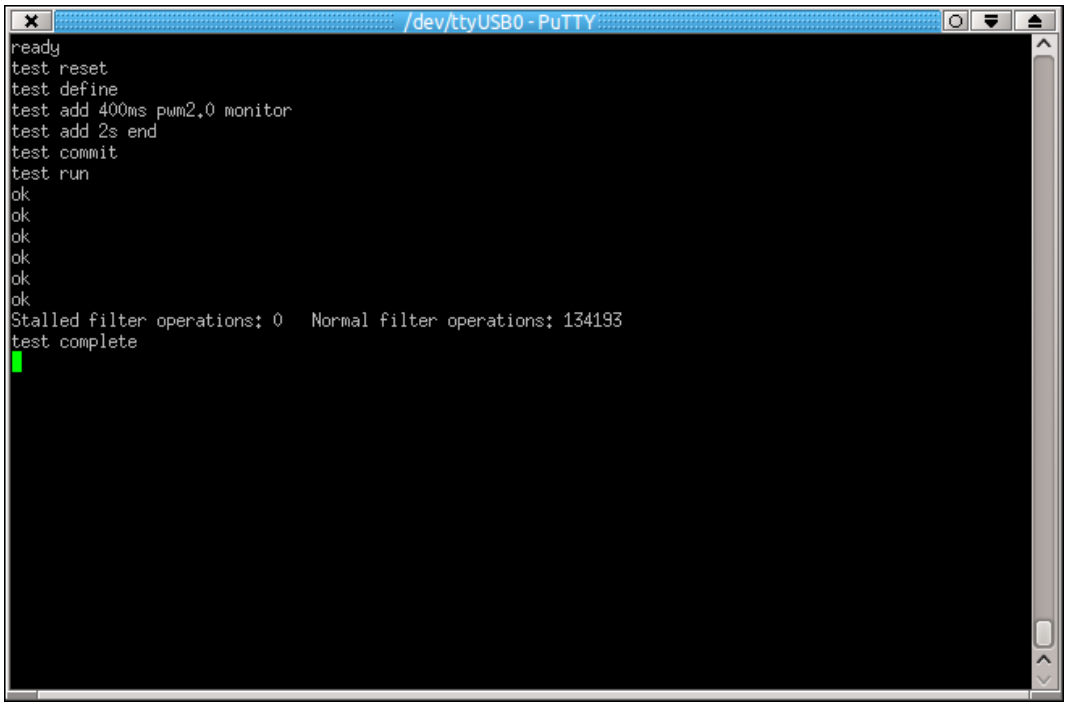

Figure 26: Debug printing off during PWM capture at 100 kHz.

This chapter demonstrated various challenges faced during the development of the Probe Microcontroller. Most of the listed problems included the PWM functionality. It turned out quite interestingly that most time-consuming problems were due to a misbehaving oscilloscope signal generator.

7 Case Study: Sequential Functional Block Activation

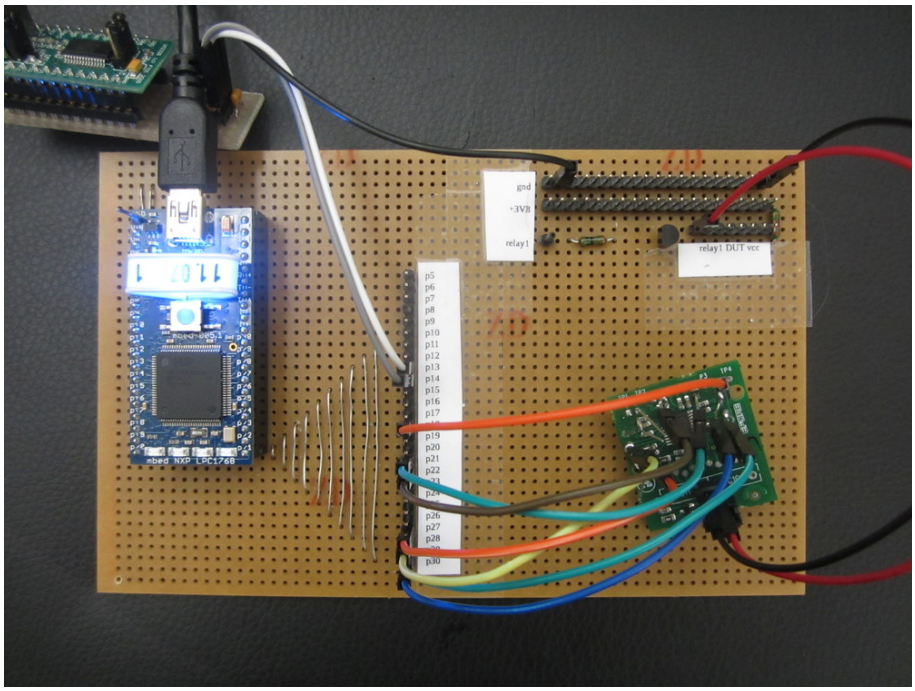

Figure 27: Power electronics product prototype being tested on the developed Probe Microcontroller housing.

The implemented Probe Microcontroller was used to test an actual power electronics product prototype. The results of this test are presented in this chapter. The test setup can be seen in Figure 27. The product prototype on the bottom right is connected via wires to the test board. The prototype is a custom PCB with the small microcontroller running the actual product firmware. The prototype PCB also has the reference resistors and capacitors needed to operate the MCU, as well as external pins to input and read signals.

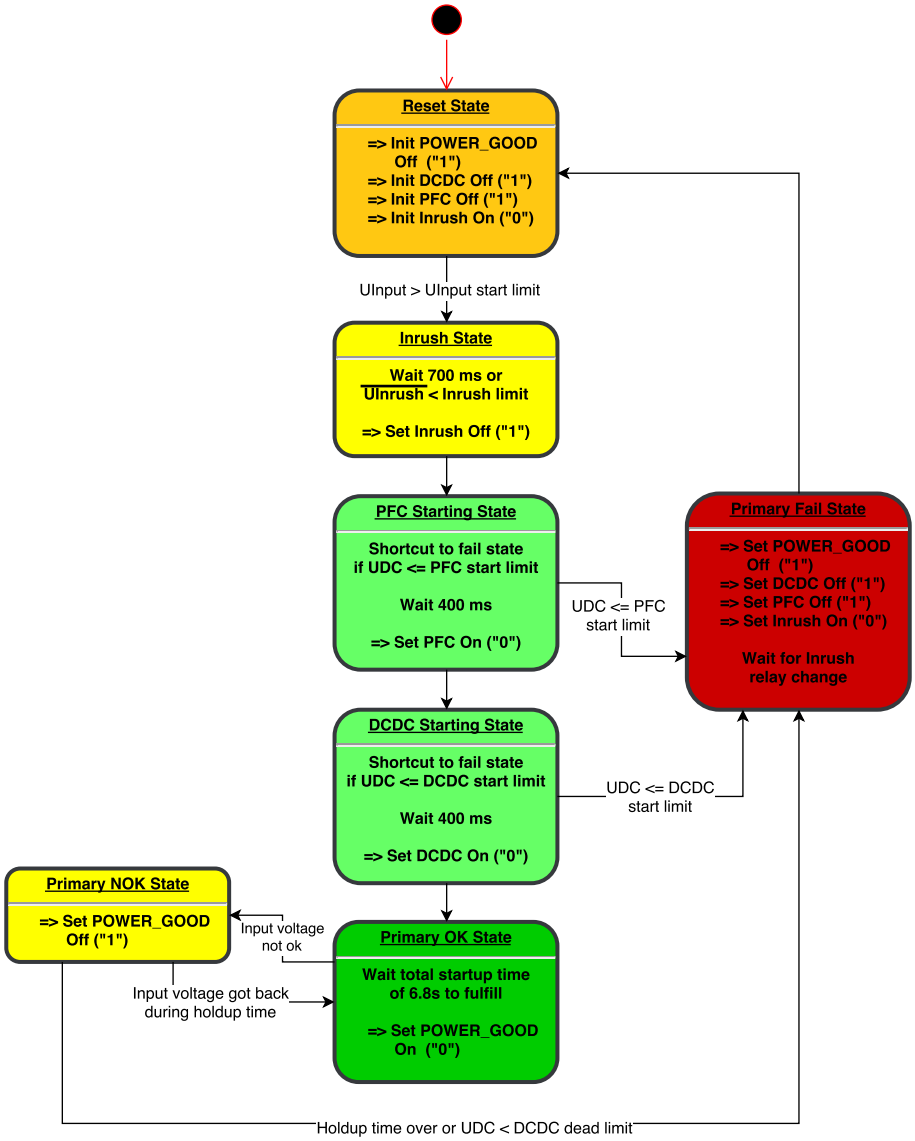

Figure 28: Functional state machine diagram of a power electronics product.

Figure 28 shows the specified functional state machine diagram of the prototype. This diagram has the same basic form as the diagram shown in Section 2.2.4 but in more complicated form. The idea of testing is to verify that the inputs and outputs according to the diagram also reflect the functionality of the actual prototype PCB MCU. In testing, we provide inputs to the PCB MCU and monitor the outputs. The test, in essence, verifies that the functional blocks of the power electronics prototype are activated in precise sequential order. In case the order is wrong, the device would not work as expected. In the worst possible case, the device could get damaged, and sometimes even explode violently.

```
test reset                    #reset the sequence
test define                   #start new sequence
test add 0s gpio22 monitor    #inrush relay state (1=disconnected)
test add 0s gpio23 monitor    #monitor PFC block state(0=active)
test add 0s gpio24 monitor    #monitor DCDC block state (0=active)
test add 0s gpio25 monitor    #monitor POWER_GOOD signal (0=active)
test add 0s dac1 set 0V       #zero DUT VCC voltage on reset
test add 0s gpio16 set 0      #set inverted Uinrush voltage for waiting
test add 0s gpio17 set 0      #set UDC measurement for failing
test add 1s relay1 set 1      #connect VCC to DUT
test add 5s dac1 set 0.5V     #input voltage < start limit
test add 10s dac1 set 0.7V    #input voltage > start, but < full ok
test add 13s end              #end marker
test commit                   #sequence ending
test run                      #run the sequence
```

Listing 9: Script to test partial functionality of the prototype PCB MCU.

Listing 9 shows the first test script used to test the state machine of the prototype device. As stated in script comments, the state of inrush relay, `PFC` block, `DCDC` block, and `POWER_GOOD` signal are monitored with GPIO pins. With this specific test, the idea is to show that when the prototype device input voltage measurement is below a limit, the state machine does not advance. The test should also show that when the input is raised above the limit, the state machine advances, but due to the `UDC` voltage has been set to zero, the state machine should loop in states shown the upper parts of the diagram. It should also be noted that the UInrush measurement is inverted in the said diagram.

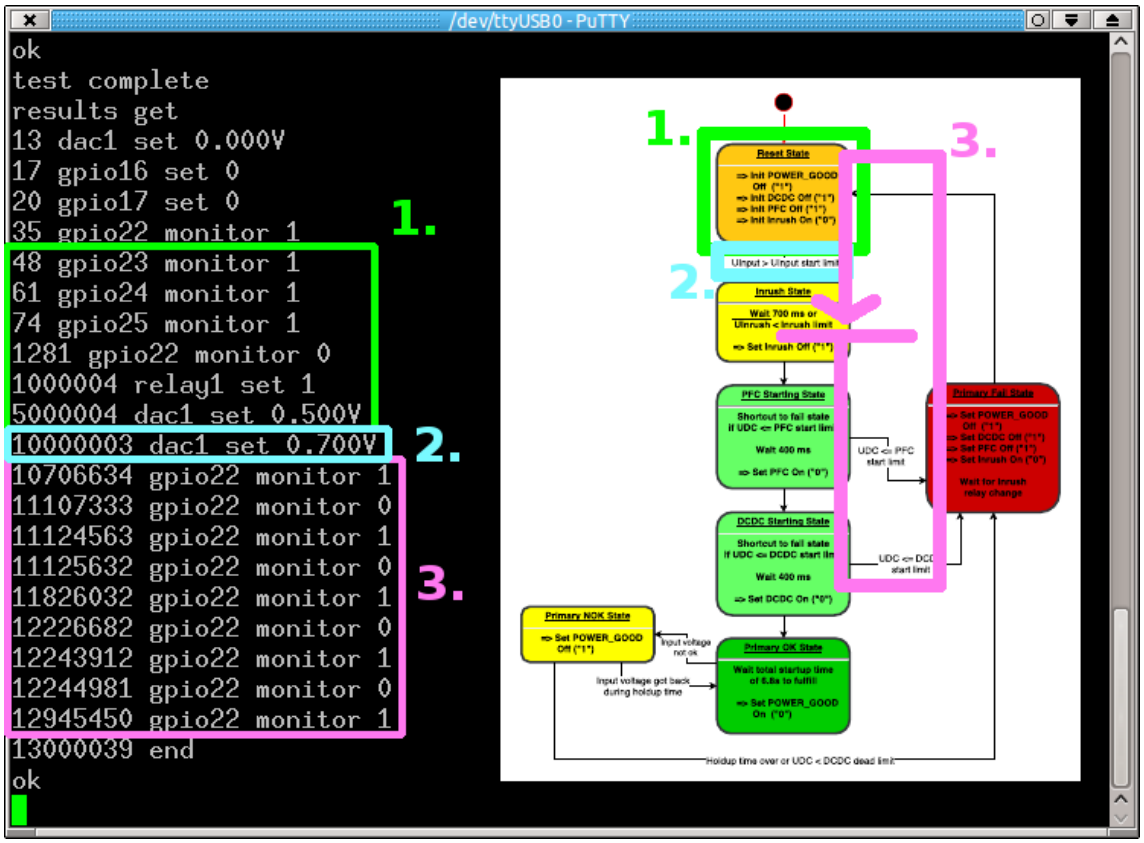

Figure 29: Test data from the Probe Microcontroller in non-finishing case, combined with execution relative to the state machine.

Figure 29 shows the test results with relation to the state machine. It can be seen that, in region 1 of the chart, the signals are kept initialized to the expected values, although it should be noted that the `gpio22` signal for inrush relay is first high and then turns to low after 1 ms. This change happens even though the prototype PCB MCU is powered down. The correct signal configuration is in effect when the system is powered via `relay1` device at around 1000 ms.

No test results are logged until approximately 5000 ms, at which point the input voltage measurement of the prototype device is set to 500 mV via the `dac1` device in the end of region `1`. The state machine does not advance due to the fact that 500 mV is below the threshold to allow this. The next line, in region `2`, sets the voltage above the threshold to allow for an advance. After this, the code path marked with symbol `3` is executed. This results in looping behavior as was the original hypothesis. From the microsecond-based timestamps in the results, it can be verified that, after the correct input voltage is applied, there is a 700 ms second delay before inrush relay disconnect. After 400 ms, there is a relay connect with another disconnect/connect cycle. The last cycle is due to a device reset in the reset state.

Even though the execution clearly loops via the path marked with symbol `3`, there is one seemingly strange behavior. It can be observed that the state of `PFC` is not changed during the loop, even though there is activation and deactivation in

```
test reset                    #reset the sequence
test define                   #start new sequence
test add 0s gpio22 monitor    #inrush relay state (1=disconnected)
test add 0s gpio23 monitor    #monitor PFC block state(0=active)
test add 0s gpio24 monitor    #monitor DCDC block state (0=active)
test add 0s gpio25 monitor    #monitor POWER_GOOD signal (0=active)
test add 0s dac1 set 0V       #zero DUT VCC voltage on reset
test add 0s gpio16 set 1      #set inverted Uinrush voltage for passing
test add 0s gpio17 set 1      #set UDC measurement for passing
test add 1s relay1 set 1      #connect VCC to DUT
test add 5s dac1 set 2.7V     #set input voltage to full ok
test add 13s end              #end marker
test commit                   #sequence ending
test run                      #run the sequence
```

Listing 10: Script to test completing functionality of the prototype PCB MCU.

the code path. The observation can be explained by the slow settle time of the `PFC` signal. After activating the signal in the `PFC Starting` state, the state machine is advanced to the `DCDC Starting` state. As the `UDC` measurement fails at this point, it short-circuits the execution directly to the `Primary Fail` state. In this state, the `PFC` signal is deactivated. This deactivation happens too fast to be noticed regarding the previous activation or the signal has no actual time to change at all. Therefore, it seems that `PFC` signal was not touched and that the `PFC Starting` state was never visited, even though this has not been the case.

Another test was also conducted on the state machine of the prototype device. Listing 10 shows the second test script. The idea of the test is to make execution reach the `Primary OK` state as smoothly as possible after input voltage via `dac1` has been set. A notable change to previous script other than the aforementioned device is setting `gpio16` to 1 to enable fast passing in the `Inrush` state. The second change is setting `gpio17` to 1 to enable passing in the `DCDC Starting` state.

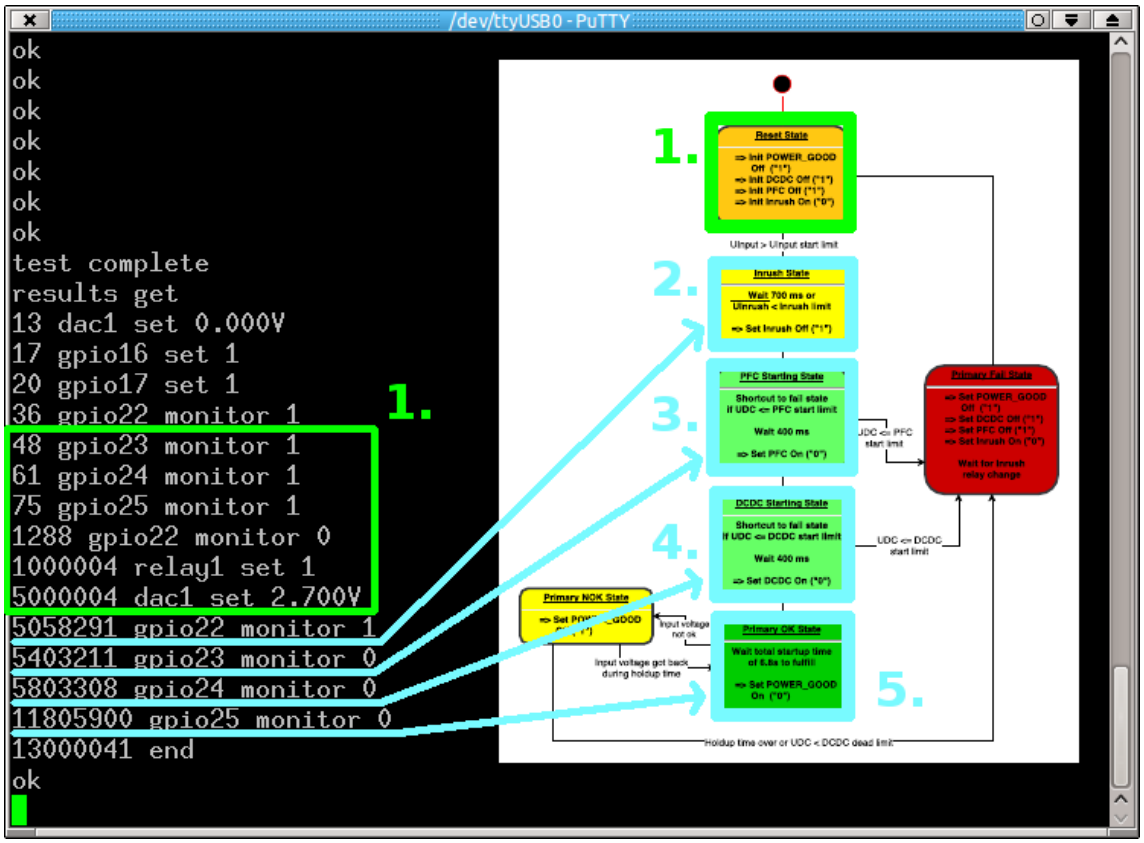

Figure 30: Test data from the Probe Microcontroller in finishing case, combined with execution relative to the state machine.

Figure 30 shows which test results were recorded, and to what part of the state machine diagram that they correspond to. The first difference happens after the `dac1` line. Whereas in the previous test there was a 700 ms wait time, now it is almost entirely absent. This behavior happens because the inverted `UInrush` voltage is set to 1 via `gpio16`, signifying zero physical voltage. No wait time is needed at this point. After the 700 ms wait, there is a 400 ms wait for the states `PFC Starting` and `DCDC Starting`, respectively. The `POWER_GOOD` signal activation requires that 6.8 seconds have passed since the input voltage to the prototype device under test was set. This voltage was set at 5000005 us. The signal activated at 11805900 us. The difference of these numbers is 6805895 us, or roughly 6.8 s, as required in the specification.

During the testing of the two cases above, it was discovered that the original state machine drawing as documentation differed significantly from the actual implementation. The diagram was later corrected to reflect reality. It can be said that the developed probe for microcontroller testing was successful in catching a documentation bug from an actual prototype power electronics product.

In this section, we discovered how a Mbed LPC1768-based Probe Microcontroller housing system was utilized to test an actual power electronics product prototype

running embedded software. The test results were analyzed and their relevance to specifications was also shown. It turned out that the prototype state machine diagrams had actual documentation bugs, which were exposed with the developed system.

8 Conclusions and Future Work

This section evaluates how the project was able to meet the design goals and requirements stated earlier in the thesis. Notable enhancements are also presented for the future.

8.1 Conclusions

The goals and requirements for the implemented system were set in Section 4.1. One requirement was to "build a system that could be used to test embedded software running on microcontrollers in power electronic applications". This requirement was partially met. Due to resource and timing constraints, the full system with all of the components and their integration was not realized. However, the Probe Microcontroller component was implemented. It was also successfully demonstrated to be able to test real power electronics prototypes as shown in Section 7. In this case study, it was also shown that the developed component was able to measure and generate the necessary signals with adequate speeds, therefore fulfilling another requirement.

As the Test Controller component was not implemented due to the aforementioned constraints, the automatic test mode requirement was not met. However, the implemented component can be used in manual mode, thereby partially fulfilling a requirement. As the Test Controller was not implemented, the repository and user interface goals were not met. The low cost of the system was achieved for the microcontroller probe, as components for building the housing and acquiring the Mbed PCB cost less than 100 EUR in the currency of 2017.

The best thing in the realization of the Probe Microcontroller was the fact that it was successfully used to test an actual power electronics prototype in a time-dependent environment[27]. An even more captivating discovery was that the developed component was actually able to catch a real bug in the project documentation. To make testing feasible, up-to-date documentation is a necessity.

8.2 Enhancements and Future Work

For future implementations of the Probe Microcontroller, many enhancements are proposed. The most important thing is to offset the lack of multiple DACs. Having one such peripheral available is not feasible for testing. GPIO pins to emulate DACs is unsatisfactory, as GPIOs may feedback unwanted voltage to the tested microcontroller. The resolution is the use of external DAC chips, which are controlled via I2C or SPI protocols. The abstraction of the existing DAC devices should be kept, but there is one important addition. A voltage slope functionality should be implemented. A slow ramp-up time of voltages can expose hidden problems in power electronics products running embedded software and can also be used to emulate the constantly changing temperature in the product.

[27] Time-dependency is a distinctive aspect of the power electronics context.

Slope functionality should also be implemented for the input power of the device under test (DUT). It should also be made possible to dip the DUT input voltage below 0 V, as temporary brownouts and under-voltages can cause unexpected behavior in microcontrollers. Sometimes this behavior is even unknown to the vendors and can cause extensive delays in debugging and development. The current transistor-based input voltage relay control needs to be superseded with a linear voltage dip-capable design. Implementing all of the necessary hardware changes means that there needs to be an actual custom circuit board designed for housing all of the related components and pathways.

A few completely software-based functionalities could also be implemented, for example the `monitor` functionality of the DAC device to read momentary voltages. In addition, as interconnected microcontrollers in power electronics sometimes need mutual communication, the bus functionalities of the implemented system should be significantly extended. New hardware buses, such as SPI or I2C, should be made available and operable with `send` and `monitor` functionalities. Generic or product family based protocols could also be implemented on top of the hardware buses.

Regarding PWM capture, the implementation currently captures only duty cycle and frequency. As a sine-form PWM signal can be used in power electronics as a regulation control signal, the generation and capture should also support sine-forms in the Probe Microcontroller. Another capture option is the monitoring of the minimum and maximum duty cycle and frequency values.

The implemented solution in this thesis operates on 3.3 V logic level. Therefore, it may be incompatible with other logic levels, such as 5 V or 2.5 V. Level shifters may be an adequate way to resolve the problem. It should be investigated though, if this solution is acceptable regarding the rise and fall times of signals, especially in high-speed bus communication. More research should also be conducted about usage of entirely interrupts-driven test sequencer.

This thesis demonstrated that it is possible to build a test system for microcontrollers using relatively cheap off-the-shelf parts. The test system was also verified as being able to test the validity of both implementation and documentation of a power electronics prototype.

References

[1] Ball, T. The concept of dynamic analysis. In *ACM SIGSOFT Software Engineering Notes* (1999), vol. 24, Springer-Verlag, pp. 216–234.

[2] Boehm, B., and Basili, V. R. Software defect reduction top 10 list. *Foundations of empirical software engineering: the legacy of Victor R. Basili 426* (2005), 37.

[3] Boehm, B. W. Software risk management: principles and practices. *IEEE software 8*, 1 (1991), 32–41.

[4] Bose, B. K. *Modern power electronics*. IEEE, 1992.

[5] Bouscayrol, A. Different types of hardware-in-the-loop simulation for electric drives. In *Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on* (2008), IEEE, pp. 2146–2151.

[6] Broekman, B., and Notenboom, E. *Testing embedded software*. Pearson Education, 2003.

[7] Burnstein, I., Suwanassart, T., and Carlson, R. Developing a testing maturity model for software test process evaluation and improvement. In *Test Conference, 1996. Proceedings., International* (1996), IEEE, pp. 581–589.

[8] Erickson, R. W., and Maksimovic, D. *Fundamentals of Power Electronics*. Springer Science & Business Media, 2001.

[9] Information technology equipment - Safety - Part 1: General requirements. Standard, International Electrotechnical Commission, 2005.

[10] Isermann, R., Schaffnit, J., and Sinsel, S. Hardware-in-the-loop simulation for the design and testing of engine-control systems. *Control Engineering Practice 7*, 5 (1999), 643–653.

[11] Jones, C. Software quality metrics: Three harmful metrics and two helpful metrics, 2012.

[12] Karlesky, M. J., Bereza, W. I., and Erickson, C. B. Effective test driven development for embedded software. In *Electro/information Technology, 2006 IEEE International Conference on* (2006), IEEE, pp. 382–387.

[13] Khan, M. E., Khan, F., et al. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Sciences and Applications 3*, 6 (2012), 12–1.

[14] Kuusijärvi, T. An FPGA implementation of a power converter controller. Master's thesis, Aalto University School of Electrical Engineering, 2017.

[15] Lee, E. A. Embedded software. *Advances in computers 56* (2002), 55–95.

[16] Lee, E. A., and Seshia, S. A. *Introduction to embedded systems: A cyber-physical systems approach*. Lee & Seshia, 2015.

[17] Leung, H. K., and White, L. Insights into regression testing (software testing). In *Software Maintenance, 1989., Proceedings., Conference on* (1989), IEEE, pp. 60–69.

[18] Lewis, J. P. *Project Manager's Desk Reference*. Probus Pub. Co., 1993.

[19] Mohan, N., Undeland, T. M. R., William, P., Tore, M. U., William, P. R., Heumann, K., Möltgen, G., Moeller, F., and Werr, T. Power electronics: converters, applications, and design. Tech. rep., John Wiley & Sons, 2003.

[20] Motor Industry Software Reliability Association. Misra c : 2012. https://www.misra.org.uk/MISRAHome/MISRAC2012/tabid/196/Default.aspx, 2012. [Online; accessed 2017-09-13].

[21] Oehlert, P. Violating assumptions with fuzzing. *IEEE Security & Privacy 3*, 2 (2005), 58–62.

[22] Oshana, R. *DSP software development techniques for embedded and real-time systems*. Newnes, 2006.

[23] Paalijärvi, J. mbed-lpc1768-ledtimer. https://github.com/usvi/mbed-LPC1768-ledtimer, 2017. [Online; accessed 2017-10-22].

[24] Paalijärvi, J. Simple mbed lpc1768 led timer demo. https://www.youtube.com/watch?v=JxRWRIWEx48, 2017. [Online; accessed 2017-10-22].

[25] Rich, N., and Taylor, R. Linear versus switching regulators in industrial applications with a 24-v bus. *National Semiconductor Corporation, Santa Clara, CA, USA* (1995).

[26] Sen, K., Marinov, D., and Agha, G. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes* (2005), vol. 30, ACM, pp. 263–272.

[27] Stolberg, S. Enabling agile testing through continuous integration. In *Agile Conference, 2009. AGILE'09.* (2009), IEEE, pp. 369–374.

[28] Stollon, N. *On-chip instrumentation: design and debug for systems on chip*. Springer Science & Business Media, 2011.

[29] Stroustrup, B. Abstraction and the c++ machine model. In *International Conference on Embedded Software and Systems* (2004), Springer, pp. 1–13.

[30] van Vliet, H. Software engineering: Principles and practice.

[31] Vector Software Inc. c and c++ unit testing tool | vectorcast | vector software_2017. https://www.vectorcast.com/software-testing-products/c-unit-testing, 2017. [Online; accessed 2017-09-13].

[32] White, R. V. Introduction to the pmbus. *System Management Interface Forum* (2005).

[33] Wikipedia. Digital signal processor — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Digital_signal_processor&oldid=745931807#Instruction_sets, 2016. [Online; accessed 2016-11-14].

[34] Wikipedia. Lint (software) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Lint_%28software%29&oldid=798994900, 2017. [Online; accessed 2017-09-13].

[35] Wolf, W. H. Hardware-software co-design of embedded systems [and prolog]. *Proceedings of the IEEE 82*, 7 (1994), 967–989.

[36] Zhivich, M., and Cunningham, R. K. The real cost of software errors. *IEEE Security & Privacy 7*, 2 (2009).

A PWM Capture of a 200 kHz Signal

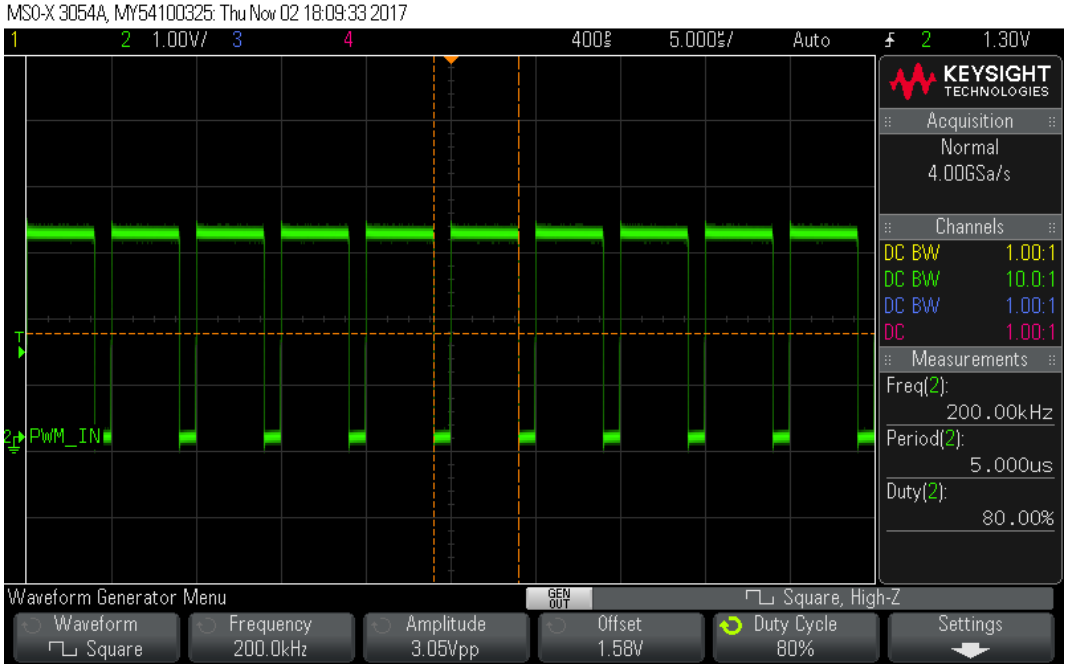

Figure A1: Oscilloscope PWM signal generator at 200 kHz after having cycled the duty cycle from 20 % to 80 %.

```
ok
test complete
results get
400013 pwm2.0 monitor 20% 200000Hz
2131261 pwm2.0 monitor 21% 200000Hz
2253288 pwm2.0 monitor 22% 200000Hz
2306845 pwm2.0 monitor 23% 199584Hz
2357435 pwm2.0 monitor 25% 200000Hz
2420022 pwm2.0 monitor 26% 200000Hz
2601937 pwm2.0 monitor 27% 200000Hz
2662173 pwm2.0 monitor 28% 200000Hz
2720818 pwm2.0 monitor 30% 200000Hz
2804492 pwm2.0 monitor 31% 200000Hz
2859565 pwm2.0 monitor 33% 199584Hz
2923412 pwm2.0 monitor 34% 200000Hz
2948104 pwm2.0 monitor 36% 199584Hz
2966143 pwm2.0 monitor 37% 200000Hz
3491534 pwm2.0 monitor 39% 199584Hz
3606333 pwm2.0 monitor 40% 200000Hz
3647627 pwm2.0 monitor 41% 200000Hz
```

```
3698087 pwm2.0 monitor 43% 200000Hz
3759150 pwm2.0 monitor 44% 200000Hz
3822337 pwm2.0 monitor 45% 200000Hz
3906407 pwm2.0 monitor 46% 200000Hz
3941075 pwm2.0 monitor 48% 200000Hz
3980760 pwm2.0 monitor 49% 200000Hz
4043639 pwm2.0 monitor 50% 200000Hz
4172005 pwm2.0 monitor 51% 200000Hz
4574371 pwm2.0 monitor 53% 199584Hz
4627608 pwm2.0 monitor 54% 199584Hz
4689781 pwm2.0 monitor 55% 200000Hz
4710675 pwm2.0 monitor 57% 200000Hz
4735286 pwm2.0 monitor 58% 200000Hz
4812761 pwm2.0 monitor 59% 200000Hz
4846390 pwm2.0 monitor 60% 199584Hz
4897360 pwm2.0 monitor 62% 200000Hz
4976422 pwm2.0 monitor 63% 200000Hz
5008740 pwm2.0 monitor 64% 200000Hz
5045590 pwm2.0 monitor 66% 200000Hz
5073543 pwm2.0 monitor 67% 200000Hz
5491913 pwm2.0 monitor 68% 200000Hz
5522636 pwm2.0 monitor 70% 200417Hz
5552346 pwm2.0 monitor 71% 200000Hz
5606024 pwm2.0 monitor 72% 200000Hz
5628413 pwm2.0 monitor 73% 200000Hz
5650610 pwm2.0 monitor 75% 200000Hz
5671120 pwm2.0 monitor 76% 200000Hz
5720515 pwm2.0 monitor 77% 200000Hz
5756668 pwm2.0 monitor 78% 200000Hz
5793115 pwm2.0 monitor 80% 200417Hz
10000010 end
ok
```

Listing 11: Test log from 200 kHz PWM signal capture when alternating the duty cycle from 20 % to 80 %.