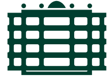

Faculty for Computer Science
Operating Systems Group

Term paper

Winter semester 2017/2018

Micropython

Porting Micropython to Bare Metal Raspberry Pi

Submitted by: Naumann, Stefan
stefan.naumann@informatik.tu-chemnitz.de

Programme of study: Master Informatik
2nd semester
Matriculation number: XXXXXX

Supervisors: Dr. Peter Tröger
Christine Jakobs

Submission deadline: 1st December 2017

Contents

1 Introduction **1**

2 Platform Considerations **2**
2.1 Raspberry Pi 2
2.1.1 Building a bootable image 2
2.1.2 Test environment 4

3 Porting Micropython and running code **7**
3.1 Port structure 7
3.2 Understanding the build-system 7
3.2.1 The Makefile 8
3.2.2 The code 10
3.3 Starting Micropython 10

4 Extending Micropython **13**
4.1 Extending Micropython 13
4.1.1 Creating and adding an empty module 13
4.1.2 Functions 15
4.1.3 Classes 16
4.1.4 Exceptions 18
4.1.5 Conclusion 18
4.2 Interrupt Handling 18
4.2.1 The inner workings of `MP_STATE_PORT` 21
4.2.2 Restrictions for Interrupt Service Routines 22

5 On code execution **24**

6 Blinking LED demo **26**

7 Conclusions **28**

Bibliography **XVI**

Listings

2.1 C-structure for mapping the GPIO-registers on BCM2835 3
2.2 Set-up the interrupt table on the Raspberry Pi 5
2.3 Registers of the interrupt controller of the Raspberry Pi 6

3.1 First Makefile of the Raspberry Pi port 8
3.2 Simple for test-loop 11
3.3 The wrong Bytecode generated by the Raspberry Pi port initially 11
3.4 The correct Bytecode generated by the QEMU port, see Appendix C 12
3.5 Setting the U-Bit in the C15-co-processor to allow the ARM-processor to load unaligned data 12

4.1 Defining an empty module 13
4.2 Type definition of `mp_map_t` 14
4.3 Type definition of `mp_obj_t` 14
4.4 Inform Micropython about its new builtin module (in *mpconfigport.h* 15
4.5 First function, prints out an integer number (as pointer) 15
4.6 Adding the function object to the table of global objects 15
4.7 Adding the function with one argument 15
4.8 Outline of a simple raw-pointer class 16
4.9 Type definition of `mp_print_t` 18
4.10 Enum definition of `mp_print_kind_t` 18
4.11 Raising an exception 18
4.12 timer.callback method for registering a timer interrupt 19
4.13 Timer 1 interrupt handler 19
4.14 Timer interrupt handler 19
4.15 Data structure of `mp_state_ctx` 21
4.16 Macro definition of the macro `MICROPY_PORT_ROOT_POINTERS` in the stm32-port 21
4.17 Pre-allocating an exception buffer 22

5.1 Code for parsing, compiling and executing Python-code 24

6.1 Python-code for blinking LED-demo 26

1 Introduction

Micropython is a Python 3 interpreter developed specifically for embedded ARM processors. The interpreter was originally developed in 2013 by Damien George. It has been targeted for the PyBoard hardware, which contains an ARM Cortex-M3 microcontroller. It has since been ported to other processors and boards like ATMega processors, stm32, ESP32 and the BBC MicroBit board.

The PylotOS project focuses on developing an operating system in an interpreted programming language like Python. It is necessary to have an interpreter up and running before work on the operating system itself can be commenced.

There are a number of free Python interpreters, namely CPython, Pypy and Micropython; many others are discontinued or have a rather niche target platform. There are also some compilers for compiling Python code into code in other programming languages, like Jython, Cython or RPython. PylotOS runs Python code in the virtual machine created by the interpreter. It is not the main goal of PylotOS to write Python code and then compile it to C or any language, but to execute the Python code on the target hardware.

This paper will focus on porting Micropython to the Raspberry Pi hardware. The code will run bare metal, i.e. without an operating system, on the platform. It outlines the necessary steps to boot the Micropython interpreter on the system and run Python code built into the image. Additionally a module is added to Micropython and a demo is shown, to blink the ACT-LED on the Raspberry Pi.

The following Section describes the Raspberry Pi target hardware as well as its booting process, the next outlines the Micropython repository [5], the existing ports and the steps necessary to create a Raspberry Pi port. Some odd behaviours of the hardware will also be addressed. At last, the interpreter will be extended by the means of a module, the Micropython way of implementing interrupt service routines and emitting code will be described and the blinking LED demo will be prepared.

2 Platform Considerations

This chapter focuses on the Raspberry Pi hardware and they way it is programmed on the operating system level. It will outline the boot process and the process of creating bootable images for the device. It will only describe the Raspberry Pi and the Raspberry Pi 0, not Raspberry Pi 2 nor Raspberry Pi 3. The latter feature a quad-core processor and are not considered in this thesis.

2.1 Raspberry Pi

The Raspberry Pi features a BCM2835 chip, which includes the main processor core and many peripheral hardware device controllers. The main CPU is an ARMv6 ARM1176jfz-s processor. The processor contains 16 32-bit general purpose registers. Some of them have another function, like the link register, stack register or program counter, which are also visible as normal general purpose register. The program status register (`cpsr`) cannot be accessed by `mov`-instructions but by `mcr` and `mrc`.

Booting the Raspberry Pi is a bit different from normal PCs, because the first thing to boot-up is the VideoCore IV graphics processing unit. It will open a certain file on the SD-card and execute it. Under normal circumstances, i.e. with standard Raspbian start-up code, it will then load the kernel, initialise hardware devices and start the main ARM CPU.

2.1.1 Building a bootable image

When the GPU starts the ARM-processor like depicted in the previous Section, the ARM starts to execute code at the memory address 0x00000000 in its own address space. It executes ARMv6 code then, which can be generated from C-code by the GCC, the GNU C-compiler. The necessary compiler, assembler and linker can be installed by selecting these packages in their respective distributions:

- Ubuntu: `gcc-arm-none-eabi` for the compiler and `binutils-arm-none-eabi` for the utilities.
- Debian: `gcc-arm-none-eabi` for the compiler and `binutils-arm-none-eabi` for the utilities.
- Arch Linux: `arm-none-eabi-gcc` for the compiler and `arm-none-eabi-binutils` for the utilities.

This will install a version of the GNU C-compiler, which comes without any of the standard C-libraries functionalities, so no `printf` or `strcpy` are available, as an example. One could install the newlib-version of the packages, so a C-library would be available or build newlib from the sources and use that, when necessary. See Appendix A for a Makefile for building newlib from sources, with the above mentioned compiler.

The most simple program would therefore be a program, which toggles the LED of the Raspberry Pi on and off, to show, that the code is actually running. The ACT-LED on the Raspberry Pi is controlled by a GPIO-pin, i.e. it is controlled by using the GPIO-peripheral.

The hardware peripherals on the Raspberry Pi are located in the ARMs address space the base address 0x20000000 (at 0x3F000000 on Raspberry Pi 2 and 3). The GPIO peripheral is located at an offset of 0x00200000 from the base address. The register layout of the GPIO controller is described in the BCM2835 peripheral manual [4, pg. 89]. The C-structure shown in listing 2.1 represents the register layout. The named pad-fields are present, because there are padding 32-bit words in the register layout.

Listing 2.1: C-structure for mapping the GPIO-registers on BCM2835

```
1  /**
2   * \brief a mapping of the memory-registers of the GPIO-controller
3   **/
4  struct _pios_gpio_t
5  {
6      uint32_t fn_select[6];            ///< function select registers for the Pins
7      uint32_t pad1;
8      uint32_t outputset[2];            ///< registers for setting the levels HIGH
9      uint32_t pad2;
10     uint32_t outputclear[2];          ///< registers for setting the levels LOW
11     uint32_t pad3;
12     uint32_t level[2];                ///< the levels of the pins for reading
13     uint32_t pad4;
14     uint32_t eventDetect[2];          ///< indicates whether an event has occured (↩
           whathever is enabled)
15     uint32_t pad5;
16     uint32_t risingEdgeDetect[2];     ///< en/disable rising edge for the eventDetect-↩
           field
17     uint32_t pad6;
18     uint32_t fallingEdgeDetect[2];    ///< en/disable falling edge for the eventDetect
19     uint32_t pad7;
20     uint32_t highDetect[2];           ///< en/disable HIGH as event for eventDetect
21     uint32_t pad8;
22     uint32_t lowDetect[2];            ///< en/disable LOW as event for eventDetect
23     uint32_t pad9;
24     uint32_t asyncRisingEdgeDetect[2];   ///< en/disable rising edge as asynchronous event ↩
           for eventDetect (i.e. not bound to system clock)
25     uint32_t pad10;
26     uint32_t asyncFallingEdgeDetect[2];  ///< en/disable falling edge as asynchronous event ↩
           for eventDetect (i.e. not bound to system clock)
27     uint32_t pad11;
28     uint32_t pullControlEnable;       ///< set the pull-mode (i.e. UP, DOWN or OFF)
29     uint32_t pullControlClock[2];     ///< set pins for pulling (see the manual for the ↩
           exact algorithm)
30 } typedef pios_gpio_t;
```

To initialise a GPIO pin, first a function needs to be chosen. The function-registers consist of the function descriptors of ten pins per register. Every pin is represented with three bits, which can be set at will (but always read and written as complete register). A pin can be set to one of the following functions:

- Input (0b000)
- Output (0b001)
- ALT0 (0b100), ALT1 (0b101), ALT2 (0b110), ALT3 (0b111), ALT4 (0b011), ALT5 (0b010)

The output setting allows changing the voltage of that pin by the ARM, the input setting allows reading a value of the pin changed externally. The ALT-modes are used for other purposes and are highly pin-specific. Some pins might be usable for UART, other for JTAG [4, pg. 102]. Note,

that there is no possible configuration to use a pin simultaneously as input and output.

When a pin is set to output mode, its value can be set by writing a 1-bit at the corresponding bit in the `outputset`-registers, and can be clear by doing so in the `outputclear`-registers. For the sake of this example, it is assumed, that the target hardware is a Raspberry Pi Zero and the ACT LED is therefore represented by GPIO-pin 47 (on Raspberry Pi 1 it would be pin 16). The code shown in Appendix A switches the LED off and on again. For slowing down the switching process, it wastes processor time in between.

The linker script can also be found in Appendix A. It places the `.init`-section at the top of the binary and the text-section, normally, where the C-code is placed, after that. Putting the `.init`-section as the first section ensures, that the first instruction, which is executed is one of the init-section. There is only one routine in that section: main, which in fact is supposed to be the entry point of the program.

The major problem with this approach is, that the programmer needs to use inline assembler to change to stack-location in the C-code. Also the main-routine should never return, the the link register, which is used to determine the address of the last branch-instruction, contains garbage.

2.1.2 Test environment

It is tedious to copy the created image file onto the SD-card every time the programmer wants to test a new feature or tests a bug fix. Therefore the *raspbootin*-loader can be used, which consists of a program for a PC, which is connected to the Raspberry Pi hardware via a serial connection.

One important thing to note is, that it is not possible to use standard serial connectors of a PC, as they deliver larger voltages as the Pi requires and may damage the hardware. It is adviced to use a USB to serial chip, ones based on the PL2303-chip seem to work. Also some JTAG Hardware like the FT232H mini module also incorporates UART-functionality, which is compatible to the Raspberry Pi.

Figure 2.1 depicts the 40 GPIO-pins of a Raspberry Pi Zero (identical to other Raspberry Pi boards) [6,7]. The two UART connections are highlighted in orange, the JTAG-pins green. The Raspberry Pi can be powered through the 5V and GND-pins on the GPIO header by the USB-to-TTL Adapter. It is important to detach the power supply from the micro-USB port usually used for that.

When opting for a JTAG-chip a major advantage is, that it is possible to stop the processor from working and watch it executing instruction by instruction with the help of a debugger. The GDB and openOCD can work together to control the ARM1176jzf-s in a way to allow single stepping, but also execute line-by-line on either C or Assembler level.

With the UART set-up and the *raspbootin* kernel on the SD-card, the kernel development can begin. When testing a new build, the *raspbootcom* program on PC must be started, given the path to that image. Then the Raspberry Pi is started. It boots into raspbootin, sending certain character over the serial connection. *raspbootcom* waits for these control character and transmits the kernel, which is in turn written into a specific memory address on the Raspberry Pi. After the transmission the Raspberry Pi branches into the kernel.

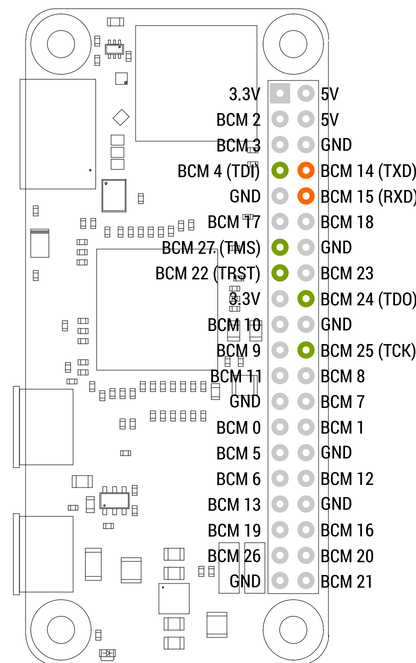

Figure 2.1: GPIO pinout for the Raspberry Pi (RPi Zero depicted)

Interrupts The registers of most hardware devices are mapped into the address space of the CPU [4]. There is one exception to that rule. The registers of the C15 coprocessor cannot be accessed by load and store instructions to memory addresses, but rather have to be accessed by special instructions (`msr`, `mrs`). Functions of this processor include enabling the Memory Managment Unit (MMU), activating instruction and data caches, enabling branch prediction or allowing the CPU to load and store unaligned data word or half-words [3].

The BCM2835 contains an interrupt controller [4, pg. 109]. The main CPU only has seven trap handlers. Several of them handle exceptions, i.e. problems that occur inside the CPU on program execution like an access to an unmapped section or an undefined instruction. The interrupt table is set up on the address 0x00000000 (of the ARMs address space). A possible assembler solution is depicted in Listing 2.2. The pointer `stack_top` points to the top of the stack to be set-up (compiled into the program).

Listing 2.2: Set-up the interrupt table on the Raspberry Pi

```
1  .section ".init"
2
3  irq_vector_start:
4  _start:
5      ldr pc, _reset_h
6      ldr pc, _undefined_instruction_vector_h
7      ldr pc, _software_interrupt_vector_h
8      ldr pc, _prefetch_abort_vector_h
9      ldr pc, _data_abort_vector_h
10     ldr pc, _unused_handler_h
11     ldr pc, _interrupt_vector_h
12     ldr pc, _fast_interrupt_vector_h
13
14 _reset_h:                           .word   _reset_
15 _undefined_instruction_vector_h:    .word   pios_exception_undef
16 _software_interrupt_vector_h:       .word   pios_exception_swi
17 _prefetch_abort_vector_h:           .word   pios_exception_abort
18 _data_abort_vector_h:               .word   pios_exception_abort
19 _unused_handler_h:                  .word   _reset_
```

```
20 _interrupt_vector_h:                .word   pios_exception_irq
21 _fast_interrupt_vector_h:           .word   pios_exception_fiq
22
23 _reset_:
24     ldr sp, =stack_top
25
26     /// copies interrupt vectors to address 0x0000 0000
27     mov     r0, #0x00
28     ldr     r1, =irq_vector_start
29     ldmia   r0!,{r2, r3, r4, r5, r6, r7, r8, r9}    /** load 32 Byte worth of data **/
30     stmia   r1!,{r2, r3, r4, r5, r6, r7, r8, r9}    /** store 32 Byte **/
31     ldmia   r0!,{r2, r3, r4, r5, r6, r7, r8, r9}
32     stmia   r1!,{r2, r3, r4, r5, r6, r7, r8, r9}
```

The ARM1176jfz-s uses a bit in the program status register `cpsr` to indicate whether interrupts and fast interrupts are masked (i.e. disabled) or not. When they are not disabled and an interrupt occurs, then the processor will jump to the corresponding code in the interrupt table, therefore the code in 0x30 (every entry in the table spans 8 byte or up to two instructions). There a jump will be executed and the processor jumps to the correct routine to handle the interrupt.

The routine would need to store the registers of the processor and put the link register on the stack for being able to restore the context of the preempted process. If the stack needs to be changed, the stack register also needs to be stored in a globally known variable. After the interrupt has been handled, the process' context needs to be restored and a return from interrupt instruction issued.

Interrupt Controller The Raspberry Pi uses an "external" interrupt controller, which indicates, which device issued an interrupt. The Interrupt Service Routine (ISR) can query its registers and find clues, what devices issued an interrupt, and therefore which devices (or device drivers) needs checking next. The interrupt controllers registers can be found mapped into memory with a base address of 0xB200, from the Input/Output (I/O) base address (which is 0x20000000 on Raspberry Pi 1 and 0). Listing 2.3 shows a struct definition for the memory mapped registers.

Listing 2.3: Registers of the interrupt controller of the Raspberry Pi

```
1  /**
2   * \brief mapping out the memory mapped registers of the interrupt controller of the BCM2835↩
       -chip
3   **/
4  struct _pios_irq_controller_t
5  {
6      uint32_t basic_pending;     ///< are basic interrupts pending?
7      uint32_t pending[2];        ///< are interrupts pending
8      uint32_t fiq_control;       ///< control register for fast interrupts
9      uint32_t enable[2];         ///< enable interrupts
10     uint32_t basic_enable;      ///< enable basic interrupts
11     uint32_t disable[2];        ///< disable interrupts
12     uint32_t disable_basic;     ///< disable basic interrupts
13 } typedef pios_irq_controller_t;
```

The interrupt controller allows masking and unmasking interrupt sources with the disable and enable-registers. The controller has 64 interrupt sources or lines, which can be triggered, but only some of them are used. Some of these lines are brought to the basic-registers, like `basic_pending`. `basic_pending` shows besides some of the 64 interrupt sources also the pending status of some special interrupts. For example it shows, whether an clock-interrupt is pending (value $1 << 0$) or whether the first 32 or the last 32 interrupt lines are raised [4, pg. 112].

3 Porting Micropython and running code

The Micropython description in this Section is based on tag v1.9.3 [5]. This chapter describes the creation of a new port of the Micropython code base. At first a port was attempted by using the newlib C-subroutine library. Later the LibC of Micropython was used, which is described here. Building newlib and incorporating it into a project is described in Appendix B.

The aim for porting Micropython is to build a kernel image, which can be put onto the SD-Card of a standard Raspberry Pi. The Raspberry Pi should be able to boot this kernel and run the code therein, in turn it should be able to run Python code with the help of the Micropython interpreter without the need of another operating system underneath.

3.1 Port structure

For porting Micropython to the Raspberry Pi first one has to make sure, that it compiles and loads, then more advanced features like compiling, telinking and running code can be tested. Before beginning work on the porting work, the structure of a port-folder in Micropython has to be clear. The following files can be found in a (normal) port:

- `Makefile` - contains rules for building the port. Will load `py/mkenv.mk`, `py/mkrules.mk` and `py/py.mk`
- `mphalport.h` - contains specific declarations for that port; empty in the *bare-arm* port. Is used in `py/mphal.h`, to check whether some I/O functions are set
- `mpconfigport.h` - describes how and with what feature set Micropython is built. This is the primary place to disable or enable features or builtin modules
- `qstrdefsport.h` - defines strings, which are used by the port to be builtin and known by the interpreter
- `kernel.ld` - a linker-script (can be called differently)

Most ports will also contain C-code and header files, which implement the functionality for this specific board, specially drivers and modules for the Python language to use.

The QStrings have to be defined in the `qstrdefsport.h`-file because Micropython tries to eliminate duplicates and builds the strings only once into the image. QStrings will be added at run-time for example, when a new function is created, a new variable is declared or a class defined.

3.2 Understanding the build-system

The *bare-arm* port can be used as starting point, as the settings are already usable to create a build for an ARM Cortex-M4 processor. The Raspberry Pi uses an ARM1176jzf-s processor. This first build is not used for any testing of the code, but rather to understand the build-system

of Micropython.

For simplicity the image shall only contain ARMv6 code, not Thumb-code. Thumb is an extension of the instruction set of the ARM processors, which contains 16 bit instructions. The ARM processor needs to change the internal state to execute the Thumb instructions and cannot execute regular ARM instructions when in this state. The transition between these states should be possible by the ARM-branch instruction `bx` (branch and exchange instruction set), which toggles the Thumb-bit in the program status register (cpsr) of the ARM processor.

An important thing to note is, that with that configuration the linking process will likely fail. The error messages state, it cannot find code for labels like `__aeabi_uidivmod` or `__aeabi_idiv`. That means, the linker cannot find code for the division of integers. The GCC probably hides integer division like that for abstraction purposes as not all CPUs can divide integers in hardware, for example the ARM1176jzf-s cannot. GCC brings its own static library which contains these functions, the libgcc. This library is located in the compilers directory, on a Linux-syste, this could be `/usr/lib/gcc/arm-none-eabi/7.0.0/`.

3.2.1 The Makefile

The Makefile makes use for several Makefiles with rules and variables, like `py/mkenv.mk`, `py/mkrules.mk` and `py/py.mk`. These files contain basically all rules that are necessary to build Micropython itself. The Makefile of the port is there for adding files specific to that port, using the right parameters of the compiler and using the right compiler of course. Listing 3.1 shows the Makefile for the Raspberry Pi port at this point.

Listing 3.1: First Makefile of the Raspberry Pi port

```
1  include ../../py/mkenv.mk
2
3  # qstr definitions (must come before including py.mk)
4  QSTR_DEFS = qstrdefsport.h
5
6  # include py core make definitions
7  include $(TOP)/py/py.mk
8
9  CROSS_COMPILE = arm-none-eabi-
10
11 OBJCPY=$(CROSS_COMPILE)objcopy
12 INC += -I.
13 INC += -I$(TOP)
14 INC += -I$(BUILD)
15
16 CFLAGS = $(INC) -Wall -std=c99 -nostdlib -marm -mcpu=arm1176jzf-s $(COPT)
17
18 #Debugging/Optimization
19 ifeq ($(DEBUG), 1)
20 CFLAGS += -O0 -ggdb
21 else
22 CFLAGS += -Os -DNDEBUG
23 endif
24
25 CCVERSION:=$(shell $(CC) --version | sed 's/[ ]/\n/g' | grep "\." | xargs | awk '{ print $$1↩
       }')
26 ARMGCCLIBPATH=/usr/lib/gcc/arm-none-eabi/$(CCVERSION)
27
28 LDFLAGS = -nostdlib -T kernel.ld -Map=$@.map --cref
29 LIBS = -L$(ARMGCCLIBPATH) -lgcc
30
31 SRC_LIB = $(addprefix lib/,\
32     libc/string0.c )
33
34 SRC_C = \
```

```
35     main.c
36
37 SRC_S = \
38     start.s
39
40 OBJ = $(PY_O) $(addprefix $(BUILD)/, $(SRC_C:.c=.o) $(SRC_S:.s=.o) $(SRC_LIB:.c=.o))
41
42 all: $(BUILD)/firmware.img
43
44 list: $(BUILD)/firmware.elf
45     $(CROSS_COMPILE)objdump -d $(BUILD)/firmware.elf > $(BUILD)/firmware.list
46
47 $(BUILD)/firmware.elf: $(OBJ)
48     $(ECHO) "LINK $@"
49     $(Q)$(LD) $(LDFLAGS) -o $@ $^ $(LIBS)
50     $(Q)$(SIZE) $@
51
52 $(BUILD)/firmware.img: $(BUILD)/firmware.elf
53     $(OBJCPY) $(BUILD)/firmware.elf -O binary $(BUILD)/firmware.img
54
55 include $(TOP)/py/mkrules.mk
```

The variable `CROSS_COMPILE` contains the prefix for the gcc-compiler for the specific platform. A cross compiler is used for generating machine code from a higher languages, where the generated machine code is not the one understood by the host processor, running the compiler. This port is intended to be built on an x86-PC, so the cross compiler `arm-none-eabi` is used. The prefix is a triple, which identified the compiler. The first value is the platform, in this case arm, the second indicates the vendor, which is mostly unused. The third value names the operating system, which identifies the syscall-abi. As the vendor field is mostly unused it can be left out.

Line 16 contains the definition of the `CFLAGS`-variable. It contains the flags and parameters, which will be passed to the gcc-call. In this case specifying, where it can find header-files (in variable `INC`), that it should report all messages, the C-version (C99 in this case), that it should not use any standard library, that is should produce ARM-code (in contrary to `-mthumb`) and that is should produce code for the ARM1176jzf-s processor. `COPT` is a variable set by the Micropython Makefiles.

Line 19 to 23 are there for deciding whether to build the code in Debug mode or not; enabling debug-mode will prevent the compiler from making any optimisations. The created image will therefore be huge and slow, but will be debuggable with GDB, whereas the non-debug version is optimised for binary-size. This makes single-stepping the code a bit harder, as a single step may jump over several C-instructions or the compiler could have reordered the code. Lines 25 and 29 try to guess the compiler-version number and the location of the libgcc library to link to.

`SRC_C`, `SRC_LIB` and `SRC_S` are variables, which need to be set with the C-files, the library-C-files and the Assembler-files intricate to that specific port. These can be modules for Micropython, but also code to be called before handing the control to Python-code. Line 40 combines the three variables, prepending the build-directory in front of the filenames and replacing the filename extensions, with an „.o" for further usage in Makefile-rules.

The *list*-rule generates a code-listing from the built binary, the *firmware.elf* rule builds the `firmware.elf`-file, which is used to generate an image to be put onto the SD-card of the Raspberry Pi. The last statement includes the creation rules of Micropython.

3.2.2 The code

This first image is not supposed to run actual Micropython code but rather to serve as prove of concept that the build-system works, and that it is capable of creating working Raspberry Pi images. It consists of two addition files to the Micropython ones: a *start.s* Assembler-file and a *main.c* C-file. The Assembler file serves the purpose of switching the stack and jumping into the main C-code. The main-code initialises the UART-chip, prints a string over that UART-connection (using `printf`) and hangs in an infinite loop.

The first problem is to convince the Micropython code to give any sort of output. Micropython has an internal implementation of `printf` (in *lib/libc/string0.c*), which performs all the necessary operations to print strings in theory, but it uses the macro `MP_PLAT_PRINT_STRN(str, len)` as a placeholder for a `write`-like function. This macro is set in the default *bare-arm*-configuration to a null-function, i.e. has no functionality.

For the output of characters a driver for an UART-chip of the Raspberry Pi is needed. In the interest of space, a full implementation is skipped here, but [10, file `source/uart.c`] contains one. It is assumed that the driver provides a function like `putchar` and allows for creating a `write`-like functionality.

Note: for using the UART implementation of [10], the files *gpio.h*, *gpio.c*, *uart.h* and *uart.c* need to be copied, the include-paths changed and preferably in *gpio.h* there has to be the preprocessor constant `PIOS_IO_BASE`.

This code will also not be linkable, as the compiler cannot find the code for `printf` and `puts` at this moment. The solution for that problem lies in the file *mpconfigport.h*. When switching the constant `MICROPY_USE_INTERNAL_PRINTF` to a 1, the compiler will use the `printf`-code of Micropython. However this code includes a quirk, that is uses `mp_hal_stdout_tx_strn_cooked` instead of the output-macro from above. This function is basically identical to that macro and can also be set to the `write`-function of the UART-driver.

3.3 Starting Micropython

To start Micropython the infinite loop can be removed. This will make the code after that, which was already in the file reachable by the processor. This code will initialise Micropython (`mp_init`) and parse, lex and compile two Python statements (`do_string`). This paper will not go into detail about the intricacies of parsing, lexing and compiling Python code in Micropython.

Micropython normally makes use of a garbage collected heap to manage its objects. The garbage collector can be enabled or disables in the *mpconfigport.h*-file. At this point it would be advisable to enable the garbage collector, so Micropython can use the heap to allocate objects. If the garbage collection is not enabled, the code should also work, but it is unclear where Micropython allocates new objects[1]. To enable the garbage collector, the constant `MICROPY_ENABLE_GC` must be set to 1.

Then the main code can initialise the garbage collector by calling `gc_init` with the beginning and end-address of the heap. These addresses can be determined by the linker, by setting vari-

[1] I suspect the stack

ables in the linker script. These variables can be placed at a known distance for example to the end of the binary, this also can be done to the stack.

Enabling the garbage collector will however make it necessary to have a `gc_collect`-function, which will be called each time the heap is full, but a new object needs to be allocated. It is supposed to clean up the heap, remove dead objects and place the alive objects at the beginning of the heap. Allocation is then very easy, as the allocator can look for the last used object in the heap and place the new one directly behind.

For the time being the `gc_collect`-function can be left blank. It should however be noted, the the heap should then never be full. Writing a collector is not a trivial objective and bugs in the collector have catastrophic impact on the stability and behaviour of the system.

The constant `MICROPY_DEBUG_VERBOSE` allows compiling Micropython very talkative. Micropython will then output loads of data, many concerning the garbage collector and its state. The C-variable `mp_verbose_flag` is used to signal Micropython, whether it should print out the bytecode representation of the Python-code once compiled. It is therefore set to 2 for testing at this stage.

In this point in time the port has some weird stability issues. Most importantly it crashes (or jumps outside the constraints of the Bytecode) when executing Python code like the one shown in Listing 3.2.

Listing 3.2: Simple for test-loop

```
1 for i in range(10):
2     print(i)
```

For further inspection the generated Bytecode is output. The emitted wrong Bytecode from the Raspberry Pi port is shown in Listing 3.3 and the Bytecode, which was expected in 3.4. Note, that the Bytecode itself is actually correct, but rather the Micropython interpretation of that is incorrect.

Listing 3.3: The wrong Bytecode generated by the Raspberry Pi port initially

```
1  File <stdin>, code block '<module>' (descriptor: 66fb0, bytecode @67040 41 bytes)
2  Raw bytecode (code_info_size=6, bytecode_size=35):
3   03 00 00 00 00 00 06 09 10 29 00 00 ff 80 35 0f
4   80 30 24 82 25 1c 81 6f 1c 82 25 64 01 32 81 e9
5   30 8a f0 36 eb 7f 32 11 5b
6  arg names:
7  (N_STATE 3)
8  (N_EXC_STACK 0)
9    bc=-1 line=1
10   bc=8 line=2
11 00 LOAD_CONST_SMALL_INT 0
12 01 JUMP 4294938425
13 04 DUP_TOP
14 05 STORE_NAME i
15 08 LOAD_NAME print
16 11 LOAD_NAME i
17 14 CALL_FUNCTION n=1 nkw=0
18 16 POP_TOP
19 17 LOAD_CONST_SMALL_INT 1
20 18 BINARY_OP 18
21 19 DUP_TOP
22 20 LOAD_CONST_SMALL_INT 10
23 21 BINARY_OP 25 __lt__
24 22 POP_JUMP_IF_TRUE 4
25 25 POP_TOP
26 26 LOAD_CONST_NONE
27 27 RETURN_VALUE
```

Listing 3.4: The correct Bytecode generated by the QEMU port, see Appendix C

```
File <stdin>, code block '<module>' (descriptor: 6efa0, bytecode @6f030 41 bytes)
Raw bytecode (code_info_size=6, bytecode_size=35):
 03 00 00 00 00 00 06 09 10 29 00 00 ff 80 35 0f
 80 30 24 82 25 1c 81 6f 1c 82 25 64 01 32 81 e9
 30 8a f0 36 eb 7f 32 11 5b
arg names:
(N_STATE 3)
(N_EXC_STACK 0)
  bc=-1 line=1
  bc=8 line=2
00 LOAD_CONST_SMALL_INT 0
01 JUMP 19
04 DUP_TOP
05 STORE_NAME i
08 LOAD_NAME print
11 LOAD_NAME i
14 CALL_FUNCTION n=1 nkw=0
16 POP_TOP
17 LOAD_CONST_SMALL_INT 1
18 BINARY_OP 18
19 DUP_TOP
20 LOAD_CONST_SMALL_INT 10
21 BINARY_OP 25 __lt__
22 POP_JUMP_IF_TRUE 4
25 POP_TOP
26 LOAD_CONST_NONE
27 RETURN_VALUE
```

Note, that the JUMP-label on line 01 is incorrect in the Raspberry Pi port. This is quite problematic as the Bytecode interpreter will jump to that line - or try to and land somewhere in memory and try to execute that code.

George [8] pointed out, that this is due to a value-rotation issue of the ARM1176jzf-s processor. In the mode it was used here, it cannot access unaligned data, that means a 16 bit half-word has to be aligned to a 2-byte address (the last number in binary notation has to be 0), a 32 bit word to a 4-byte address. The compiler seems to use unsafe optimisations to generate code, which loads a 16-bit value instead of two 8-bit ones.

Trying to fix the issue by forcing the compiler to not optimise the code, which decodes Python Bytecode was a first solution. This worked for the printed representation of the Bytecode, but the code was not executed, probably due to another alignment problem in *py/emitglue.c*.

The ARM1176jzf-s processor can load unaligned values in two main ways. It can either try to load the unaligned data, trap into the operating system kernel and load the data in software (i.e. analysing the problem, which occurred, figuring out the correct address(es) to load, and loading them byte-wise). On the other hand it can load unaligned value by itself, when the option is switch on in the C15-co-processor [1,2]. The registers of this co-processor can also be accessed only by the dedicated instructions `mrc` and `mcr`. The bit 22 (U-Bit) switches the ability to load unaligned data on or off. Listing 3.5 shows the necessary code to set this bit to 1, added in the start-up assembler code.

Listing 3.5: Setting the U-Bit in the C15-co-processor to allow the ARM-processor to load unaligned data

```
/// load C15 control register c1-c1,0,0 (c1 Control Register) into r0
mrc p15, 0, r0, c1, c0, 0
ldr r1, =0x00400000     /// set bit 22 (U-bit)
orr r0, r1
mcr p15, 0, r0, c1, c0, 0
```

4 Extending Micropython

This chapter explains the basics of module in Micropython and gives an insight on how to extend Micropython with new modules. A Micropython module is a set of functions and classes, which can be used from the Python language level. These modules may be written as imported files in Python or in C by using certain data structures to inform Micropython about its new module. This allows adding C-code to the Python programs, so time-critical functionality may be implemented, C-data structures modified directly or the hardware may be accessed directly.

The second part of this chapter then looks into the Micropython way of implementing interrupt service routines in Python. A interrupt may occur at any point in time, even when a memory operation on the Python-heap is not finished or a Python-instruction is not finished. Section 4.2 describes the Micropython solution to these problems and also to enabling interoperability between the Python and C-levels.

4.1 Extending Micropython

First this section looks into extending Micropython by adding modules to it. This allows Micropython to call user-defined C-routines, when using the corresponding Python-routine from Python code. This makes the Python code more powerful, as it allows to access memory addresses directly, as will be shown. This section will set-up a class for raw-pointer access to memory locations.

4.1.1 Creating and adding an empty module

Before adding a class to a module, an (empty) module is needed. A module basically serves as container for functions and classes for Micropython. When using the `import` statement in Python, Micropython uses its builtin import-function to locate a module, which corresponds to that name, loads it into main memory if necessary, and makes the global names of that module visible to the calling module.

A module is an object for Micropython. In a C-file a declaration like the one shown in Listing 4.1 could be used to declare a module-structure.

Listing 4.1: Defining an empty module

```
1  #include "py/mpconfig.h"
2  #include "py/nlr.h"
3  #include "py/misc.h"
4  #include "py/qstr.h"
5  #include "py/obj.h"
6  #include "py/runtime.h"
7
8  // create a table of global strings, here the only one is name
9  STATIC const mp_map_elem_t global_table[] =
10 {
11     { MP_OBJ_NEW_QSTR(MP_QSTR___name__), MP_OBJ_NEW_QSTR(MP_QSTR_cptr) },
12 };
13
14 // create a dictionary of strings from the table
```

```
15 STATIC MP_DEFINE_CONST_DICT(cptr_globals, global_table);
16
17 // fill the mp_obj_module_t struct, which defines the module itself
18 const mp_obj_module_t cptr =
19 {
20     .base = { &mp_type_module },
21     .globals = (mp_obj_dict_t*)&cptr_globals,
22 };
```

Line 18 to 22 define the module structure - as with all objects in Python it has a base, identifying the base-class for inheritance-purposes. Also it has a globals-field, which contains a dictionary of all global names linked with their C-representation. This dictionary is the essential part, as it on the one hand contains all Python-names, on the other hand it stores the object to associate with that name on the C-level.

A dictionary is infact also an object with a base, but it has an `mp_map_t` field in addition [5, py/obj.h l.764]. A map is in fact not an object in Micropython, but represented as a struct; as shown in Listing 4.2 [5, py/obj.h l.341 and l.356].

Listing 4.2: Type definition of `mp_map_t`

```
1  typedef struct _mp_map_elem_t {
2      mp_obj_t key;
3      mp_obj_t value;
4  } mp_map_elem_t;
5
6  typedef struct _mp_map_t {
7      size_t all_keys_are_qstrs : 1;
8      size_t is_fixed : 1;    // a fixed array that can't be modified; must also be ordered
9      size_t is_ordered : 1;  // an ordered array
10     size_t used : (8 * sizeof(size_t) - 3);
11     size_t alloc;
12     mp_map_elem_t *table;
13 } mp_map_t;
```

A map consists of several flags, which are unimportant for the moment, but also an array of `mp_map_elem_t`. These structures contain a key and a value-object, so in turn the key and the value could be of any Python type. An object in Micropython is represented by a void-pointer, so behind that pointer could by any structure. The type definition for `mp_obj_t` is listed in Listing 4.3.

Listing 4.3: Type definition of `mp_obj_t`

```
1  // This is the definition of the opaque MicroPython object type.
2  // All concrete objects have an encoding within this type and the
3  // particular encoding is specified by MICROPY_OBJ_REPR.
4  #if MICROPY_OBJ_REPR == MICROPY_OBJ_REPR_D
5  typedef uint64_t mp_obj_t;
6  typedef uint64_t mp_const_obj_t;
7  #else
8  typedef void *mp_obj_t;
9  typedef const void *mp_const_obj_t;
10 #endif
```

Lines 9 to 12 in Listing 4.1 show the definition of a map-structure, which contains one element, the key value is an object containing the QString "__name__" and the value is the QString "cptr". Line 15 uses a macro to define a dictionary with the name "cptr_globals" from the previously defined map of global strings. The dectionary is then uses as the dictionary of the globally visible names in the module.

For informing Micropython that there is a new module two things need to be done. First the missing QStrings, in this case ony the module name "cptr" need to be defined. The port had a file called *qstrdefsport.h*. The QString can be added there, by appending a line `Q(qptr)`. When

building Micropython all QStrings are collected and preprocessor constants are formed like MP_QSTR_qptr. These can then be used in the code.

Secondly the module needs to be known to the import routine of Micropython. The file *mpconfigport.h* contains a preprocessor constant MICROPY_PORT_BUILTIN_MODULES, which is a map of names and their respective module structure. Listing 4.4 shows the map the previously shown empty module.

Listing 4.4: Inform Micropython about its new builtin module (in *mpconfigport.h*

```
1 extern const struct _mp_obj_module_t cptr;
2 #define MICROPY_PORT_BUILTIN_MODULES
3        { MP_OBJ_NEW_QSTR(MP_QSTR_cptr),  (mp_obj_t)&cptr }
```

The module can then be imported in Python code by using import cptr.

4.1.2 Functions

For adding a function to a Micropython module, a C-function has to be defined. As writing a class for raw-pointer memory access is the goal, Listing 4.5 shows a C-function printing an integer (the constant 42, which will be replaced later).

Listing 4.5: First function, prints out an integer number (as pointer)

```
1 #include <stdio.h>
2
3 STATIC mp_obj_t rawptr_print(void)
4 {
5     printf("RawPtr-value: %p\n",  (void*)42);
6     return mp_const_none;
7 }
```

The function returns an mp_obj_t. It is not possible to have a function, which does not return anything. If the programmer needs a function without a (usable) return value, she has to return the None, an object in Python indicating that the value does not exist (i.e. no return value, etc). This can be done by the macro mp_const_none.

Micropython can't use the C-function itself. So, a function object has to be created. Micropython has macros for that, the programmer however needs use the correct macro for the number of parameters of her function. Then the function can be added to the table of global objects with a name. Listing 4.6 shows the change of the global table.

Listing 4.6: Adding the function object to the table of global objects

```
1 // create a function object from the C-function
2 STATIC MP_DEFINE_CONST_FUN_OBJ_0(rawptr_print_obj, rawptr_print);
3
4 STATIC const mp_map_elem_t global_table[] =
5 {
6     { MP_OBJ_NEW_QSTR(MP_QSTR___name__),        MP_OBJ_NEW_QSTR(MP_QSTR_cptr)   },
7     // add the function object to the globals table
8     { MP_OBJ_NEW_QSTR(MP_QSTR_rawptr_print),    MP_OBJ_NEW_QSTR(MP_QSTR_rawptr_print_obj) }
9 };
```

Function arguments also have the type mp_obj_t. Listing 4.7 lists the same procedure for a function, which takes one argument, in this case a number to print.

Listing 4.7: Adding the function with one argument

```
1 STATIC mp_obj_t rawptr_setAddr ( mp_obj_t addr )
2 {
3     printf("The new pointer value: %p!\n", (void*) mp_obj_get_int( addr );
4     return mp_const_none;
```

```
5 }
6
7 STATIC MP_DEFINE_CONST_FUN_OBJ_1(rawptr_set_obj, rawptr_set);
```

4.1.3 Classes

A class is like a module defined by filling out the correct struct and attaching it to the globals table of the module. Defining a class is like defining a type in Python. The programmer has to specify, what operations (methods) are possible with this type. This is done by a list of function objects, which are local to that type.

In Python a class has always a method `print`, which outputs the class in some way on the standard output, and a method `new`, which is the constructor of that object. Also the type has to have a name. Listing 4.8 shows a simple raw-pointer class with the functions from above used as methods. Note, that they moved from the global table to the local table of the `rawptr_type`.

Listing 4.8: Outline of a simple raw-pointer class

```
1  typedef struct _rawptr_t
2  {
3      mp_obj_base_t base;
4      void* address;
5  } rawptr_t;
6
7  extern const mp_obj_type_t rawptr_type;
8
9  // constructor of the rawptr-class
10 mp_obj_t rawptr_new ( const mp_obj_type_t *type,
11                       size_t n_args,
12                       size_t n_kw,
13                       const mp_obj_t *args )
14 {
15     // test if number of arguments is exactly one (?) - on error: raise exception
16     mp_arg_check_num(n_args, n_kw, 1, 1, true);
17     // I want an object of my class, right?
18     rawptr_t *self = m_new_obj(rawptr_t);
19     // set the type
20     self->base.type = &rawptr_type;
21     // set the parameter
22     self->address = (void*) mp_obj_get_int(args[0]);
23     return MP_OBJ_FROM_PTR(self);
24 }
25
26 // print method of the rawptr-class
27 STATIC void rawptr_print ( const mp_print_t *print,
28                            mp_obj_t self_in,
29                            mp_print_kind_t kind )
30 {
31     rawptr_t *self = MP_OBJ_TO_PTR(self_in);
32     printf("RawPtr-value: %p\n", (void*)self->address );
33 }
34
35 // setter of the rawptr-class
36 STATIC mp_obj_t rawptr_setAddr ( mp_obj_t self_in,
37                                  mp_obj_t addr )
38 {
39     rawptr_t* self = MP_OBJ_TO_PTR ( self_in );
40     self->address = (void*) mp_obj_get_int( addr );
41     printf("The new pointer value: %p!\n", (void*) mp_obj_get_int( addr );
42     return mp_const_none;
43 }
44 // create a function object for the setter
45 MP_DEFINE_CONST_FUN_OBJ_2 ( rawptr_setAddr_obj, rawptr_setAddr );
46
47 // the local names and functions for the rawptr-type
```

```
48 STATIC const mp_map_elem_t rawptr_locals_dict_table[] =
49 {
50     { MP_ROM_QSTR(MP_QSTR_setAddr), MP_ROM_PTR(&rawptr_setAddr_obj) },
51 };
52 STATIC MP_DEFINE_CONST_DICT ( rawptr_locals_dict, rawptr_locals_dict_table );
53 
54 // defining the rawptr-type; the name for using the type is given by the
55 // QString in the name-field
56 const mp_obj_type_t rawptr_type =
57 {
58     { &mp_type_type },
59     .name = MP_QSTR_RawPtr,
60     .print = rawptr_print,
61     .make_new = rawptr_new,
62     .locals_dict = (mp_obj_dict_t*)&rawptr_locals_dict,
63 };
64 
65 // the updated global table for the cptr-module
66 STATIC const mp_map_elem_t globals_table[] =
67 {
68     { MP_OBJ_NEW_QSTR(MP_QSTR_RawPtr), (mp_obj_t)&rawptr_type }
69 };
70 
71 // create a dictionary of strings from the table
72 STATIC MP_DEFINE_CONST_DICT(cptr_globals, global_table);
73 
74 // fill the mp_obj_module_t struct, which defines the module itself
75 const mp_obj_module_t cptr =
76 {
77     .base = { &mp_type_module },
78     .globals = (mp_obj_dict_t*)&cptr_globals,
79 };
```

The RawPtr-class is represented by two structures. The first one is the struct `_rawptr_t`, which is the inner class and contains the attributes of the class for the methods fot work with (lines 1 to 5). The second is of type `mp_obj_type_t` and is used to inform Micropython about the type, i.e. what is its `new`, `print` and methods.

The `new`-method of the RawPtr-class has 4 parameters. The second and fourth are of interest here, as the second identifies the number of arguments provided, and the fouth and last argument is a list of arguments. Line 16 checks the number of parameters.

Line 18 to 23 are the interesting lines of code here. A new instance of the inner object is created, the type is set to a pointer to the type-definition. This is important, as whenever Micropython encounters a method-call to the object, which will be returned on line 23, it will look into the list of methods in the local dictionary of that type definition. Line 22 sets the initial address of the pointer, and line 23 returns the inner object as new instance of the class.

Micropython needs to be able to know, where the type-information is stored. It will, when it encounters the object, not know how the inner structure is set-up. It will basically act blindly and hopes, that the base-field is the first one, speculating to find the type-pointer on a certain point in the structure.

The attentive reader may have noticed, that the prototype of the print and setter-method has changed from above. The `rawptr_setAddr`-method has gotten a `self`-argument, which is known already to Python programmers. This argument allows access to the attributes of the object.

The print-method has also received additional arguments. The first argument is of type `mp_print_t`, the code of which is shows in Listing 4.9 [5, *py/mpprint.h* l.48].

Listing 4.9: Type definition of `mp_print_t`

```
typedef void (*mp_print_strn_t)(void *data, const char *str, size_t len);

typedef struct _mp_print_t {
    void *data;
    mp_print_strn_t print_strn;
} mp_print_t;
```

The second argument is the `self`-reference and the last is of type `mp_print_kind_t`, which specifies the kind of print-functionality to be executed. This allows specifying modifiers to the print-statement. The possible values for that are listed in Listing 4.10.

Listing 4.10: Enum definition of `mp_print_kind_t`

```
typedef enum {
    PRINT_STR = 0,
    PRINT_REPR = 1,
    PRINT_EXC = 2, // Special format for printing exception in unhandled exception message
    PRINT_JSON = 3,
    PRINT_RAW = 4, // Special format for printing bytes as an undercorated string
    PRINT_EXC_SUBCLASS = 0x80, // Internal flag for printing exception subclasses
} mp_print_kind_t;
```

4.1.4 Exceptions

Python code - and in turn modules in Python - may raise an exception when unexpected or errnous things occur, instead of returning an integer value, that represents an error-state like in C. In the self-defined module, the programmer can use either `mp_raise_msg` or `nlr_raise` directly to raise an exception. Listing 4.11 shows `mp_raise_msg`.

Listing 4.11: Raising an exception

```
NORETURN void mp_raise_msg(const mp_obj_type_t *exc_type, const char *msg) {
    if (msg == NULL) {
        nlr_raise(mp_obj_new_exception(exc_type));
    } else {
        nlr_raise(mp_obj_new_exception_msg(exc_type, msg));
    }
}
```

4.1.5 Conclusion

The complete and tested version of the RawPtr-class is depicted in Appendix D. But there is a caveat, as both of the implementations are based on the assumption, that Micropython chooses the same length of the integer as the pointers are on the platform. In other words: a pointer might be smaller than an integer, the integer object may in fact not contain a value representable by a single integer in C, but rather a larger number. This has not been tested yet, and may lead to unexpected results.

4.2 Interrupt Handling

Micropython tackled a lot of the problems writing drivers and interrupt service routines already, so it is worth looking into how Micropython overcame some of these problems. This section is based on the `stm32`-port of the Micropython interpreter [5] and will use the Timer interrupt as an example on how interrupt service routines work in Micropython. All files reside in the folder `ports/stm32`.

Micropython allows writing "interrupt service routines" by the means of registering a callback function for example with the Timer-class in Python itself. A programmer can then write a function which is executed, when the timer interrupt occurs. The Timer-class is written entirely in C and defined as a class in the module `pyb`. Listing 4.12 shows the code of the callback-method of Timer-class [5, file timer.c, line 1155 to 1178].

Listing 4.12: timer.callback method for registering a timer interrupt

```
/// \method callback(fun)
/// Set the function to be called when the timer triggers.
/// `fun` is passed 1 argument, the timer object.
/// If `fun` is `None` then the callback will be disabled.
STATIC mp_obj_t pyb_timer_callback(mp_obj_t self_in, mp_obj_t callback) {
    pyb_timer_obj_t *self = self_in;
    if (callback == mp_const_none) {
        // stop interrupt (but not timer)
        __HAL_TIM_DISABLE_IT(&self->tim, TIM_IT_UPDATE);
        self->callback = mp_const_none;
    } else if (mp_obj_is_callable(callback)) {
        __HAL_TIM_DISABLE_IT(&self->tim, TIM_IT_UPDATE);
        self->callback = callback;
        // start timer, so that it interrupts on overflow, but clear any
        // pending interrupts which may have been set by initializing it.
        __HAL_TIM_CLEAR_FLAG(&self->tim, TIM_IT_UPDATE);
        HAL_TIM_Base_Start_IT(&self->tim); // This will re-enable the IRQ
        HAL_NVIC_EnableIRQ(self->irqn);
    } else {
        mp_raise_ValueError("callback must be None or a callable object");
    }
    return mp_const_none;
}
STATIC MP_DEFINE_CONST_FUN_OBJ_2(pyb_timer_callback_obj, pyb_timer_callback);
```

The method takes two parameters, the first one is the `self`-object, a reference to the class-instance, the called method is associated with, the second one is a function object. The method distinguishes three possible scenarios: a None-object is passed as function object, then the interrupt will be disabled, a callable function object is passed, then this will be set as timer-interrupt handler and the timer will be started, or something different is passed, then an exception is raised and nothing will be changed. On setting any callback function (or unsetting them) interrupts are disabled using the `__HAL_TIM_DISABLE_IT`-macro, so no interrupt occurs while setting or unsetting the callback function, as this could lead to disastrous behaviour.

The table of interrupt vector is set in [5, file startup_stm32.S]; there the label `TIM1_CC_IRQHandler` can be found. Following this label will lead to code like the one depicted in Listing 4.13. The `IRQ_Enter` and `IRQ_Exit` macros increase and decrease the nested interrupt counter, similar to the one found in Linux (later in this chapter).

Listing 4.13: Timer 1 interrupt handler

```
void TIM1_CC_IRQHandler(void) {
    IRQ_ENTER(TIM1_CC_IRQn);
    timer_irq_handler(1);
    IRQ_EXIT(TIM1_CC_IRQn);
}
```

The function `timer_irq_handler` is defined in [5, timer.c]. It looks up the correct timer for the requested interrupt by using the `MP_STATE_PORT`-macro and calls the callback using the `timer_handle_irq_channel`. This function will lock the Garbage Collector to disallow any memory allocations and catches all exceptions thrown. If there where any exceptions, the callback function will be disabled. Listing 4.14 shows the code of these functions.

Listing 4.14: Timer interrupt handler

```
STATIC void timer_handle_irq_channel(pyb_timer_obj_t *tim, uint8_t channel, mp_obj_t callback) {
    uint32_t irq_mask = TIMER_IRQ_MASK(channel);

    if (__HAL_TIM_GET_FLAG(&tim->tim, irq_mask) != RESET) {
        if (__HAL_TIM_GET_ITSTATUS(&tim->tim, irq_mask) != RESET) {
            // clear the interrupt
            __HAL_TIM_CLEAR_IT(&tim->tim, irq_mask);

            // execute callback if it's set
            if (callback != mp_const_none) {
                mp_sched_lock();
                // When executing code within a handler we must lock the GC to prevent
                // any memory allocations.  We must also catch any exceptions.
                gc_lock();
                nlr_buf_t nlr;
                if (nlr_push(&nlr) == 0) {
                    mp_call_function_1(callback, tim);
                    nlr_pop();
                } else {
                    // Uncaught exception; disable the callback so it doesn't run again.
                    tim->callback = mp_const_none;
                    __HAL_TIM_DISABLE_IT(&tim->tim, irq_mask);
                    if (channel == 0) {
                        printf("uncaught exception in Timer(%u) interrupt handler\n", tim->tim_id);
                    } else {
                        printf("uncaught exception in Timer(%u) channel %u interrupt handler\n", tim->tim_id, channel);
                    }
                    mp_obj_print_exception(&mp_plat_print, (mp_obj_t)nlr.ret_val);
                }
                gc_unlock();
                mp_sched_unlock();
            }
        }
    }
}

void timer_irq_handler(uint tim_id) {
    if (tim_id - 1 < PYB_TIMER_OBJ_ALL_NUM) {
        // get the timer object
        pyb_timer_obj_t *tim = MP_STATE_PORT(pyb_timer_obj_all)[tim_id - 1];

        if (tim == NULL) {
            // Timer object has not been set, so we can't do anything.
            // This can happen under normal circumstances for timers like
            // 1 & 10 which use the same IRQ.
            return;
        }

        // Check for timer (versus timer channel) interrupt.
        timer_handle_irq_channel(tim, 0, tim->callback);
        uint32_t handled = TIMER_IRQ_MASK(0);

        // Check to see if a timer channel interrupt was pending
        pyb_timer_channel_obj_t *chan = tim->channel;
        while (chan != NULL) {
            timer_handle_irq_channel(tim, chan->channel, chan->callback);
            handled |= TIMER_IRQ_MASK(chan->channel);
            chan = chan->next;
        }

        // Finally, clear any remaining interrupt sources. Otherwise we'll
        // just get called continuously.
        uint32_t unhandled = tim->tim.Instance->DIER & 0xff & ~handled;
        if (unhandled != 0) {
            __HAL_TIM_DISABLE_IT(&tim->tim, unhandled);
            __HAL_TIM_CLEAR_IT(&tim->tim, unhandled);
            printf("Unhandled interrupt SR=0x%02lx (now disabled)\n", unhandled);
```

```
68            }
69        }
70  }
```

One thing to note here is that it is in fact possible to call other Python code than the one currently executed (or interrupted by the IRQ) by simply using the Micropython function `mp_call_function_<X>` (with <X> number of parameters).

4.2.1 The inner workings of MP_STATE_PORT

To understand how and why the above code works is key to write this or similar code. Diving through the Micropython code, it becomes clear, that `MP_STATE_PORT` is the same as `MP_STATE_VM` [5, ports/stm32/mpconfigport.h]. This macro, given a parameters x, evaluates to `(mp_state_ctx.vm.x)` [5, py/mpstate.h]. Listing 4.15 shows the data structure of the variable `mp_state_ctx`. Leftout code is denoted by "...".

Listing 4.15: Data structure of `mp_state_ctx`

```
1  // This structure hold runtime and VM information.  It includes a section
2  // which contains root pointers that must be scanned by the GC.
3  typedef struct _mp_state_vm_t {
4      ////////////////////////////////////////////////////////////
5      // START ROOT POINTER SECTION
6      // everything that needs GC scanning must go here
7      // this must start at the start of this structure
8      //
9
10     ...
11
12     // include any root pointers defined by a port
13     MICROPY_PORT_ROOT_POINTERS
14
15     ...
16
17 } mp_state_vm_t;
18
19 ...
20
21 // This structure combines the above 3 structures.
22 // The order of the entries are important for root pointer scanning in the GC to work.
23 // Note: if this structure changes then revisit all nlr asm code since they
24 // have the offset of nlr_top hard-coded.
25 typedef struct _mp_state_ctx_t {
26     mp_state_thread_t thread;
27     mp_state_vm_t vm;
28     mp_state_mem_t mem;
29 } mp_state_ctx_t;
30
31 extern mp_state_ctx_t mp_state_ctx;
```

Note the macro `MICROPY_PORT_ROOT_POINTERS`. Coincidentally the `stm32`-port [5, ports/stm32/mpconfigport.h] defines this macro like seen in Listing 4.16.

Listing 4.16: Macro definition of the macro `MICROPY_PORT_ROOT_POINTERS` in the stm32-port

```
1  #define MICROPY_PORT_ROOT_POINTERS \
2      const char *readline_hist[8]; \
3      \
4      mp_obj_t pyb_hid_report_desc; \
5      \
6      mp_obj_t pyb_config_main; \
7      \
8      mp_obj_t pyb_switch_callback; \
9      \
10     mp_obj_t pin_class_mapper; \
11     mp_obj_t pin_class_map_dict; \
```

```
12     \
13     mp_obj_t pyb_extint_callback[PYB_EXTI_NUM_VECTORS]; \
14     \
15     /* Used to do callbacks to Python code on interrupt */ \
16     struct _pyb_timer_obj_t *pyb_timer_obj_all[14]; \
17     \
18     /* stdio is repeated on this UART object if it's not null */ \
19     struct _pyb_uart_obj_t *pyb_stdio_uart; \
20     \
21     /* pointers to all UART objects (if they have been created) */ \
22     struct _pyb_uart_obj_t *pyb_uart_obj_all[MICROPY_HW_MAX_UART]; \
23     \
24     /* pointers to all CAN objects (if they have been created) */ \
25     struct _pyb_can_obj_t *pyb_can_obj_all[2]; \
26     \
27     /* list of registered NICs */ \
28     mp_obj_list_t mod_network_nic_list; \
```

Note, that this macro lists all data structures, that are needed by any driver. This works fine enough, when it is clear what devices will be attached to the system, but does not enable the system to have any form of plug and play, as it is not possible to add any device objects to the list at run-time.

4.2.2 Restrictions for Interrupt Service Routines

Micropython imposes some restrictions for Interrupt Service Routines [9]. First ISRs may not allocate new memory, i.e. no Python object creation and no floating point arithmetic is possible[1]. That is due to an object allocation needing memory from the heap, which is not re-entrant capable, so it could be possible that an ISR is interrupted half-way through allocating new memory leaving the heap in an inconsistent state. This could have disastrous consequences, especially when two ISRs get the same space of memory for different uses. So an ISR would need to incorporate pre-allocated buffers.

This restriction can also make communication between an ISR with the main program difficult. But Micropython allows the use of methods of objects as callback for an ISR. So the interrupt handler can use the member variables of the object to communicate with the main program.

This also implies, that an exception cannot be thrown inside an ISR, except the program had pre-allocated an exception buffer. Listing 4.17 shows the recommended way of pre-allocating an exception buffer. Note that this implies the presence of the `micropython`-module.

Listing 4.17: Pre-allocating an exception buffer

```
1 import micropython
2 micropython.alloc_emergency_exception_buf(100)
```

An interrupt may occur at any point in time, even when the currently executed bytecode is not executed completely. This may lead to a situation where operations on lists, sets or dictionaries may not be completed, meaning they could still be in an inconsistent state. Using an inconsistent data structure may lead to a crash or data loss. A solution could be to secure operations on shared data structures by critical sections, i.e. disabling interrupts before the operation and enabling them afterwards.

Integers are represented as 32-bit, larger integers as Python objects, so using the latter might still result in unwanted behaviour. Using a bytearray or array is generally not dangerous as an operation on these structures can be done in one machine instruction.

[1] in a normal Micropython port

Floating point arithmetic is generally forbidden in ISRs in Micropython. If an interrupt handler needs to use floating point, it can instead use the hardware directly by issuing Thumb instructions as inline assembly in the Python function.

`micropython.schedule` can be used to schedule work for execution "very soon". The scheduled function will be executed when the heap is not locked, so it can create objects and use floats, as well as assume, that data structures are consistent, as they will be executed when a bytecode has finished executing.

5 On code execution

Micropython can emit code for several different machines from the Python code. For the purposes of this port all the native code emitters are left off for the time being. This enables the bytecode emitter, where Micropython creates bytecode from the Python code and interprets it in software.

This text will not go into greater detail on these emitters or the compilation process, but the way Micropython comes to executing the code is very interesting. The most important functions for doing so are listed in Appendix E. For easily compiling new Python code a function `do_string` is used by the port, which was already there in the *bare-arm*-port. Listing 5.1 lists the code for this function. This section focuses on `mp_compile` and `mp_call_function_0`.

Listing 5.1: Code for parsing, compiling and executing Python-code

```
1  void do_str(const char *src, mp_parse_input_kind_t input_kind) {
2      mp_lexer_t *lex = mp_lexer_new_from_str_len(MP_QSTR__lt_stdin_gt_, src, strlen(src), 0);
3      if (lex == NULL) {
4          return;
5      }
6
7      nlr_buf_t nlr;
8      if (nlr_push(&nlr) == 0) {
9          qstr source_name = lex->source_name;
10         mp_parse_tree_t parse_tree = mp_parse(lex, input_kind);
11         mp_obj_t module_fun = mp_compile(&parse_tree, source_name, MP_EMIT_OPT_NONE, true);
12         mp_call_function_0(module_fun);
13         nlr_pop();
14     } else {
15         // uncaught exception
16         mp_obj_print_exception(&mp_plat_print, (mp_obj_t)nlr.ret_val);
17     }
18 }
```

`mp_compile` uses the parse-tree and creates an `mp_obj_t`, i.e. a void-pointer hiding the internal structure of whatever lays behind. It however consists of merely two function calls, first `mp_compile_to_raw_code`, which returns `mp_raw_code_t` and later `mp_make_function_from_raw_code` with the resulting raw code as parameter.

In the first function the parse-tree will be used to compile a tree of `scope_t`-objects. The root-object is called `module_scope` and its attribute `raw_code` is of type `mp_raw_code_t`. The scope objects are initialised through the functions `scope_new_and_link`, `scope_new` and then `mp_emit_glue_new_raw_code`. The newly created scope is added to the tree. Through several calls of `compile_scope` the parse-tree is compiled and code for the target platform generated.

Near the end of the `raw_code` attribute of `module_scope` is singled out and later returned to be passed to `mp_make_function_from_raw_code` back in `mp_compile`. The code object will fulfil the default-path of the switch-statement, so `fun` (the return value) will be created with the return value of `mp_obj_new_fun_bc`.

In that function a real Python object is created. The variable `o` in that function is a C-structure `mp_obj_fun_bc_t`. It contains a base-attribute, which in turn holds a type-attribute. This

type-attribute is set to a pointer to `mp_type_fun_bc`. This class[1] contains a function for printing, a name and most importantly a function for calling an object.

Back in the `do_string` function the function object is to be called. `mp_call_function_n_kw` in turn looks into the object and reads the type pointer of the object. It then calls the call-function with the function object, and the number of parameters as attributes. This function, constantly set to `fun_bc_call`, calls after doing preparational work, `mp_execute_bytecode` to finally execute the bytecode.

It is to note, that even the compiling and calling of Python code in Micropython is inherently Python-based. The parse-tree is used to create a Python object, which can be called, i.e. this functionality can be used inside the Python-language level to compile and run code at run-time.

[1]The structure describes a Python-class as it has been shown in Section 4.1.3.

6 Blinking LED demo

This chapter outlines the blinking LED demo. This demo makes use of the raw pointer class of Section 4.1 to address the GPIO-memory. The RawPtr class is expected to be inside a module names „C". Listing 6.1 shows the Python-code which is executing for the blinking LED demo.

Listing 6.1: Python-code for blinking LED-demo

```
1  import C;
2
3  def gpio_pinmode ( pin, mode ):
4      if ( mode >= 0 and mode <= 3 and pin >= 0 and pin <= 53 ):
5          ptr = C.RawPtr ( 0x20200000 )
6          while (pin >= 10) :
7              pin = pin - 10
8              ptr.offset(4)
9
10         pin = pin * 3
11         mode = mode << pin
12
13         mask = 7 << pin
14         mask = ~mask
15
16         val = ptr.read()
17         val = (val & mask) | mode
18         ptr.write ( val )
19
20 def gpio_write ( pin, val ):
21     ptr = C.RawPtr ( 0x20200000 )
22     if ((val==0 or val==1) and (pin>=0 and pin<=53)) :
23         if (pin > 31) :
24             pin=pin-32
25             ptr.offset(4)
26         if (val == 0) :
27             ptr.offset(40)
28         else :
29             ptr.offset(28)
30         v = 1 << pin
31         ptr.write ( v )
32
33 gpio_pinmode ( 47, 1 )
34 while (1):
35     res = gpio_write ( 47, 0 )
36     x = 0x8000
37     while ( x > 0 ):
38         x = x - 1
39     res = gpio_write ( 47, 1 )
40     x = 0x8000
41     while ( x > 0 ):
42         x = x - 1
```

This demo is basically identical to the first bootable image built in Section 2.1.1 but running as interpreted Python code on top of the Micropython interpreter. It therefore works essentially the same way. At the end of the module, the pinmode of the GPIO-pin 47 is set to 1 (output). The value of the pin is then set to 0 and 1 in an infinite loop. In between each write operation the processor is slowed down to make the switching visible to the naked eye. Again this demo is written for the Raspberry Pi Zero; the correct GPIO-pin to be used for the standard Raspberry Pi is 16.

Instead of the direct memory access using pointers and a mapping structure for the GPIO-registers as done in the C-version, this example is a bit more simple. The raw pointer object in `gpio_write` is initialised with the value of the base address of the registers of the GPIO-controller. Then an offset of 4 byte is added if the pin in question is greater then 31, i.e. not represented anymore in the first set/clear register. Then, according to the value to be written (0 or 1) another offset is added to point to the set-registers (when writing 1), or the clear-registers (when writing 0). At last, the the value is written to the correct location using the raw pointer class.

A thing to note is, that Micropython does not change any settings of the CPU, so it will not put it into a different mode or initialise memory management in any way. This version of the raw pointer class will not add any security to the code, if no virtual memory management is in place. Security can either be done in software, for example by adding a fixed value to the memory address or constraining the raw pointer class to a specific range of addresses, or in hardware by the use of the memory management unit [11].

Note, that this demo makes no use of the interrupts. It would be possible to put the switch on/off functionality inside an interrupt service routine to allow the processor to execute other code, while the interrupt is not triggered. The interrupt could then be triggered by a timer, which has to be programmed beforehand. Then the system could have other functionalities, and every time the interrupt hits, this task is preempted and the LED is switched on or off. As this code has no additional tasks to execute, this complexity is unnecessary.

7 Conclusions

This paper showed a possible way to port Micropython bare-metal to the Raspberry Pi. First it gave an introduction on writing a low level application or kernel for the Raspberry Pi as target hardware, then went into creating Micropython builds for the Raspberry Pi, and later fixing major issues with the Raspberry Pi. The interpreter has been extended by a Python module, which allows direct memory access in Python without the usage of ARM inline assembly on the Python level. Later the paper covered interrupts in Micropython for the reason, that this might become important when writing low level applications in Python. Lastly a blinking LED-demo was presented.

The coverage of Micropython and its internals should shed light on the more intricate details of the interpreter. However even those details are important, when writing code for the platform or maybe porting it to a different board. The shown porting method can be adopted to port Micropython to a different ARM board or maybe even different architecture altogether.

The Micropython port described should be stable enough to develop applications on top. The Raspberry Pi is a good platform for that, as it has enough memory and a GPIO header; the only problem are the proprietary chips, like the SD-card chip or the USB controller, which are complex to program drivers for. But the Video Core IV GPU is quite easy to program, so that a framebuffer can be used to write output to screen in software. The UART can be used as input device, or a set of buttons could be used attached to the GPIO-pins to input data. Also a LCD-display could be used to print debug information.

Also the internals of emitting code has been shown in an overview, so extending Micropython becomes easier, if new bytecode instructions would need to be introduced in the development of an application. This is just a beginning, and further work would have to be done to add bytecode instructions or extend Micropython in any other way, but it gives an overview of the Micropython way of working.

The blinking LED demo is used twice in the paper, once in a C/Assembler form, then later rewritten in Python on top of Micropython, showing, that it is in fact possible to write drivers in Python with the use of the RawPtr-class. Using Micropython to write an application or drivers for the Raspberry Pi should be possible with the RawPtr-class and with inline assembler instructions in Python. One could evaluate if enabling the MMU and enabling caching enhances the performance of the interpreter.

Appendix A - Code for the blinking LED demo in C

Listing 1: main.c; Code for the blinking LED demo

```
#include <stdint.h>


/**
 * \brief a mapping of the memory-registers of the GPIO-controller
 **/
struct _pios_gpio_t
{
    uint32_t fn_select[6];              ///< function select registers for the Pins
    uint32_t pad1;
    uint32_t outputset[2];              ///< registers for setting the levels HIGH
    uint32_t pad2;
    uint32_t outputclear[2];            ///< registers for setting the levels LOW
    uint32_t pad3;
    uint32_t level[2];                  ///< the levels of the pins for reading
    uint32_t pad4;
    uint32_t eventDetect[2];            ///< indicates whether an event has occured (↩
        whathever is enabled)
    uint32_t pad5;
    uint32_t risingEdgeDetect[2];       ///< en/disable rising edge for the eventDetect-↩
        field
    uint32_t pad6;
    uint32_t fallingEdgeDetect[2];      ///< en/disable falling edge for the eventDetect
    uint32_t pad7;
    uint32_t highDetect[2];             ///< en/disable HIGH as event for eventDetect
    uint32_t pad8;
    uint32_t lowDetect[2];              ///< en/disable LOW as event for eventDetect
    uint32_t pad9;
    uint32_t asyncRisingEdgeDetect[2];  ///< en/disable rising edge as asynchronous event ↩
        for eventDetect (i.e. not bound to system clock)
    uint32_t pad10;
    uint32_t asyncFallingEdgeDetect[2]; ///< en/disable falling edge as asynchronous event ↩
        for eventDetect (i.e. not bound to system clock)
    uint32_t pad11;
    uint32_t pullControlEnable;         ///< set the pull-mode (i.e. UP, DOWN or OFF)
    uint32_t pullControlClock[2];       ///< set pins for pulling (see the manual for the ↩
        exact algorithm)
} typedef pios_gpio_t;

volatile pios_gpio_t* const pios_gpio = (volatile pios_gpio_t* const) (0x20200000);

void __attribute__((optimize("O0"))) wait ( int sec )
{
    while ( sec > 0 )
    {
        int counter = 0x400000;
        while ( counter > 0 )
            counter--;
        sec--;
    }
}

/**
 * \brief writes a value to the GPIO-Pin (i.e. writes 1 to set or 1 to clear-registers)
 * \param pin GPIO-Pin (preferably between 0 and 53)
 * \param val output-value (1 for on, 0 for off)
 **/
```

```
53 void pios_gpio_write ( uint32_t pin, uint32_t val )
54 {
55     if ( pin > 53 )
56         return;
57
58     volatile uint32_t* base = &pios_gpio->outputset[0];
59     if ( val == 0 )
60     {
61         base = &pios_gpio->outputclear[0];
62     }
63
64     if ( pin > 31 )
65     {
66         base++;
67         pin -= 32;
68     }
69     (*base) = 1 << pin;
70 }
71
72 /**
73  * \brief sets the PIN into the mode we want
74  * \param pin GPIO-Pin
75  * \param val GPIO-Mode
76  * \return NONE
77  **/
78 void pios_gpio_pinmode ( uint32_t pin, uint32_t val )
79 {
80     if ( val > 7 )
81         return;
82
83     if ( pin > 53 )
84         return;
85
86     volatile uint32_t *base = &pios_gpio->fn_select[0];
87     while ( pin >= 10 )
88     {
89         base++;
90         pin-=10;
91     }
92
93     uint32_t v = *base;
94     v &= ~(7 << ((pin<<1) + pin));     /// move the 7 pin*3 times, so it masks the correct ↩
           bits
95     v |= (val << ((pin<<1) + pin));
96     *base = v;
97 }
98
99 int main() __attribute__ ((section(".init")));
100 int main ()
101 {
102     __asm volatile (
103         "mov sp, #0x8000"
104     );
105     // set pin 47 to output mode
106     pios_gpio_pinmode ( 47, 1 );
107     while (1)
108     {
109         pios_gpio_write ( 47, 1 );
110         wait ( 1 );
111         pios_gpio_write ( 47, 0 );
112         wait ( 1 );
113     }
114     while (1) ;
115 }
```

Listing 2: kernel.ld; Linker script

```
1 SECTIONS {
2     . = 0x8000;
3     .init . : { *(.init) }
4     .text : { *(.text) }
```

```
5      .data : { *(.data) }
6      .bss : { *(.bss COMMON) }
7  }
```

Listing 3: Makefile

```
1  ARM = arm-none-eabi
2  CC = $(ARM)-gcc
3  AS = $(ARM)-as
4
5  CPU=-mcpu=arm1176jzf-s
6  ASOPTS=$(CPU) -g
7  LDOPTS=
8  CFLAGS=$(CPU) -std=c99 -Wall -pedantic -g
9
10 KRNL=kernel
11
12 # The names of all object files that must be generated. Deduced from the
13 # assembly code files in source.
14 ASSRCS := $(patsubst $(SOURCE)%.s,$(BUILD)%.o,$(wildcard $(SOURCE)*.s))
15 CSRCS := $(patsubst $(SOURCE)%.c, $(BUILD)%.o, $(wildcard $(SOURCE)*.c))
16
17 OBJECTS = $(ASSRCS) $(CSRCS)
18
19 all: $(KRNL).img
20
21 # Rule to remake everything. Does not include clean.
22 rebuild: all
23
24 # Rule to make the listing file.
25 $(KRNL).list : $(KRNL).elf
26     $(ARM)-objdump -d $(KRNL).elf > $(KRNL).list
27
28 # Rule to make the image file.
29 $(KRNL).img : $(KRNL).elf
30     $(ARM)-objcopy $(KRNL).elf -O binary $(KRNL).img
31
32 # Rule to make the elf file.
33 $(KRNL).elf : $(OBJECTS) $(LINKER)
34     $(ARM)-ld --no-undefined $(OBJECTS) $(LDOPTS) -Map $(KRNL).map -o $(KRNL).elf -T $(KRNL)↩
           .ld
35
36 # Rule to make the object files.
37 %.o: %.s
38     $(AS) -I. $(ASOPTS) $< -o $@
39
40 %.o: %.c
41     $(CC) -I. $(CFLAGS)  $< -c -o $@
```

Appendix B - Building newlib for Raspberry Pi

Newlib is a standard C subroutine library, which can be built for embedded platforms and it can be used by operating system developers or low level application developers on these platforms. It is interesting for the purposes of writing code on the Raspberry Pi, because it supports the ARM-architecture and a C library can take care of the necessary but annoying tasks when dealing with data on this low level.

The Micropython port described in the main text does not use newlib anymore, but the built-in library written by the Micropython developers. The knowledge of building newlib should however is helpful when using a C library for an embedded project.

Newlib is quite excellent as it is written that way, that it can be used on several platforms without an operating system. For some functionalities it just wants a routine from the system, or the developer writing the low level application. For example she has to provide a `_write`-function, so the newlib library function can use an output function to print some strings. The specifics of that function are not pre-defined, the output could be done to disk, to UART, to screen, or maybe even as a LED-blinking pattern.

Listing 4 shows the configure-command for the newlib-build for the Raspberry Pi. The library is cross-compiled on an x86-PC, not built on the Raspberry Pi itself.

Listing 4: configure-call for building newlib

```
1 CFLAGS_FOR_TARGET="-mcpu=arm1176jzf-s -mfpu=vfp -marm -mfloat-abi=hard"
2 configure --target=arm-none-eabi --enable-newlib-hw-fp --with-float=hard --with-cpu=↩
      arm1176jzf-s --with-fpu=vfp --disable-multilib --disable-shared --enable-target-optspace↩
      --disable-newlib-supplied-syscalls
```

The variable `CFLAGS_FOR_TARGET` is necessary to give some parameters to the compiler in the later Makefile. Especially these parameters are the specific processor the build is for, the FPU (floating point unit), the command for building ARM-code instead of Thumb and specifying the float-ABI as the "hard" ABI, i.e. float calculations should be done in hardware rather than emulated in software.

The configure call then creates a Makefile and tries to guess most of the information, which are not provided as parameters. As configure and make are actually used to create the GCC compiler, newlib also uses the three values for target platform, host platform and build platform. GCC defines the build platform as the platform, on which the GCC is built, host as the one it is run on, target as the platform it will generate code for. Newlib however is a library, and does not use the build platform; the host platform corresponds to the gcc build platform and the target platform to host, i.e. target identifies the platform newlib is supposed to run on, and host where it is built on.

The shown call only specifies the target platform and sets ot to the triple `arm-none-eabi`, enables hardware float support (the library will use float-versions of its math-routines instead of integer ones), sets the hard-float-ABI, processor and FPU. It also disables the multilib-flag, disables the build of shared libraries (so the result will be a static library). The target-optspace

option indicates that the library should use space-optimised versions not time-optimised, and disabling the supplied syscalls will cause newlib to require system calls like `_write` to be present when linking with a low level application.

The following system calls are then required by newlib:

- `_exit` - exit a program without cleaning up files
- `close` - close a file
- `execve` - transfer control to a new process
- `fork` - create a new process
- `fstat` - status of an open file
- `getpid` - get the process id
- `isatty` - query whether output stream is a terminal
- `kill` - send a signal
- `link` - establish a new name for an existing file
- `lseek` - set position in a file
- `open` - open a file
- `read` - read from a file
- `sbrk` - increase program data space (`malloc` is based on this)
- `stat` - status of a file (by name)
- `times` - timing information for current process
- `unlink` - remove a file's directory entry
- `wait` - wait for a child process
- `write` - write to a file

Appendix C - QEMU port of Micropython

There is already an QEMU-port of Micropython, but this one described in this Appendix does not mean that one. Instead the Raspberry Pi port can be ported very easily to QEMUs versatilePb platform. The versatilePb platform emulates an ARM926ej-s processor, a predecessor of the ARM1176jzf-s processor used in the Raspberry Pis SoC. Also the UART emulated in versatilePb can be found in the Raspberry Pis SoC, although it was unfortunately not used by the Micropython port.

The first difference is the position of the first instruction executed. In QEMU this is 0x10000, so the linker-script has to be changed. It would be possible to force the Raspberry Pi to start at an arbitrary position in RAM, which is not described in this paper.

The boot-up code can be copied from the Raspberry Pi version. If the Raspberry Pi version uses the start-up code to set up the interrupt table at 0x0, a thing to note is, that the QEMU versatilePb does not put its interrupt table to 0x0, but a different address. This does however not matter, as Micropython does not use interrupts at this stage.

The last thing to change is the UART-driver. Listing 5 shows the driver, which implements the same interface as the UART-driver for the Raspberry Pi from [10].

Listing 5: UART driver for versatilePb platform of QEMU

```
/**
 * \file qemu/uart.c
 * \brief code for the qemu virtual UART (which apparently emulates a PrimeCell UART)
 * see http://infocenter.arm.com/help/topic/com.arm.doc.ddi0183f/DDI0183.pdf for reference
 **/

#include "pios/uart.h"

#ifdef PIOS_PLATFORM_QEMU

#define PIOS_QEMU_UART_BASE 0x101f1000

enum _pios_qemu_uart_t
{
    QEMU_UART_DR      = 0x0000,       ///< data register
    QEMU_UART_RSR     = 0x0001,       ///< receive status register / error clear registe

    QEMU_UART_FR      = 0x0006,       ///< flag register
    QEMU_UART_ILPR    = 0x0008,       ///< low-power counter register
    QEMU_UART_IBRD    = 0x0009,       ///< integer baud rate register
    QEMU_UART_FBRD    = 0x000a,       ///< fractional baud rate register
    QEMU_UART_LCR_H   = 0x000b,       ///< line control register
    QEMU_UART_CR      = 0x000c,       ///< control register
    QEMU_UART_IFLS    = 0x000d,       ///< interrupt fifo level select register
    QEMU_UART_IMSC    = 0x000e,       ///< interrupt mask set/clear register
    QEMU_UART_RIS     = 0x000f,       ///< raw interrupt status register
    QEMU_UART_MIS     = 0x0010,       ///< masked interrupt status regsiter
    QEMU_UART_ICR     = 0x0011,       ///< interrupt clear register
    QEMU_UART_MACR    = 0x0012,       ///< DMA control register

    QEMU_UART_PID0    = 0x03f8,       ///< don't ask me
    QEMU_UART_PID1    = 0x03f9,       ///< nope, no idea
    QEMU_UART_PID2    = 0x03fa,       ///< really, no idea what this does
    QEMU_UART_PID3    = 0x03fb,       ///< well actually...
```

```
35      QEMU_UART_PCELL0 = 0x03fc,       ///< I don't have a clue
36      QEMU_UART_PCELL1 = 0x03fd,       ///< see the manual and find out yourself
37      QEMU_UART_PCELL2 = 0x03fe,       ///< because I don't know
38      QEMU_UART_PCELL3 = 0x03ff,       ///< nor do I care enough
39  } pios_qemu_uart_t;
40
41  volatile uint32_t* const pios_qemu_uart = (volatile uint32_t* const) PIOS_QEMU_UART_BASE;
42
43  void pios_uart_init ( )
44  {
45
46  }
47
48  void pios_uart_write ( const char* str, size_t len )
49  {
50      for ( ; len>0; len-- )
51      {
52          pios_uart_putchar ( *str );
53          str++;
54      }
55  }
56
57  void pios_uart_read ( char* buff, size_t len )
58  {
59      for ( size_t i = 0; i<len; i++ )
60      {
61          buff[i] = pios_uart_getchar ();
62      }
63  }
64
65
66  void pios_uart_putchar ( const char c )
67  {
68      while (1)
69      {
70          if ( (pios_qemu_uart[QEMU_UART_FR] & (1 << 5)) == 0 )      /// transmit FIFO full?
71              break;
72      }
73      pios_qemu_uart[QEMU_UART_DR] = c;
74  }
75
76  uint32_t pios_uart_getchar ( )
77  {
78      while ( (pios_qemu_uart[QEMU_UART_FR] & (1 << 4)) != 0)
79      {
80
81      }
82      return pios_qemu_uart[QEMU_UART_DR];
83  }
84
85  void pios_uart_puts ( const char* str )
86  {
87      pios_uart_write ( str, strlen( str ) );
88  }
89  void pios_uart_flush ()
90  {
91      while (1)
92      {
93          if ( (pios_qemu_uart[QEMU_UART_FR] & (1 << 5)) == 0 )      /// transmit FIFO full?
94              break;
95      }
96  }
97
98  bool pios_uart_rxReady () {}
99  bool pios_uart_txReady () {}
100 int pios_uart_rxQueue () {}
101 int pios_uart_txQueue () {}
102 void pios_uart_setBaud ( uint32_t baud ) {}
103 void pios_uart_setDataSize ( int size ) {}
104
105 #endif
```

Appendix D - RawPtr class

Listing 6: RawPtr class for Micropython

```
#include "py/mpconfig.h"
#include "py/nlr.h"
#include "py/misc.h"
#include "py/qstr.h"
#include "py/obj.h"
#include "py/runtime.h"

#include <stdio.h>

typedef struct _rawptr_t
{
    mp_obj_base_t base;
    void* address;
} rawptr_t;

extern const mp_obj_type_t rawptr_type;

mp_obj_t rawptr_new(const mp_obj_type_t *type, size_t n_args, size_t n_kw, const mp_obj_t *↩
    args)
{
    mp_arg_check_num(n_args, n_kw, 1, 1, true);
    rawptr_t *self = m_new_obj(rawptr_t);
    self->base.type = &rawptr_type;
    self->address = (void*) mp_obj_get_int(args[0]);
    return MP_OBJ_FROM_PTR(self);
}

STATIC void rawptr_print(const mp_print_t *print, mp_obj_t self_in, mp_print_kind_t kind)
{
    rawptr_t *self = MP_OBJ_TO_PTR(self_in);
    printf ("RawPtr-Type: { .address = 0x%lu }\n", self->address );
}

STATIC mp_obj_t rawptr_increment(mp_obj_t self_in)
{
    rawptr_t *self = MP_OBJ_TO_PTR(self_in);
    self->address++;
    return mp_const_none;
}

STATIC mp_obj_t rawptr_decrement(mp_obj_t self_in)
{
    rawptr_t* self = MP_OBJ_TO_PTR ( self_in );
    self->address--;
    return mp_const_none;
}

STATIC mp_obj_t rawptr_offset ( mp_obj_t self_in, mp_obj_t off )
{
    rawptr_t* self = MP_OBJ_TO_PTR ( self_in );
    int offset = mp_obj_get_int( off );
    self->address = (void*) ((uint32_t) self->address) + ((int) offset);
    return mp_const_none;
}

STATIC mp_obj_t rawptr_setAddr ( mp_obj_t self_in, mp_obj_t addr )
{
    rawptr_t* self = MP_OBJ_TO_PTR ( self_in );
    self->address = (void*) mp_obj_get_int( addr );
```

```
59     return mp_const_none;
60 }
61
62 STATIC mp_obj_t rawptr_getAddr ( mp_obj_t self_in )
63 {
64     rawptr_t* self = MP_OBJ_TO_PTR ( self_in );
65     return mp_obj_new_int( (uint32_t) self->address );
66 }
67
68 STATIC mp_obj_t rawptr_read ( mp_obj_t self_in )
69 {
70     rawptr_t* self = MP_OBJ_TO_PTR ( self_in );
71     uint32_t val = *((uint32_t*) self->address);
72     return mp_obj_new_int( val );
73 }
74
75 STATIC mp_obj_t rawptr_write ( mp_obj_t self_in, mp_obj_t val )
76 {
77     rawptr_t* self = MP_OBJ_TO_PTR ( self_in );
78     uint32_t v = mp_obj_get_int( val );
79     *((uint32_t*)self->address) = v;
80     printf ("WRITE TO 0x%8p -> value: 0x%lx\n", self->address, v );
81     return mp_const_none;
82 }
83
84 MP_DEFINE_CONST_FUN_OBJ_1(rawptr_increment_obj, rawptr_increment);
85 MP_DEFINE_CONST_FUN_OBJ_1(rawptr_decrement_obj, rawptr_decrement);
86 MP_DEFINE_CONST_FUN_OBJ_2(rawptr_setAddr_obj, rawptr_setAddr);
87 MP_DEFINE_CONST_FUN_OBJ_2(rawptr_offset_obj, rawptr_offset);
88 MP_DEFINE_CONST_FUN_OBJ_1(rawptr_getAddr_obj, rawptr_getAddr);
89 MP_DEFINE_CONST_FUN_OBJ_1(rawptr_read_obj, rawptr_read);
90 MP_DEFINE_CONST_FUN_OBJ_2(rawptr_write_obj, rawptr_write);
91
92 STATIC const mp_rom_map_elem_t rawptr_locals_dict_table[] =
93 {
94     { MP_ROM_QSTR(MP_QSTR_inc), MP_ROM_PTR(&rawptr_increment_obj) },
95     { MP_ROM_QSTR(MP_QSTR_dec), MP_ROM_PTR(&rawptr_decrement_obj) },
96     { MP_ROM_QSTR(MP_QSTR_offset), MP_ROM_PTR(&rawptr_offset_obj) },
97     { MP_ROM_QSTR(MP_QSTR_setAddr), MP_ROM_PTR(&rawptr_setAddr_obj) },
98     { MP_ROM_QSTR(MP_QSTR_getAddr), MP_ROM_PTR(&rawptr_getAddr_obj) },
99     { MP_ROM_QSTR(MP_QSTR_read), MP_ROM_PTR(&rawptr_read_obj) },
100    { MP_ROM_QSTR(MP_QSTR_write), MP_ROM_PTR(&rawptr_write_obj) },
101 };
102 STATIC MP_DEFINE_CONST_DICT ( rawptr_locals_dict, rawptr_locals_dict_table );
103
104 const mp_obj_type_t rawptr_type =
105 {
106     { &mp_type_type },
107     .name = MP_QSTR_RawPtr,
108     .print = rawptr_print,
109     .make_new = rawptr_new,
110     .locals_dict = (mp_obj_dict_t*)&rawptr_locals_dict,
111 };
112
113 STATIC const mp_map_elem_t cmodule_globals_table[] =
114 {
115     { MP_OBJ_NEW_QSTR(MP_QSTR_RawPtr), (mp_obj_t)&rawptr_type }
116 };
117
118 STATIC MP_DEFINE_CONST_DICT ( mp_cmodule_globals, cmodule_globals_table );
119
120 const mp_obj_module_t cmodule =
121 {
122     .base = { &mp_type_module },
123     .globals = (mp_obj_dict_t*)&mp_cmodule_globals,
124 };
```

Appendix E - Code necessary for compiling and executing Python-code

Listing 7: Code for compiling and executing Python-code

```c
// from py/emitglue.h
typedef struct _mp_raw_code_t {
    mp_raw_code_kind_t kind : 3;
    mp_uint_t scope_flags : 7;
    mp_uint_t n_pos_args : 11;
    union {
        struct {
            const byte *bytecode;
            const mp_uint_t *const_table;
            #if MICROPY_PERSISTENT_CODE_SAVE
            mp_uint_t bc_len;
            uint16_t n_obj;
            uint16_t n_raw_code;
            #endif
        } u_byte;
        struct {
            void *fun_data;
            const mp_uint_t *const_table;
            mp_uint_t type_sig; // for viper, compressed as 2-bit types; ret is MSB, then ↩
                arg0, arg1, etc
        } u_native;
    } data;
} mp_raw_code_t;


// from py/emitglue.c
mp_raw_code_t *mp_emit_glue_new_raw_code(void) {
    mp_raw_code_t *rc = m_new0(mp_raw_code_t, 1);
    rc->kind = MP_CODE_RESERVED;
    return rc;
}


// from py/objfun.h
typedef struct _mp_obj_fun_bc_t {
    mp_obj_base_t base;
    mp_obj_dict_t *globals;          // the context within which this function was defined
    const byte *bytecode;            // bytecode for the function
    const mp_uint_t *const_table;    // constant table
    // the following extra_args array is allocated space to take (in order):
    //  - values of positional default args (if any)
    //  - a single slot for default kw args dict (if it has them)
    //  - a single slot for var args tuple (if it takes them)
    //  - a single slot for kw args dict (if it takes them)
    mp_obj_t extra_args[];
} mp_obj_fun_bc_t;

// from py/objun.c
const mp_obj_type_t mp_type_fun_bc = {
    { &mp_type_type },
    .name = MP_QSTR_function,
#if MICROPY_CPYTHON_COMPAT
    .print = fun_bc_print,
#endif
    .call = fun_bc_call,
    .unary_op = mp_generic_unary_op,
#if MICROPY_PY_FUNCTION_ATTRS
```

```
57     .attr = fun_bc_attr,
58 #endif
59 };
60 
61 mp_obj_t mp_obj_new_fun_bc(mp_obj_t def_args_in, mp_obj_t def_kw_args, const byte *code, ↩
       const mp_uint_t *const_table) {
62     size_t n_def_args = 0;
63     size_t n_extra_args = 0;
64     mp_obj_tuple_t *def_args = MP_OBJ_TO_PTR(def_args_in);
65     if (def_args_in != MP_OBJ_NULL) {
66         assert(MP_OBJ_IS_TYPE(def_args_in, &mp_type_tuple));
67         n_def_args = def_args->len;
68         n_extra_args = def_args->len;
69     }
70     if (def_kw_args != MP_OBJ_NULL) {
71         n_extra_args += 1;
72     }
73     mp_obj_fun_bc_t *o = m_new_obj_var(mp_obj_fun_bc_t, mp_obj_t, n_extra_args);
74     o->base.type = &mp_type_fun_bc;
75     o->globals = mp_globals_get();
76     o->bytecode = code;
77     o->const_table = const_table;
78     if (def_args != NULL) {
79         memcpy(o->extra_args, def_args->items, n_def_args * sizeof(mp_obj_t));
80     }
81     if (def_kw_args != MP_OBJ_NULL) {
82         o->extra_args[n_def_args] = def_kw_args;
83     }
84     return MP_OBJ_FROM_PTR(o);
85 }
86 
87 STATIC mp_obj_t fun_bc_call(mp_obj_t self_in, size_t n_args, size_t n_kw, const mp_obj_t *↩
       args) {
88     MP_STACK_CHECK();
89 
90     DEBUG_printf("Input n_args: " UINT_FMT ", n_kw: " UINT_FMT "\n", n_args, n_kw);
91     DEBUG_printf("Input pos args: ");
92     dump_args(args, n_args);
93     DEBUG_printf("Input kw args: ");
94     dump_args(args + n_args, n_kw * 2);
95     mp_obj_fun_bc_t *self = MP_OBJ_TO_PTR(self_in);
96     DEBUG_printf("Func n_def_args: %d\n", self->n_def_args);
97 
98     // bytecode prelude: state size and exception stack size
99     size_t n_state = mp_decode_uint_value(self->bytecode);
100    size_t n_exc_stack = mp_decode_uint_value(mp_decode_uint_skip(self->bytecode));
101 
102 #if VM_DETECT_STACK_OVERFLOW
103     n_state += 1;
104 #endif
105 
106     // allocate state for locals and stack
107     size_t state_size = n_state * sizeof(mp_obj_t) + n_exc_stack * sizeof(mp_exc_stack_t);
108     mp_code_state_t *code_state = NULL;
109     if (state_size > VM_MAX_STATE_ON_STACK) {
110         code_state = m_new_obj_var_maybe(mp_code_state_t, byte, state_size);
111     }
112     if (code_state == NULL) {
113         code_state = alloca(sizeof(mp_code_state_t) + state_size);
114         state_size = 0; // indicate that we allocated using alloca
115     }
116 
117     code_state->fun_bc = self;
118     code_state->ip = 0;
119     mp_setup_code_state(code_state, n_args, n_kw, args);
120 
121     // execute the byte code with the correct globals context
122     code_state->old_globals = mp_globals_get();
123     mp_globals_set(self->globals);
124     mp_vm_return_kind_t vm_return_kind = mp_execute_bytecode(code_state, MP_OBJ_NULL);
125     mp_globals_set(code_state->old_globals);
```

```
126
127 #if VM_DETECT_STACK_OVERFLOW
128     if (vm_return_kind == MP_VM_RETURN_NORMAL) {
129         if (code_state->sp < code_state->state) {
130             printf("VM stack underflow: " INT_FMT "\n", code_state->sp - code_state->state);
131             assert(0);
132         }
133     }
134     // We can't check the case when an exception is returned in state[n_state - 1]
135     // and there are no arguments, because in this case our detection slot may have
136     // been overwritten by the returned exception (which is allowed).
137     if (!(vm_return_kind == MP_VM_RETURN_EXCEPTION && self->n_pos_args + self->n_kwonly_args↩
         == 0)) {
138         // Just check to see that we have at least 1 null object left in the state.
139         bool overflow = true;
140         for (size_t i = 0; i < n_state - self->n_pos_args - self->n_kwonly_args; i++) {
141             if (code_state->state[i] == MP_OBJ_NULL) {
142                 overflow = false;
143                 break;
144             }
145         }
146         if (overflow) {
147             printf("VM stack overflow state=%p n_state+1=" UINT_FMT "\n", code_state->state,↩
                 n_state);
148             assert(0);
149         }
150     }
151 #endif
152
153     mp_obj_t result;
154     if (vm_return_kind == MP_VM_RETURN_NORMAL) {
155         // return value is in *sp
156         result = *code_state->sp;
157     } else {
158         // must be an exception because normal functions can't yield
159         assert(vm_return_kind == MP_VM_RETURN_EXCEPTION);
160         // return value is in fastn[0]==state[n_state - 1]
161         result = code_state->state[n_state - 1];
162     }
163
164     // free the state if it was allocated on the heap
165     if (state_size != 0) {
166         m_del_var(mp_code_state_t, byte, state_size, code_state);
167     }
168
169     if (vm_return_kind == MP_VM_RETURN_NORMAL) {
170         return result;
171     } else { // MP_VM_RETURN_EXCEPTION
172         nlr_raise(result);
173     }
174 }
175
176
177 // from py/emitglue.c
178 mp_obj_t mp_make_function_from_raw_code(const mp_raw_code_t *rc, mp_obj_t def_args, mp_obj_t↩
         def_kw_args) {
179     DEBUG_OP_printf("make_function_from_raw_code %p\n", rc);
180     assert(rc != NULL);
181
182     // def_args must be MP_OBJ_NULL or a tuple
183     assert(def_args == MP_OBJ_NULL || MP_OBJ_IS_TYPE(def_args, &mp_type_tuple));
184
185     // def_kw_args must be MP_OBJ_NULL or a dict
186     assert(def_kw_args == MP_OBJ_NULL || MP_OBJ_IS_TYPE(def_kw_args, &mp_type_dict));
187
188     // make the function, depending on the raw code kind
189     mp_obj_t fun;
190     switch (rc->kind) {
191         #if MICROPY_EMIT_NATIVE
192         case MP_CODE_NATIVE_PY:
193             fun = mp_obj_new_fun_native(def_args, def_kw_args, rc->data.u_native.fun_data, ↩
```

```
                rc->data.u_native.const_table);
194             break;
195         case MP_CODE_NATIVE_VIPER:
196             fun = mp_obj_new_fun_viper(rc->n_pos_args, rc->data.u_native.fun_data, rc->data.↩
                u_native.type_sig);
197             break;
198         #endif
199         #if MICROPY_EMIT_INLINE_ASM
200         case MP_CODE_NATIVE_ASM:
201             fun = mp_obj_new_fun_asm(rc->n_pos_args, rc->data.u_native.fun_data, rc->data.↩
                u_native.type_sig);
202             break;
203         #endif
204         default:
205             // rc->kind should always be set and BYTECODE is the only remaining case
206             assert(rc->kind == MP_CODE_BYTECODE);
207             fun = mp_obj_new_fun_bc(def_args, def_kw_args, rc->data.u_byte.bytecode, rc->↩
                data.u_byte.const_table);
208             break;
209     }
210
211     // check for generator functions and if so wrap in generator object
212     if ((rc->scope_flags & MP_SCOPE_FLAG_GENERATOR) != 0) {
213         fun = mp_obj_new_gen_wrap(fun);
214     }
215
216     return fun;
217 }
218
219
220 // from py/compile.c
221 mp_raw_code_t *mp_compile_to_raw_code(mp_parse_tree_t *parse_tree, qstr source_file, uint ↩
    emit_opt, bool is_repl) {
222     /**
223      ...
224      **/
225
226     // free the parse tree
227     mp_parse_tree_clear(parse_tree);
228
229     // free the scopes
230     mp_raw_code_t *outer_raw_code = module_scope->raw_code;
231     for (scope_t *s = module_scope; s;) {
232         scope_t *next = s->next;
233         scope_free(s);
234         s = next;
235     }
236
237     if (comp->compile_error != MP_OBJ_NULL) {
238         nlr_raise(comp->compile_error);
239     } else {
240         return outer_raw_code;
241     }
242 }
243
244 mp_obj_t mp_compile(mp_parse_tree_t *parse_tree, qstr source_file, uint emit_opt, bool ↩
    is_repl) {
245     mp_raw_code_t *rc = mp_compile_to_raw_code(parse_tree, source_file, emit_opt, is_repl);
246     // return function that executes the outer module
247     return mp_make_function_from_raw_code(rc, MP_OBJ_NULL, MP_OBJ_NULL);
248 }
249
250
251 // from py/runtime.c
252 mp_obj_t mp_call_function_0(mp_obj_t fun) {
253     return mp_call_function_n_kw(fun, 0, 0, NULL);
254 }
255
256 mp_obj_t mp_call_function_n_kw(mp_obj_t fun_in, size_t n_args, size_t n_kw, const mp_obj_t *↩
    args) {
257     // TODO improve this: fun object can specify its type and we parse here the arguments,
```

```
258     // passing to the function arrays of fixed and keyword arguments
259
260     DEBUG_OP_printf("calling function %p(n_args=" UINT_FMT ", n_kw=" UINT_FMT ", args=%p)\n"↩
            , fun_in, n_args, n_kw, args);
261
262     // get the type
263     mp_obj_type_t *type = mp_obj_get_type(fun_in);
264
265     // do the call
266     if (type->call != NULL) {
267         return type->call(fun_in, n_args, n_kw, args);
268     }
269
270     if (MICROPY_ERROR_REPORTING == MICROPY_ERROR_REPORTING_TERSE) {
271         mp_raise_TypeError("object not callable");
272     } else {
273         nlr_raise(mp_obj_new_exception_msg_varg(&mp_type_TypeError,
274             "'%s' object is not callable", mp_obj_get_type_str(fun_in)));
275     }
276 }
```

Declaration of authorship

Ich erkläre an Eides statt, gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angebenen Quellen und Hilfsmittel angefertigt habe. Diese Aussage gilt auch für die Implementation und Dokumentation im Rahmen diesen Projekts.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche gekennzeichnet.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keinem anderen Prüfer als Prüfungsleistung eingereicht und ich auch noch nicht veröffentlicht.

I hereby certify to the Technische Universität Chemnitz that this paper is completely my own work and uses no external material other than that acknowledged in the text. This declaration also holds for the implementation and documentation done in the context of this project.

This work contains no plagiarism and all sentences or passages directly quoted from other people's work or including content derived from such work have been specifically credited to the authors and sources.

This paper has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.

Chemnitz, 1st December 2017 (Naumann, Stefan)

Bibliography

[1] ARM. ARM1176jzf-s - About unaligned and mixed-endian support. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0301h/Beieghbg.html. [accessed 2017-11-16].

[2] ARM. ARM1176jzf-s - c15-co-processor Register Allocation. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0301h/ch03s02s01.html. [accessed 2017-11-16].

[3] ARM. ARM1176jzf-s Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI0301H_arm1176jzfs_r0p7_trm.pdf. [accessed 2017-09-20].

[4] Broadcom. BCM2838 ARM Peripherals Manual. https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf. [accessed 2017-09-20].

[5] Micropython Developers. Github - micropython/micropython. https://github.com/micropython/micropython. [accessed 2017-11-10].

[6] Raspberry Pi Foundation. Mechanical Drawing of the Raspberry Pi Zero. https://www.raspberrypi.org/documentation/hardware/raspberrypi/mechanical/Raspberry-Pi-Zero-V1.2-Mechanical.pdf. [accessed 2017-11-03].

[7] Gadgetoid. Raspberry Pi Pinout. https://pinout.xyz/. [accessed 2017-11-03].

[8] Damien George and Stefan Naumann. Github-issue: Wrong label-calculation for JUMP-Bytecode on Raspberry Pi, due to unaligned access. https://github.com/micropython/micropython/issues/3201. [accessed 2017-11-16].

[9] Peter Hinch and Mike Causer. Writing Interrupt Handlers - Microypthon Documentation. http://docs.micropython.org/en/latest/pyboard/reference/isr_rules.html. [accessed 2017-09-12].

[10] Stefan Naumann. PiOS - Simple prototype operating system for the Raspberry Pi. https://github.com/naums/PiOS. [accessed 2017-11-16].

[11] David Welch and Stefan Naumann. Github - MMU on Raspberry Pi. https://github.com/naums/raspberrypi/blob/master/mmu/README.md. [accessed 2017-11-16].