

20-sim Template for Raspberry Pi 3

J. (Jeffrey) Dokter

BSc Report

Committee:

Dr.ir. J.F. Broenink
Z. Lu, MSc

July 2016

019RAM2016
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

20-sim is a software package designed to model and simulate mechatronic systems. It can generate code based on the models and run this on a specific target with real-time measurements using an extension called 20-sim 4C. This allows for rapid prototyping of control software.

The Raspberry Pi 3 is a single board computer designed for educational purposes. With respect to the embedded boards currently used at RaM it is a lot cheaper and sometimes more compact. This means that it might be able to replace those boards in some projects. However, since the Raspberry Pi is a general purpose board, its capabilities with respect to mechatronic systems have to be investigated. It runs Linux, which is a regular operating system without real-time capabilities. These real-time capabilities are required for mechatronic systems, so this could cause problems.

During this project, the Raspberry Pi 3 was made into a 20-sim 4C target. This means that it should be capable of running code generated from models made in 20-sim. Furthermore, the usefulness of the board was explored. Testing was done to see which models could be run, along with how much resources these used. This way the limits of the Raspberry Pi could be determined.

The Raspberry Pi 3 was implemented as a 20-sim 4C target. The first approach was to make it a real-time system by adding the real-time extension Xenomai to the Raspberry Pi and configuring 20-sim 4C accordingly. This did not work due to compatibility issues between the Raspberry Pi 3 and the platform functions provided by 20-sim 4C. Finally, the system was successfully implemented without real-time support.

Using a simple setup, the implementation was tested. The Raspberry Pi can run the code from a model on this test setup, with a sample rate of 1 kHz, which is often used in control engineering. The lack of real-time capabilities is compensated by the relative high processing speed of the board. This means that while the models are simple and nothing else is done, the board is able to run models properly. This means that it can be useful for projects, where real-time criteria are not that critical. However, if missing a deadline cannot happen in any situation, more dedicated boards should be considered.

Contents

1	Introduction	1
2	Background	2
2.1	Raspberry Pi 3	2
2.2	20-sim 4C	3
2.3	Linux	5
3	Design and Implementation	7
3.1	Raspberry Pi	7
3.2	20-sim 4C Files	8
3.3	Additional Implementation Details	10
4	Testing	11
4.1	Test Setup	11
4.2	Test Results	12
4.3	Analysis and Discussion	13
5	Conclusions	14
5.1	Conclusion	14
5.2	Recommendations	14
	Bibliography	22

1 Introduction

In the field of robotics and mechatronics, modelling and simulating are essential to make proper designs. 20-sim is a software package made by Controllab specifically developed for this purpose (Controllab Products, 2016a). It allows the user to make models of robotic and mechatronic systems and simulate them. It also has a built-in C code generator, which can generate C code according to the model.

20-sim 4C is an extension of 20-sim which can be used to upload the code generated by 20-sim to a device and link inputs and outputs of the model to those of the device. Using this extension the inputs and outputs can then be controlled and monitored during the testing. This allows for rapid prototyping of a system (Controllab Products, 2016b).

The Raspberry Pi is a single board computer designed to educate children in programming and computer science, starting from age 6 up to 15. Since it is compact, cheap and multifunctional, it is an interesting device for hobbyists and academics as well. Released a couple of years ago, the concept has proven to be successful, with the foundation selling one million boards within a year (Edwards, 2013).

The boards that are currently used in combination with 20-sim are dedicated towards use in mechatronics systems. Although this means that they will work well on these research projects, there are some disadvantages. Because the boards have a specific purpose, they cannot be used for other types of projects. Furthermore, dedicated hardware is expensive, which is also the case for these boards. Finally, not all the boards used are compact because they include a lot of hardware to support different functions, which means they cannot be used in projects with small prototypes. The Raspberry Pi is compact, general purpose and relatively cheap, which means that it could be a solution if the dedicated boards are not required. However, the performance of the Raspberry Pi 3 has to meet certain standards: using general rule of thumb of a 1 kHz sample rate, the board should be able to control a mechatronic setup. So the main goal of this project is to find out if the Raspberry Pi could contribute something if it was implemented as a 20-sim 4C target.

The work done to make the Raspberry Pi a functional 20-sim 4C target will mostly be based on earlier targets made for similar systems. Controllab provided a target for an older model of the Raspberry Pi. Luuk Grefte previously made a target for the NI MyRIO, which runs on the same operating system (Grefte, 2016). Even though both of these targets differ quite a lot from the one that has to be made, they provide a basic structure and some additional insights into what has to be done and how.

When the board is functioning as a target, the usefulness of the board has to be investigated. Tests are done to see whether the board can keep up with the 1 kHz sample rate and how much resources the 20-sim models use when they are running on the board.

The report starts with a background on several parts of the project required to understand the design and the design process. Then the design itself along with the choices made for the design are discussed. Consequently, the testing of the design as well as the results are explained. The main matter of the report is concluded with the conclusions for this project and the recommendations for future projects about the same subject. Finally some appendices are added, containing additional code, figures, tables and the bibliography.

2 Background

2.1 Raspberry Pi 3

With the third generation of the Raspberry Pi, its creators added more processing power to the board by upgrading the system on chip (SoC). Wireless capabilities, such as Wifi and Bluetooth are also new to the board. Even with all of these new features, it costs as much as its predecessor and the size is also equally small. The entire list of specifications is as follows (Raspberry Pi, 2016):

- SoC: Broadcom BCM2837
- CPU: Quad-Core ARM Cortex-A53, 1.2 GHz
- GPU: Broadcom VideoCore IV
- RAM: 1 GB LPDDR2 (900 MHz)
- Networking: 10/100 Ethernet, 2.4GHz 802.11n wireless
- Bluetooth: Bluetooth 4.1 Classic, Bluetooth Low Energy
- Storage: microSD
- Ports: HDMI, 3.5 mm analogue audio-video jack, 4x USB 2.0, Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI)

An overview of the Raspberry Pi 3 layout is shown in Figure 2.1a.

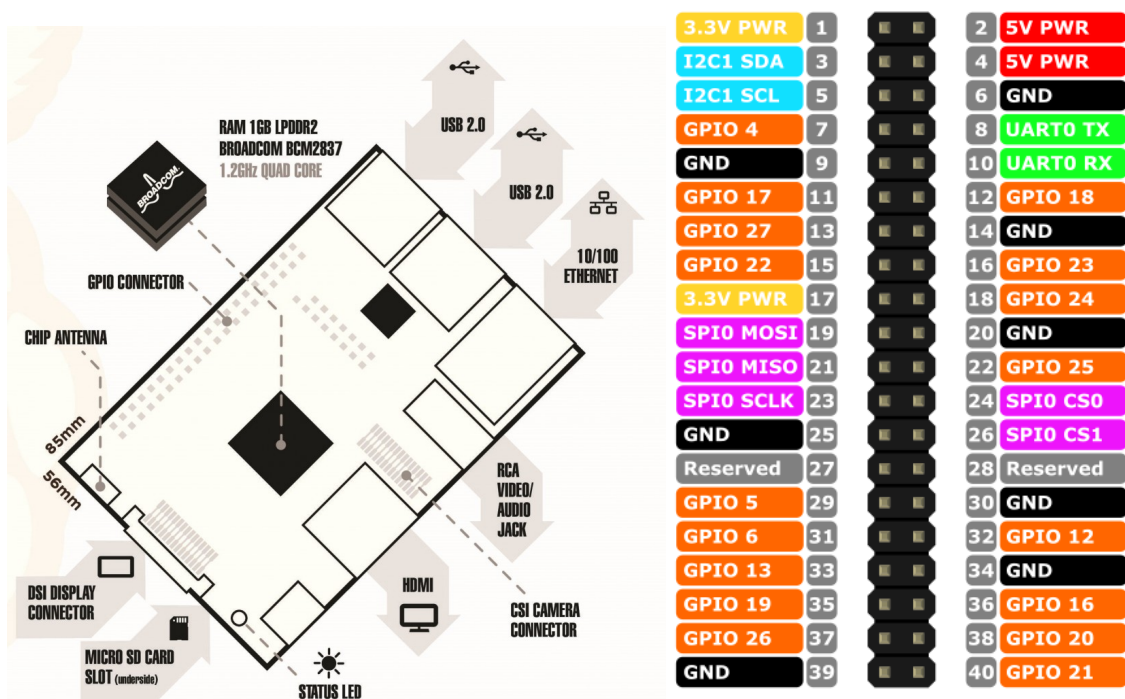


Figure 2.1: Several diagrams showcasing the Raspberry Pi 3

The ARM Cortex-A53 is a CPU designed to have low power consumption and is relatively small. It has an ARM v8 architecture, which includes features like pipelining and branch prediction (ARM, 2016). These features improve the average task execution time, but are inconvenient when processing real-time commands. The commands cannot get a higher priority to make sure they are executed in time. For instance, the pipeline cannot be emptied when a real-time command wants to go first. First all the commands still in the pipeline have to be completed before the real-time command is fully executed. With an eight stage pipeline such as in the Cortex-A53, this will take eight clock cycles, which may be too long.

On the Raspberry Pi there are several expansion headers that can be controlled, but most of them are connected to the peripherals already present on the board, so those cannot be used freely. There is only one large expansion header that can be freely used as digital input and output. It consists of 40 pins, but some of these pins function as voltage supply or ground, so there are effectively 26 pins that can be used as general purpose input and output (GPIO), numbered 2 up to 27. Some of these pins also have additional functions such as: pulse width modulation (PWM), UART or SPI. The pinout of the Raspberry Pi 3 can be seen in Figure 2.1b. In this figure the most important additional functions are shown for some pins, but these pins can still be used as GPIO if this is desired. The reserved, GND or PWR pins can only be used for their designated purpose.

2.2 20-sim 4C

The real-time extension of 20-sim, called 20-sim 4C allows for real-time testing on embedded system using a graphical interface. This means that users do not have to struggle with code and connections and other things that are not really part of the process of designing and implementing a controller for a mechatronic system. Using an ethernet connection between the PC running 20-sim 4C and the target, the 20-sim 4C will take care of every aspect. This is done in four steps: configure, connect, compile and command. An overview of the workflow of 20-sim 4C is displayed in Figure 2.2

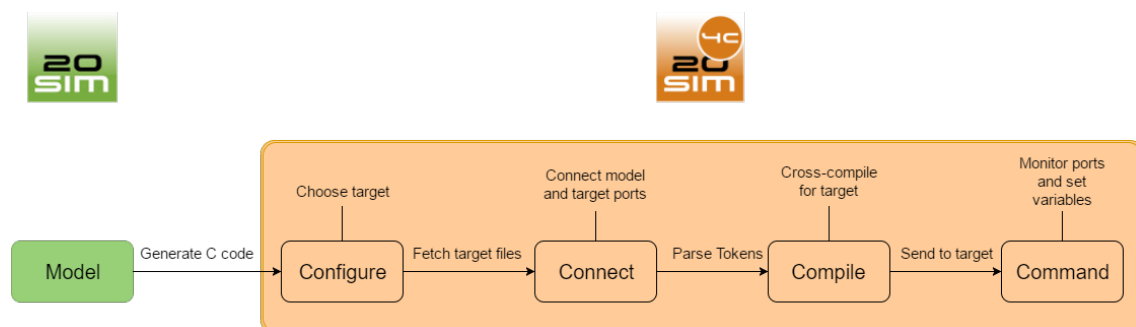


Figure 2.2: Workflow of 20-sim 4C

In the configure step the target is chosen from a list of supported devices. Some additional information is shown in this list about the device and software it should run according to 20-sim 4C. When the device is selected, 20-sim 4C automatically locates the device on the network the computer is connected to and if it is found, connect with the target. Then, in the connect step, every in- and output from the model has to be connected to the inputs and outputs of the device using a graphic interface, which simplifies this process. When all the ports are connected, the compile step is next. During this step, a status window is shown which displays the compile commands and potential errors. When the program is successfully compiled, it is time to command the set-up. The user can choose how long the program has to run and which inputs and outputs have to be monitored or logged. There is also a possibility to adjust the values of constants that are in the models. During all these four steps, there is a log in the

bottom of the screen, providing the user with information about what is done and if anything is out of the ordinary.

All of these steps need information about the target device to function and because there are two devices connected over a network, both sides need files to made or adjustments to be done. The distribution of the different files and programs is shown in Figure 2.3

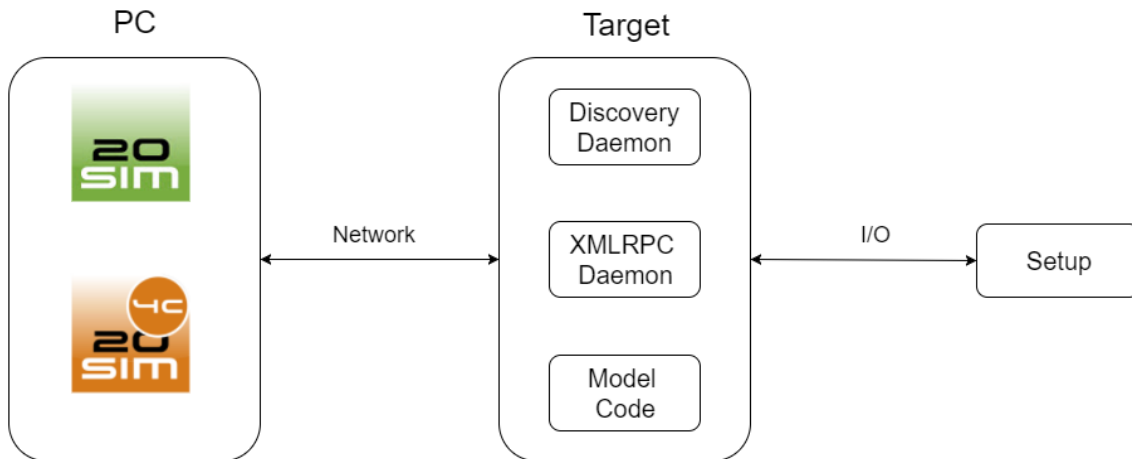


Figure 2.3: Distribution of Programs Across the Devices

2.2.1 Target Side

Because the two devices are communicating across a network, some kind of protocol needs to be used. Controllab developed two pieces of software for this that have to run on the target device, which are both daemons (Kleijn, 2013). Daemons are processes that run in the background on an operating system. This also means that the target device needs to run an operating system, because otherwise such a process cannot be started. The software consists of:

- **Discovery Daemon:** This is a daemon which checks for devices on the network, so the user does not have to fill in the address manually. This is not required for 20-sim 4C to function with the device, but it is convenient and a good starting point, it is easier to run than the XMLRPC daemon and can be used to check whether the system fits the target requirements of 20-sim 4C. This daemon can also be used to check the network connection between 20-sim 4C and the target.
- **XMLRPC Daemon:** essentially this daemon handles all the communication between the device and 20-sim 4C on the computer. It receives and transmits data over the network, executing instructions given by 20-sim 4C. This means it is in charge of setting and reading the pins on the device, as well as running the compiled code

In mechatronic systems, that mostly operate in real-time, there are hard deadlines for events or functions to take place. These are mostly emergency protocols, both can sometimes also be built into regular robots. When controlling a robot, consistency with regards to sending signal to the actuators is critical. If the inputs are too slow, the robot will respond too late and miss a set point, for instance. Because 20-sim focuses on mechatronic systems, it requires some real-time software component on the target device in order to meet these real-time deadlines. Such a component should allow for real-time function calls and memory allocation. For a Linux system, two such components are supported: RTAI and Xenomai.

2.2.2 20-sim 4C Side

In addition to the daemons on the Raspberry Pi, more software has to be written on the side of the computer where 20-sim 4C runs. The C code generated by 20-sim has to be compiled to run on the device. Normally a compiler compiles code that will be used on the same machine as the compiler itself. This means that it only has to know the instruction set of its own architecture. If a compiler has to compile code that will run on a different architecture, it has to use the instruction set of that architecture as well. This is called cross-compiling. A cross-compiler has to be designed for this specific combination since there are now two architectures that have to be combined into one compiler. 20-sim 4C requires the code to be compiled on the system running 20-sim 4C, which has a x86 architecture. The compiled code has to run on the Raspberry Pi, which has an ARM architecture. Furthermore, 20-sim 4C runs on Windows and the Raspberry Pi Linux, so it has an additional change of platform that has to be accounted for. This means a cross-compiler has to be used.

Furthermore, 20-sim 4C has to know how to control the in- and output of the target, which functions to call and how to pass variables. This is specified in a C code file with the according header file. These files are not available and have to be made.

Throughout all the files that 20-sim uses, tokens are used to make the code more readable for software developers and users. To actually use these files in a compiler, these tokens have to be parsed into code that can be interpreted by the compiler. A token-parser file that specifies which files have to be parsed is necessary to make this process happen correctly. This file is made in XML.

The final and main file that 20-sim consults for information is the target configuration file (TCF). This is the main file that will be checked by 20-sim. Here the compiler commands, input and output definitions and general info is provided. This file collects all the information and functions provided by the other parts and presents them to 20-sim 4C in a simple and straightforward manner. This file is written in XML which makes it readable and easy to write.

2.3 Linux

Generally, the Raspberry Pi runs on Linux, which is a general operating system based on UNIX. It consists of a kernel, the core of the operating system which handles fundamental tasks, along with some packages and modules that allow for extended capabilities. These loadable kernel modules (LKM) are an extension to the Linux kernel, that can be loaded when necessary and unloaded when they are not needed anymore. This is a way for Linux to get device drivers when they are required (Corbet et al., 2015). Linux is not designed for real-time applications, so in order to provide these kind of functions on a Linux system, real-time extensions are required such as RTAI and Xenomai.

Both RTAI and Xenomai stem from the same project. Both extensions are based on the implementation of the Adeos/I-Pipe nanokernel between the hardware and the Linux kernel. Real-time systems mostly revolve around interrupts. Interrupts are signals from the hardware that are used to interrupt the current task and do a more urgent task. These interrupts are used to quickly respond to sudden changes. The Adeos/I-Pipe nanokernel functions as a pipeline that will present interrupts from the hardware to the kernel with the first priority. In order to intervene in this process and get the interrupts first, an additional real-time kernel is running next to the regular Linux kernel and has the highest priority. This way the real-time kernel can decide whether the interrupt concerns real-time tasks or not and which kernel should handle it. While Xenomai completely implemented this configuration, RTAI has made some changes. The RTAI

kernel also has a connection to the hardware and can bypass the Adeos layer for critical real-time interrupts. This way the overhead from Adeos is avoided. The difference in layering is shown in Figure 2.4.

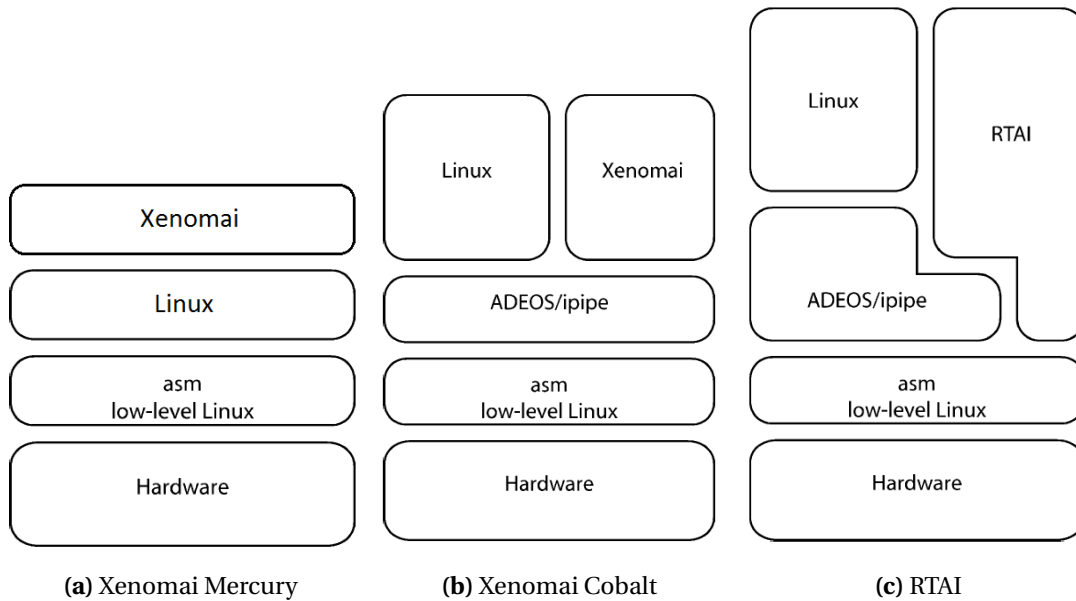


Figure 2.4: Different layering of RTAI and both Xenomai versions. Taken from Barbalace et al. (2008)

While RTAI focuses more on performance, which can be inferred from the way their extension is layered, Xenomai also focuses on simple migration and implementation of applications. It is also better documented and therefore more user friendly. When Xenomai 3 was released, another configuration was introduced as Mercury. The old configuration is also still supported by Xenomai 3, using the name Cobalt. Mercury differs from Cobalt in that it does not require a dual-kernel system along the I-Pipe to function. This means that it is easier to implement and use than Cobalt. It provides multiple real-time application programming interfaces (APIs) such as VxWorks and pSOS. The downside of Mercury is that because there is no longer a dedicated pipeline or kernel, the latencies become longer and less predictable. The different layering of both of the Xenomai versions and RTAI can be seen in Figure 2.4. The layers go from low-level at the bottom to high-level at the top.

3 Design and Implementation

During the execution of the project, several elements had to be designed and implemented. In this chapter the design choices are substantiated. It is split up in two parts: choosing the configuration for the Raspberry Pi and making the files 20-sim 4C needs to use the Raspberry Pi as a target. An overview of the things required for 20-sim 4C to function are displayed in Figure 3.1

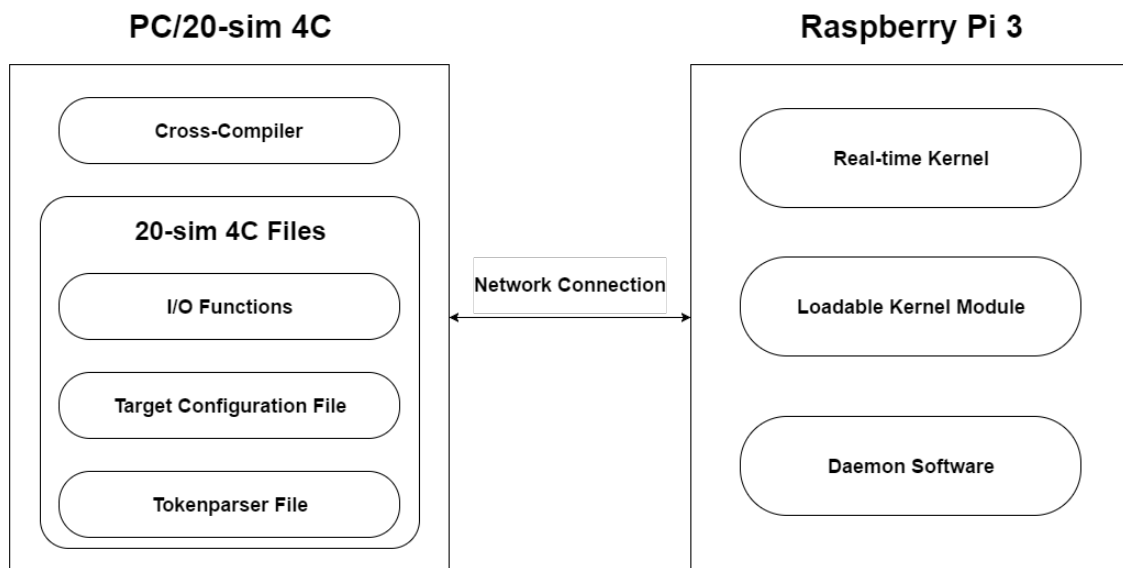


Figure 3.1: Overview of elements to design or produce

3.1 Raspberry Pi

The configuration of the Raspberry Pi is vital for the correct functioning of the device with 20-sim. The activities on the system can be divided in three main categories: configuring the kernel along with the real time extension, getting the daemon software from Controllab to run by installing the right libraries and modifying the files to fit the system and implementing a way to access the GPIO. This final point will be discussed in Section 3.2.2, along with the I/O functions made for 20-sim 4C.

3.1.1 Kernel Configuration

As discussed before, Linux is the operating system used on the Raspberry Pi, with Raspbian as the distribution that is implemented. The Linux kernel that is used is 4.1.21, which is the earliest supported kernel on this model. Setting this up is straightforward: the Raspberry Pi Foundation provides images of these distributions on their website, which than can be flashed onto the SD Card that serves as the main memory of the system. The file system is expanded because additional files need to be stored and samba is installed to allow for easy copying of files onto the Raspberry Pi.

After this, a real-time extension needs to be added to the Raspberry Pi. The choice is between RTAI and Xenomai. According to Barbalace et al. (2008), RTAI is several μ s faster than Xenomai when it comes to just the interrupt latency, with even Linux outperforming Xenomai. This makes sense, because the nanokernel that handles interrupt introduces some overhead with

respect to Linux. When rescheduling is involved, Xenomai does improve the performance of the system, but RTAI is still faster. However, these tests were performed using outdated versions of the software and may not be a good indicator of the current speeds. Furthermore, since Xenomai also focuses more on user-friendliness, Xenomai was chosen as the extension that would be implemented.

The configuration that was implemented was Cobalt, because it ensures more predictable latencies. This is the most significant aspect of this real-time extension. The target provided by Controllab used Xenomai 2.6.4, which only supports the Cobalt configuration, so this also influenced the decision to implement Cobalt.

Although Xenomai is well documented on their website, not much can be found on configuring it or about its hardware floating-point system. Using a combination of the tutorials from both kinsa (2012) and Blaess (2016), Xenomai 3.0.2 was configured onto the kernel. Because 20-sim 4C configures some files into a shared library that is dynamically linked when the application is started on the target, `-enable-dlopen-libs` has to be specifically set, along with the other settings required for the Raspberry Pi. Because Xenomai 3.0.2 only supports up to the 4.1 Linux kernels, the 4.1.21 kernel was used on the system and not a newer kernel that has additional patches and bugfixes, even though those were already available.

3.1.2 Running Daemon software

To build and run the daemon software several packages had to be installed. Both required the installation of WxWidgets, which is a library that provides developers with functions to create applications. Version 2.8.12 had to be installed since some of the functions used in the daemons are outdated, so a newer version will not work. Also it is important to use an ANSI build, since the WxWidgets is installed with a Unicode build by default, which will compromise some function calls. This is enough to build the discovery daemon, which just consists of two cpp files that have to be compiled.

For the construction of the xmlrpc daemon some additional libraries are necessary. The building is done by using a bash script, which will handle the linking and compilation of all the required files. To make the script work, both CMake and SDL are required. CMake is used to generate correct make-files that are used for compiling the software. SDL is a package that is used for quick access to peripherals. It is widely used for video games. Finally, all the paths for the linked files had to be changed, since the daemon that was provided was designed on another computer with a different file system. This completes all the preparation for the build of the xmlrpc daemon. Most of these packages could be installed using the native package manager of the Raspberry Pi, except for Wxwidgets, which has a configure and make script on its own.

3.2 20-sim 4C Files

After everything is finished on the Raspberry Pi, some files for 20-sim 4C itself have to be produced.

3.2.1 Cross-compiler

Building a cross-compiler from scratch is an extensive and precise process that takes a lot of time. Therefore a cross-compiler that was available was used for this project. It is the cross-compiler found on the website of Sysprogs (2016). Running a simple program that was compiled with

this compiler succeeded. It is practically impossible to test all possible programs. Hence, it is assumed that the compiler works properly.

3.2.2 I/O Functions

In order for 20-sim 4C to know how to operate the inputs and outputs of the target device, some target specific IO functions need to be made. There are several ways to control the GPIO pins on the Raspberry Pi, which are listed below.

- WiringPi library (Henderson, 2016)
- BCM2835 (McCauley, 2016)
- Direct Register Access from C files
- Direct Register Access from Loadable Kernel Module (LKM)

The libraries are the easiest to use, but introduce overhead, so not the quickest. Using direct register access is faster, but requires to determination of the register addresses. They also need to be controlled using bit wise operations, which is uncommon. The registers can be accessed in C programs, which runs in user space, or using a LKM, which runs in kernel space. Kernel space has a higher priority than user space, so it is likely that a LKM will be faster. A benchmark was done by Pihlajamaa (2015) and the results are shown in Table 3.1. These benchmarks were done on a Raspberry Pi 2 and consisted of measuring the maximal speed that could be achieved in switching the pins which was measured with an oscilloscope.

Method	Speed[MHz]
WiringPi	4.6
BCM 2708	5.4
Direct Register Access	22

Table 3.1: Benchmarks done on Raspberry Pi 2. Taken from Pihlajamaa (2015)

Unfortunately there is no benchmark using a loadable kernel module or the new Raspberry Pi 3, but this provides some insight about the speed of the several GPIO access methods. Since accessing the registers directly is not that much more difficult than using libraries and it does result in a big increase in speed, the registers will be directly accessed in the code. A loadable kernel modules is used to do this, since it could give another speed boost and there is some experience building such a module, so it should not require additional work to implement this.

By doing as much work as possible in the kernel module, the task that is required should be finished as quickly as possible. Because kernel and user space can only communicate using strings, these are used with delimiters to send the information required to the kernel module. The module will split the strings into tokens, that can then be converted to integers and used for handling the functions, such as setting a pin mode, writing to a pin or reading pin levels.

3.2.3 Target Configuration File

The target configuration file is the main source of information for 20-sim 4C. It is split up into several sections: general, logging, compiler assistant and components. In the general section, information is provided that can be seen in the 20-sim 4C interface, such as name, type, image, icon and a small description. Furthermore, the application files that go to the target are specified, the tokenparser file is linked and global includes and initializations can be done here. The logging section provides the logging plug-in, which is provided by 20-sim 4C, as well as some settings about ip-addresses and port numbers required to properly log everything the XMLRPC

daemon sends.

In the compiler assistant section the compiler and corresponding commands have to be specified. The path to the compiler is provided as well as two compiler commands. The first one compiles all the c files into an executable with the name of the model. The second command compiles all the real-time functions provided by 20-sim 4C along with the specific real-time platform functions into a shared object file that can be linked to the executable of the model later on. Because the Xenomai libraries are linked dynamically, these compiler and linker flags for the commands have to be extracted from the Xenomai installation in the Raspberry Pi. Using the command `xeno-config -skin= -cflags -ldflags`, the flags are found. The skin that was used for the Xenomai platform functions already present was the native skin, so in the config command `-skin=native` was specified. Using these flags, the compile commands were constructed and filed into the TCE.

Next is the components section. Here all the inputs and outputs are specified, along with what functions have to be called to use them. The components are separated into inputs and outputs. For each specific component, four functions are specified. This is where the I/O functions that were made earlier come into place. There has to be a function to construct, initialize, destruct and use the component. The functions are simply specified by their function call. This also shows how the variables from the model are used in the function call. After this, the ports for the component are specified, along with their pins and channel. The channel is the variable that is used in the functions, the pins are displayed in the user-interface. A port can be any set of pins that support the function of the component. So digital input has 26 different ports, while PWM output only has four. While all the functions that are implemented only use a single pin, other functions such as an encoder may require multiple pins, in that case every set of corresponding pins functions as one port.

3.3 Additional Implementation Details

After Xenomai was configured, some problems arose. First off all, the only kernel that was both supported by Xenomai 3.0.2 and the Raspberry Pi 3 was the 4.1.21 version. This is not a problem in itself, but the Raspberry Pi Foundation did not release Linux headers for the kernel version, which are required to compile kernel modules. This problems could still be solved if instead of kernel modules direct register access was used, but there was another problem. The `xenomai_ivcplatformfunctions.c` file required to run the models on the Xenomai platform was outdated. This file comes with the standard 20-sim 4C installation, but it only supports Xenomai 2.6.4 so it could not be used in combination with the 3.0.2 version. Since Xenomai 3.0.2 is a completely rewritten version of Xenomai, this file should also be rewritten to support the new version. This meant researching Xenomai in its entirety and that was not feasible in the final weeks.

Therefore, the choice was made to implement a Linux target without real-time capabilities. This also meant writing platformfunctions for this platform, but since the real-time functions did not have to be written, this was much simpler. Using pthreads, this model is run in a separate thread and the main thread will wait until the thread that runs the model is finished. After this, execution is terminated.

More information about setting up the Raspberry Pi 3 and the produced files can be found in Appendix 2 and Appendix 3.

4 Testing

4.1 Test Setup

The test setup consists of a system that has been built and used before. It consists of a pendulum on a stand, with a motor and encoder. There is one electronics board on the setup that serves two purposes: it has a H-bridge that can be used to control the motor and there is a safety part where optocouplers are used to separate the power between the Raspberry Pi and the motor. This set-up is relatively simple because it only has one motor and one encoder, but it can properly test whether if the Raspberry Pi functions properly as a 20-sim target. It uses all three components that are specified in the TCF: inputs to read the encoder, outputs to set the correct function of the H-bridge and PWM to control the H-bridge. A picture and schematic of the setup are shown in Figure 4.1

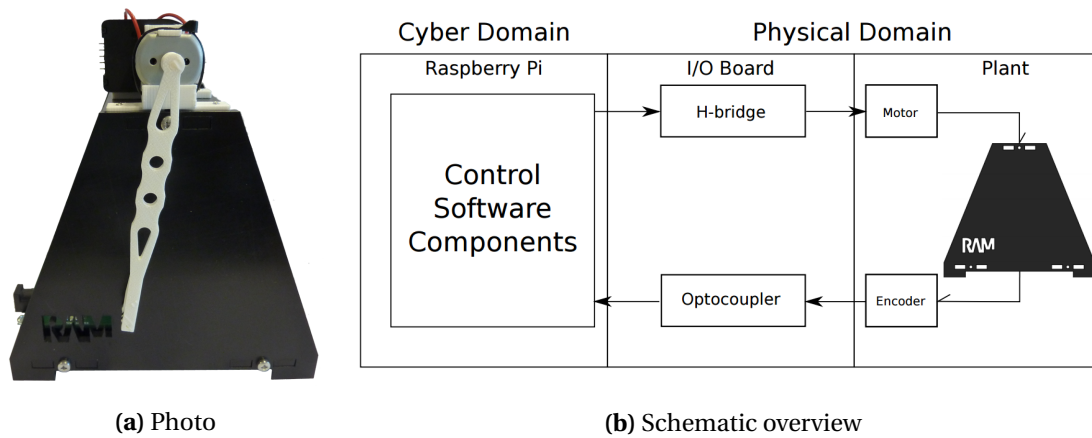


Figure 4.1: CPS set-up. Taken from Broenink et al. (2015)

A simple PD controller was built, using a control law devised in the course where this setup was previously used, 'embedded control system implementation' from module 10 of the Electrical Engineering bachelor. The model was made in 20-sim and is shown in Figure 4.2. It consists of a simple PD controller which uses the control law specified in Equation 4.1. Because the control law uses other units than the model has at its input and output, some conversions are done before and after the controller block. Using the setpoint block, a desired position could be given and with the encoder counts the error could be calculated, which resulted in an PWM output.

The control law used for this setup is as follows:

$$U_d = G_p \cdot (\phi_{setpoint} - \phi) + G_d \cdot \frac{d(\phi_{setpoint} - \phi)}{dt} \quad (4.1)$$

Where U_d is the input voltage of the motor, ϕ are the angles and G are the gains. The gains were taken from Broenink et al. (2015). The proportional gain G_p was set to 17 and the derivative gain G_d was 3. The encoder used on the setup is an X4 encoder, which means that the counts increase on both the rising and falling edges of the pulses. So while the encoder sends 500 pulses per rotation, the counts actually increase with 2000 per rotation. In order to properly read out the encoder, it is polled in the LKM, where also the number of counts is stored. The file with IO functions from 20-sim contains a function that fetches the counts, which are then put into the model.

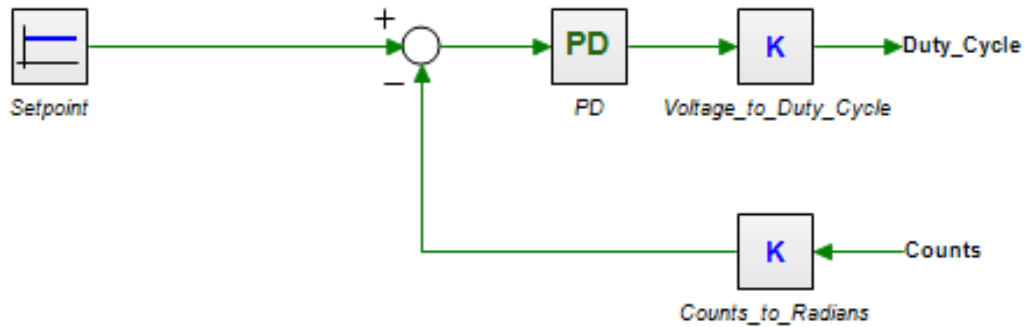


Figure 4.2: The 20-sim model used to control the setup

To run the test, the setpoints were generated as a square wave with an amplitude of 10 rad and an ω of 1 rad s^{-1}

To analyze the amount of resources used by the 20-sim model running on the Raspberry Pi, the `top` command is used to monitor the CPU time used by the specific process of 20-sim 4C. This is possible because the XMLRPC daemon states the process id when it starts the application. While this command is an easy way to monitor the states of several components of the computer, it also puts a strain on those resources itself, because it is a software-based monitor. This means that the system has to run the monitor itself, which can bias the results. So while these values are not very accurate, they provide an indication about the resources used to run the models. The command wrote 10 times per second for 20 seconds to a log file, providing 200 values in total. The CPU time is indicated for a single core of the CPU, which has four cores in total. This means that if a process has a 100% value in `top`, it uses all the CPU-time of a single core.

4.2 Test Results

The target can run the program generated by 20-sim 4C, while monitoring value of the variables in 20-sim 4C itself. The pendulum setup can be controlled, up to a sample rate of 1 kHz. However, the system sometimes cannot settle on the setpoint and can be unstable. Also, logging is not possible due a synchronization error between the application on the Raspberry Pi 3 and 20-sim 4C. This causes the application on the board to finish earlier than the 20-sim 4C logging is done, which corrupts the log files. Fortunately, the monitor data could be stored and a plot of a measurement is shown in Figure 4.3a

Because the model was not entirely stable, the controlled was tuned to be less aggressive, the resulting measurement is shown in Figure 4.3b. For this measurement, the gains were adjusted to 7 and 1.5 for K_p and K_d , respectively.

Running the `top` command when executing the model with a sampling rate of 1kHz results in an average used CPU time of 4.64%. This means that 4.64% of the time the CPU runs, it is busy running the model. The maximum amount used was 9.9% and the minimum $>0.01\%$. For the memory the value was 1.8% during the entire run time of the model. A histogram of the CPU-Time data is shown in Figure 4.4 and the exact data can be found in 3.

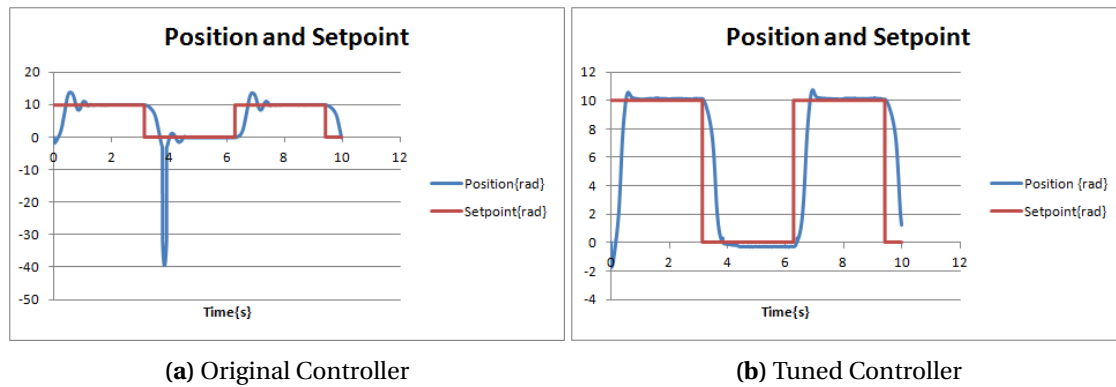


Figure 4.3: Plots of measurements of the setpoint and position

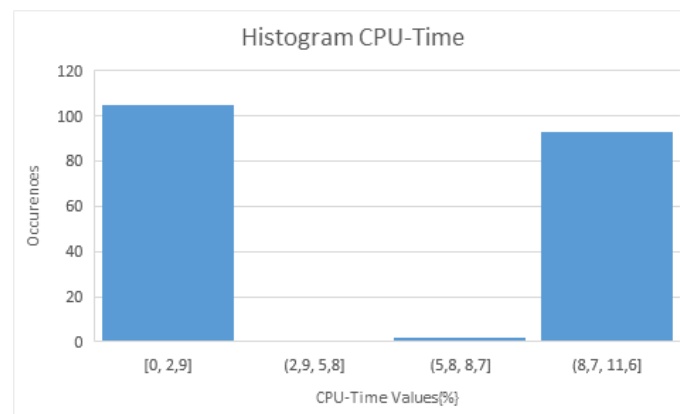


Figure 4.4: Histogram of CPU-Time measured with top

4.3 Analysis and Discussion

While the system can run the model, the setup is unstable and cannot settle on every set point. There are two possible causes for these problems. The controller could malfunction because it was not designed during this project or thoroughly checked before implementing it. This could mean that it is too aggressive, so the gains are too high. The other option is that the system cannot keep up with the sampling rate, causing too much jitter. If the system is too slow, it can be that the motor input is too late every time and the pendulum overshoots. This would mean that the system is not fit for running the model and could be due to the board not having real-time capabilities. By tuning the controller to be less aggressive, the unstable behaviour was mostly eliminated. This means that it is probably due to a bad controller that the system is unstable.

The problem with the logging could be because of a wrong implementation of the platform functions or some kind of communication error between the board and 20-sim 4C. In the second case, this would be because of an error with the XMLRPC daemon, which is something that should be repaired by Controllab.

The usage of resources is very low, especially because the percentage is an indicator for one of four cores. This means that for all four cores of the CPU, this value could be divided by four. This means that this model only uses 1.16% of the entire CPU time, which is very little. This means that even though these results might not be very reliable, the model uses little of the CPU time. Because the amount of resources used by the model is not high, it seems that the instability of the setup is probably caused by a badly tuned controller.

5 Conclusions

5.1 Conclusion

The Raspberry Pi 3 can control a basic setup using 20-sim 4C while keeping up with a general sample rate of 1 kHz and using very little CPU time and memory. This means that the system can certainly be an asset to research chairs that want to do quick prototyping using 20-sim, especially because of its low price. Also, for student projects these boards are useful, because they are small, inexpensive and multi-purpose. For hard real-time applications this board is not qualified with its current configuration. The lack of the real-time extension can mostly be compensated by the high processing speed, but it could be that a critical deadline for a signal is missed. Also the limited input and output capabilities are a downside, because there are no analog inputs and the encoder in the current kernel module is dedicated to a specific setup. Furthermore, all the communication protocols that are supported by the board are not configured for 20-sim 4C. This means that the implementation options are limited.

5.2 Recommendations

Firstly, the timing error that is currently still in the system has to be fixed. If nothing can be logged due to this bug, then none of the test results can be stored or interpreted. Furthermore, if a real-time extension like Xenomai or RTAI could be implemented, this would be a great improvement for the Raspberry PI, since hard real-time requirements could be satisfied. This would require a complete rewrite of the platform functions in the case of Xenomai, but this could be done as a follow up on this project. Another option would be to use a Raspberry Pi 2, which still supports Xenomai 2.6.4 because it can run older Linux kernels. However, this would come at the cost of less processing power, since it has a 900 MHz quad-core processor, while the Raspberry Pi 3 has a 1.2 GHz quad core processor. Support for additional I/O functions could still be added, to extend the range of application for the board. Finally, additional measurements could be done using more complicated models, to see whether it is still feasible to use the Raspberry Pi on complicated projects. Specific timing measurements could also provide additional insights into the real-time capabilities of the board. Doing hardware-based measurements on the controlling of the inputs and outputs and usage of resources provides more quantitative and reliable results. This could indicate what kind of models could be run and also when a real-time extension is really necessary.

All in all, the Raspberry Pi 3 is promising as a board to be used in embedded systems. If the capabilities are extended, this board could be widely used, especially in promising markets, such as domotics and mechatronics as well as the academic world.

Appendix 1: Data Gathered from Top

Time{s}	CPU-Time{%	Time{s}	CPU-Time{%	Time{s}	CPU-Time{%
0	6.4	4.6	9.8	9.2	9.8
0.1	0	4.7	9.8	9.3	0
0.2	9.8	4.8	0	9.4	9.8
0.3	0	4.9	9.8	9.5	0
0.4	0	5	0	9.6	9.8
0.5	9.8	5.1	9.8	9.7	0
0.6	9.8	5.2	0	9.8	0
0.7	0	5.3	0	9.9	9.8
0.8	9.7	5.4	9.7	10	0
0.9	0	5.5	0	10.1	9.7
1	9.7	5.6	9.8	10.2	0
1.1	0	5.7	0	10.3	9.8
1.2	9.8	5.8	9.8	10.4	0
1.3	0	5.9	0	10.5	0
1.4	0	6	9.8	10.6	9.7
1.5	9.7	6.1	0	10.7	0
1.6	0	6.2	9.8	10.8	9.8
1.7	9.8	6.3	0	10.9	0
1.8	0	6.4	9.8	11	0
1.9	9.7	6.5	8.7	11.1	9.8
2	0	6.6	9.7	11.2	0
2.1	0	6.7	9.8	11.3	9
2.2	9.8	6.8	9.8	11.4	0
2.3	0	6.9	9.8	11.5	9.9
2.4	9.8	7	9.8	11.6	0
2.5	0	7.1	9.8	11.7	0
2.6	0	7.2	0	11.8	9.9
2.7	9.7	7.3	0	11.9	0
2.8	0	7.4	9.8	12	9.9
2.9	9.8	7.5	0	12.1	0
3	0	7.6	9.8	12.2	0
3.1	9.8	7.7	0	12.3	9.9
3.2	0	7.8	9.8	12.4	0
3.3	9.8	7.9	0	12.5	9.9
3.4	0	8	9.8	12.6	0
3.5	9.7	8.1	0	12.7	0
3.6	9.8	8.2	0	12.8	9.9
3.7	9.8	8.3	9.7	12.9	0
3.8	9.8	8.4	0	13	9.9
3.9	0	8.5	9.8	13.1	0
4	9.8	8.6	0	13.2	0
4.1	9.8	8.7	9.8	13.3	9.9
4.2	9.8	8.8	0	13.4	0
4.3	9.8	8.9	9.8	13.5	9.9
4.4	9.8	9	0	13.6	0
4.5	9.8	9.1	0	13.7	0

Time{s}	CPU-Time{%	Time{s}	CPU-Time{%	Time{s}	CPU-Time{%
13.8	9.9	15.9	9.9	18	0
13.9	0	16	0	18.1	9.9
14	9.9	16.1	0	18.2	0
14.1	0	16.2	9.9	18.3	0
14.2	0	16.3	0	18.4	9.9
14.3	9.9	16.4	9.9	18.5	0
14.4	0	16.5	0	18.6	9.9
14.5	9.9	16.6	0	18.7	0
14.6	0	16.7	9.9	18.8	0
14.7	9.9	16.8	0	18.9	9.9
14.8	0	16.9	9.9	19	0
14.9	0	17	0	19.1	9.9
15	9.9	17.1	0	19.2	0
15.1	0	17.2	9.9	19.3	0
15.2	9.9	17.3	0	19.4	0
15.3	0	17.4	9.9	19.5	9.9
15.4	9.9	17.5	0	19.6	0
15.5	0	17.6	0	19.7	9.9
15.6	0	17.7	9.9	19.8	0
15.7	9.9	17.8	0	19.9	0
15.8	0	17.9	9.9		

Appendix 2: 20-Sim 4C File Structure and Code

In the 20-sim 4C main folder, there is a target folder. Here, every target has a separate folder that uses a specific structure. In this appendix, the folder structure and code files for the Raspberry Pi target are explained. The main target folder contains at most 6 different folders: doc, etc, examples, lib, include and src. For the Raspberry Pi target, the doc, examples, lib and include folders are empty. The doc folder is for documentation, and the lib and include folder are for libraries that have to be included into the target code. The structure of folders and files is shown in Figure 5.1.

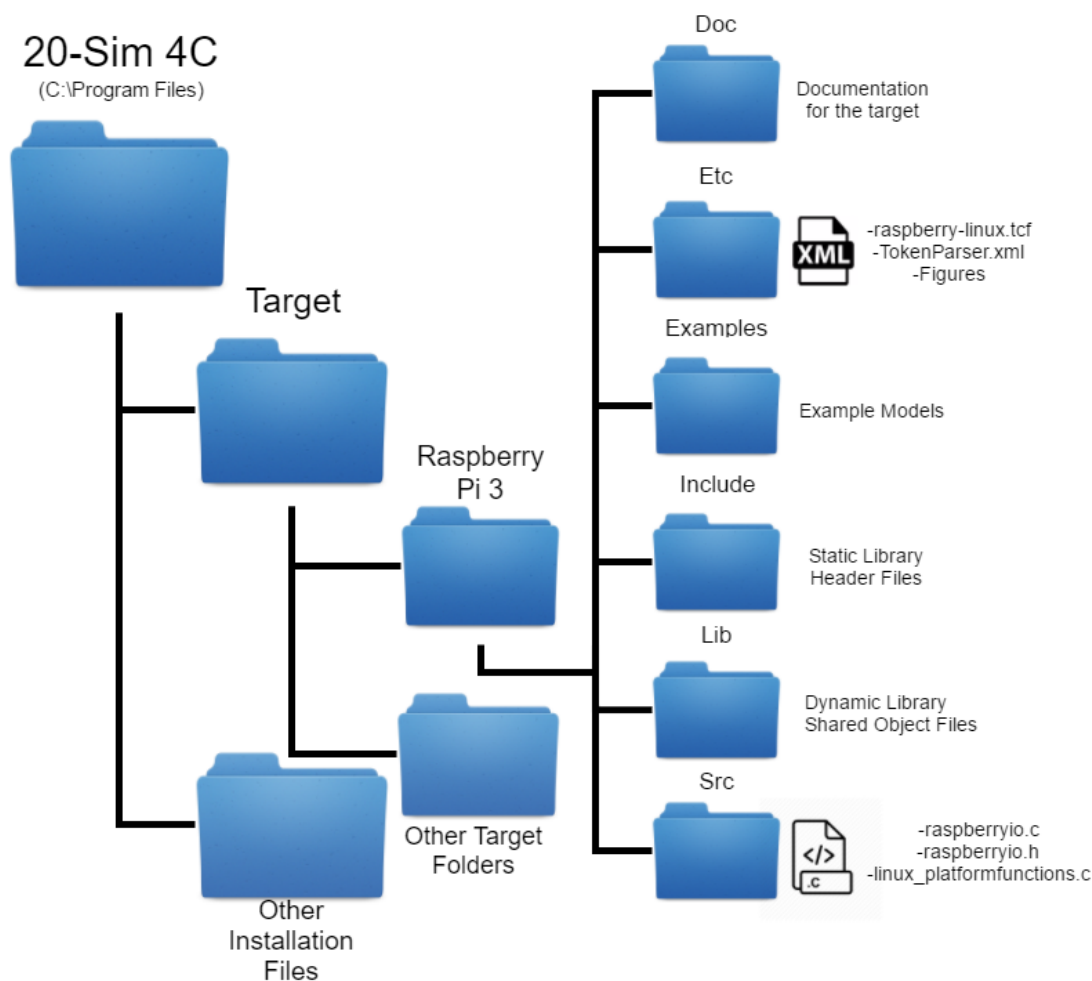


Figure 5.1: Folder and File Structure of 20-sim 4C target folder

Etc Folder

The etc folder can be seen as the main folder. This is where the target configuration file is located, which is the main file that 20-Sim 4C consults. There are four files present in this folder:

- raspberry-linux.tcf
- TokenParser.xml
- Debian.ico

- raspberry.jpg

Where `Debian.ico` and `raspberry.jpg` are figures used in the target configuration file(TCF) to show in the 20-sim 4C user interface.

5.2.1 TokenParser

The file `TokenParser.xml` specifies all the files that have to be parsed by the tokenparser. This was done in the TCF for previous versions of 20-sim 4C, but to keep the code more tidy, this was separated. Now the TCF links to this file. In this file, all the files from the `src` folder and the common `ivc` files included in the 20-sim installation are specified. If some real-time platform is used, the library files should be included here as well.

5.2.2 Target Configuration File

In the target configuration file, which is already discussed in Section 3.2.3, all the files are linked to the target, and all additional information is specified. The sections of this file that have been notably changed are the general, compilerassistant and components sections. The general file just includes the figure mentioned earlier, as well as a small description of the target.

The compilerassistant contains the compile commands to cross-compile the model code for the Raspberry Pi, the commands are shown in Listing 5.1.

```

1
2 "C:\SysGCC\Raspberry\bin\arm-linux-gnueabi-gcc.exe" -o
   %FC_MODELNAME% -O2 %4C_COMPILE_DEFINITIONS%
   %4C_COMPILE_INCLUDE_DIRECTORIES%
   %4C_COMPILE_LIBRARY_DIRECTORIES% *.c
   %4C_COMPILE_LIBRARIES% -lm -lpthread
3
4 "C:\SysGCC\Raspberry\bin\arm-linux-gnueabi-gcc.exe" -o
   lib%FC_MODELNAME%_ipc.so -DDEBUG -DIPC_LIBRARY -shared
   -O2 -Wall ivcipc.c ivcipclibrary.c ivcstorage.c
   linux_ivcplatformfunctions.c -lm -lpthread

```

Listing 5.1: Compiler Commands

First of all, these commands specify the compiler executable. After the `-o` option, the name of the resulting compiled executable is stated. Then there are some compiler flags, after which the files that have to be compiled are given. This is every C file in the folder for the first command (*.c) and several `ivc` C files for the second command. After the file definitions, there are some linker flags which specify the libraries that have to be linked to the compiled file.

In the components section, the GPIO Input, Encoder Input, GPIO Output and PWM Output are defined. As an example, the GPIO Input component is partially shown in Listing 5.2.

```

1 <COMPONENT>
2 <NAME>GPIO Digital Input</NAME>
3 <DESCRIPTION>
4 <![CDATA[Connect to the Expansion Header as digital
   input]]>
5 </DESCRIPTION>
6 <INCLUDES>
7 <![CDATA[#include "raspberryio.h"]]>

```

```

8  </INCLUDES>
9  <CONSTRUCT>
10 <![CDATA[initdriver(); /* GPIO Digital Input */]]>
11 </CONSTRUCT>
12 <INIT>
13 <![CDATA[gpio_SelectPinFunction(%PORT_CHANNEL%,
14         GPIO_INPUT);]]>
15 </INIT>
16 <DESTRUCT>
17 <![CDATA[exitdriver();]]>
18 </DESTRUCT>
19 <FUNCTION>
20 <![CDATA[%VAR% = gpio_ReadPin(%PORT_CHANNEL%);]]>
21 </FUNCTION>
22 <PORTS>
23 <PORT>
24 <NAME>GPIO 2</NAME>
25 <CHANNEL>2</CHANNEL>
26 <PINS>
27 <PIN>Rpi/P1/3</PIN>
28 </PINS>
29 </PORT>

```

Listing 5.2: GPIO Input Component

Here, all the functions that have to do with this component can be seen, as well as one of the ports that is specified.

Src Folder

The src folder contains all the C code that has to be included into the compilation of the model code. There are three files in the folder:

- raspberryio.c
- raspberryio.h
- linux_platformfunctions.c

5.2.3 Raspberry IO

Since the model code runs on the Raspberry Pi 3 in user-space and the LKM, which has to control the I/O, runs in kernel space, some kind of gateway has to be made for the model code to control the I/O. The raspberryio files serve this purpose. During the initialization function, a file pointer is opened that allows for the communication of strings between the kernel module and the raspberryio code. Using this file pointer, strings are sent that contain the information about the tasks the kernel modules has to do. As an example, the function used to read pins is displayed in Listing 5.3.

```

1  int gpio_ReadPin (unsigned int pin)
2  {
3      char str[20];
4      char level[20];
5      int level_read;

```

```
6     //Put parameters in string with / delimiters. 3
7     //for function pin read, pin for the pin to read
8     sprintf (str, "3/%d", pin);
9     //Request level read by sending string to kernel
10    //module
11    write(fp, str, 20);
12    //Read pin level from kernel module
13    read(fp, level, 20);
14    //Convert to integer
15    level_read = strtol(level, NULL, 10);
16    return level_read;
17 }
```

Listing 5.3: Function to read pin level from `raspberrypi.c`

The header file `raspberrypi.h` contains all the function and constant declarations and is included in the TCF, while `raspberrypi.c` contains the actual function code.

5.2.4 Linux Platformfunctions

Because regular target of 20-sim 4C run using a real-time platform or extension like Xenomai or RTAI, platform functions have to be defined. These functions define how to do general real-time tasks like allocating memory, starting a task or thread and waiting for tasks or threads to finish. Even though not all of these tasks are necessary because this Raspberry Pi target is not real-time, some functions are always called in the routine of 20-sim 4C. These functions are specified in `ivcmain.c`, which is located in the common target folder, included in the 20-sim 4C installation. Some of these functions were already specified for this target by Controlab, who provided an earlier version of the target. The two functions that had to be made are `ivcStartTask` and `ivcWaitForTaskFinish`. At first, these functions were empty, which means that the model will not run. In order to make the model run, `ivcStartTask` starts a thread that runs the task. In order to completely let the model run its specified time, `ivcWaitForTaskFinish` waits until this thread is finished.

Appendix 3: Setting up the Raspberry Pi 3 for 20-sim 4C

This Appendix will serve as a guide for someone who wants to set up the Raspberry Pi 3 as a 20-sim 4C target. It will contain how to run the daemon code from controllab and how the kernel module works. All of the required files can be found in the Rpi3 target folder in the 20-sim 4c templates folder on the RaM SVN server. From this folder, the daemon, IOdriver and WxWidgets-2.8.12 folders need to be copied to the Raspberry Pi. Everything in the folder on the SVN server are for the 20-sim 4C target folder. More about this is explained in Appendix 2. The cross-compiler was installed using the instructions on <http://gnutoolchains.com/raspberry/tutorial/>. The path of the installation folder of the cross-compiler is also specified in the Target Configuration File, so installing it anywhere else might require changing the path for the compile command.

Running the Daemons

The library that has to be manually installed on the Raspberry Pi, is WxWidgets. In order to correctly build the library, use the command `./configure --disable-unicode --prefix=$(pwd)`. This will put the library in the folder where the configure command is run and make sure an ansi-build is made. Because of the paths specified in the daemon software, the WxWidgets-2.8.12 build folder needs to be called ansi-build and be in the home folder of user pi. So the path to the build folder should be `\home\pi\WxWidgets-2.8.12\ansi-build`. Otherwise the filepaths to the WxWidgets library should be adjusted in the makefiles of the daemon software.

For building the daemons, the Cmake and SDL packages have to be installed: `sudo apt-get install cmake build-essential libsdl1.2-dev`. The discovery daemon has to be compiled using the makefile in the controllab-discoverydaemon folder. The executable can then be run from the build subfolder using `./discoverydaemon`.

The XMLRPC daemon has to be installed using the `installxmlrpc.sh` script found in the daemon folder. It can then be run from `\usr\sbin` using the `./controllab-xmlrpcd` command. The log information from the XMLRPC daemon can only be seen on the terminal that started the daemon. The daemon will keep running when the terminal is closed, but the logs are not shown anymore.

Kernel Module Insertion

The kernel module can be compiled using the makefile in the IOdriver folder. In order to be able to compile a kernel module, the correct linux headers need to be installed. This was done using `rpi-update` and `rpi-source`. `Rpi-update` can be installed using the package manager and then executed with the command `sudo rpi-update`. This will also update the kernel, so watch out if this is not what is desired. Then `rpi-source` is used to install the correct header for the updated system, as well as create the correct paths. It can be installed using `sudo wget https://raw.githubusercontent.com/notro/rpi-source/master/rpi-source -O /usr/bin/rpi-source && sudo chmod +x /usr/bin/rpi-source && /usr/bin/rpi-source -q --tag=update`. Then run it using by running `sudo rpi-source` from the command line. This should be enough to compile the kernel module, which can then be inserted using the `insmod iodriver.ko` command.

Bibliography

- (2016), Raspberry Pi 3 Model B - Newest Version Pi Supply.
<https://www.pi-supply.com/product/raspberry-pi-3-model-b-newest-version/>
- ARM (2016), Cortex-A53 Processor.
<http://www.arm.com/products/processors/cortex-a/cortex-a53-processor.php>
- Barbalace, A., A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa and C. Talierno (2008), Performance Comparison of VxWorks, Linux, RTAI, and Xenomai in a Hard Real-Time Application, **vol. 55**, no.1, pp. 435–439, ISSN 0018-9499, doi:10.1109/TNS.2007.905231.
- Blaess, C. (2016), Xenomai 3 sur Raspberry Pi 2.
<http://www.blaess.fr/christophe/2016/05/22/xenomai-3-sur-raspberry-pi-2/>
- Broenink, T., F. Kuipers, M. Venema, M. Derks and J. Broenink (2015), Embedded Control Systems Implementation Manual.
https://blackboard.utwente.nl/bbcswebdav/pid-896043-dt-content-rid-1876344_2/courses/2015-201500053-1B/ECSImanual-v4.pdf
- Controllab Products, B. (2016a), 20-sim home page.
<http://20sim.com/>
- Controllab Products, B. (2016b), What is 20-sim 4C.
<http://20sim.com/products/20sim4c.html>
- Corbet, J., A. Rubini and G. Kroah-Hartman (2015), *Linux Device Drivers*, O'Reilly Media, Inc, 3rd edition, ISBN 0-596-00590-3.
<http://free-electrons.com/doc/books/ldd3.pdf>
- Edwards, C. (2013), Not-so-humble raspberry pi gets big ideas, **vol. 8**, no.3, pp. 30–33, ISSN 1750-9637, doi:10.1049/et.2013.0301.
- Grefte, L. (2016), *NI myRIO as target for 20-sim*, BSc Thesis, Univeristy of Twente.
- Henderson, G. (2016), WiringPi.
<http://wiringpi.com/>
- kinsa (2012), Xenomai Packages.
<https://www.raspberrypi.org/forums/viewtopic.php?f=71&t=74686>
- Kleijn, C. (2013), *20-sim 4C 2.1 Reference Manual*, Controllab Products, ISBN 978-90-79499-14-4.
- McCauley, M. (2016), bcm2835: C library for Broadcom BCM 2835 as used in Raspberry Pi.
<http://www.airspayce.com/mikem/bcm2835/index.html>
- Pihlajamaa, J. (2015), Benchmarking Raspberry Pi GPIO Speed.
<http://codeandlife.com/2012/07/03/benchmarking-raspberry-pi-gpio-speed/>
- Raspberry Pi, F. (2016), Raspberry Pi 3 is out now! Specs, benchmarks & more.
<https://www.raspberrypi.org/magpi/raspberry-pi-3-specs-benchmarks/>
- Sysprogs (2016), Tutorial: building Raspberry PI apps from Windows.
<http://gntoolchains.com/raspberry/tutorial/>
- weergave (2015), Raspberry Pi 2.
<https://hackflag.org/forum/showthread.php?tid=5372>