MEASURING THE REAL-TIME LATENCY OF AN I.MX7D USING XENOMAI AND THE YOCTO PROJECT

MÄTA RESPONSTIDEN AV EN I.MX7D MED HJÄLP AV XENOMAI OCH YOCTO PROJEKTET

Bram Coenen
Abstract

In this thesis the real-time latency of an i.MX7D processor on a CL-SOM-IMX7 board is evaluated. The real-time Linux for the system is created using Xenomai with both the I-Pipe patch and the PREEMPT_RT patch. The embedded distribution is built using the Yocto Project and uses a vendor i.MX kernel maintained by NXP.

The maximum latency for the cobalt core is 268µs for user-space tasks with a loaded CPU. These types of tasks have the highest latency of Xenomai’s three task categories. A latency measurement of the PREEMPT_RT patch showed a maximum latency of 412µs with an idle CPU. Therefore it is concluded that the cobalt core has a lower latency and is therefore better suited for real-time applications.

A comparison is made with other modules and it is found that the latency measured in this thesis is high compared to for example a Raspberry Pi 3B.

The source code and configurations for the project can be found at, https://github.com/bracoe/meta-xenomai-imx7d

Sammanfattning

Denna uppsats utvärderar realtidsfördröjningen för en i.MX7D på en CL-SOM-IMX7. Realtidoperativsystemet skapas med hjälp av Linux och både Xenomais I-Pipe patch och PREEMPT_RT patch implementeras. Den inbyggda distributionen byggs med hjälp av Yocto projektet och använder NXP:s egna Linux kärna.

Den maximala fördröjningen för cobalt kärnan är 268µs för user-space uppgifter med en belastad CPU. Dessa typer av uppgifter har den högsta fördröjningen av Xenomais tre uppgiftskategorier. En fördröjningsmätning av PREEMPT_RT patchen visade en maximal fördröjning på 412µs med en överksam CPU. Slutsatsen görs att cobalt kärnan har en lägre fördröjning och är därför mer lämpad för realtidsapplikationer.

En jämförelse görs med andra moduler och den visar att fördröjningen mätt i denna uppsats är hög jämfört med till exempel en Raspberry Pi 3B.

Källkoden och konfigurationer kan hittas på, https://github.com/bracoe/meta-xenomai-imx7d
Acknowledgements

I would like to thank Bosch Rexroth in Mellansel for giving me the opportunity to be a co-op-student at the company and allowing me to write my thesis with them. I am especially grateful to everyone at DC-HD/ENG2 with whom I have spent the last two summers and these ten weeks. They have welcomed me with open arms and we have shared many funny moments at the fika-table together.

A special thanks goes to Börje Pauler who was kind enough to be my supervisor at the company and supported me during this thesis.

I would also like to thank my university supervisor, John Berge, for teaching me about embedded systems and thereby fuelling my passion which led to this thesis.
Contents

List of Abbreviations 1

1 Introduction 2
   1.1 Business Application 2
   1.2 Purpose and Goal 3
   1.3 Specifications 3
      1.3.1 Real-time operating system and benchmarking 3
      1.3.2 Yocto 4
      1.3.3 Licenses 4
      1.3.4 Summary 4

2 Theory 5
   2.1 Linux 5
   2.2 Real-time Operating system 6
   2.3 Xenomai as a Real-time Operating System 7
   2.4 The Yocto Project 8
   2.5 The Hardware 9
   2.6 Toolchain for cross-compilation 9
   2.7 Related Literature 10
   2.8 Open-source licences 10

3 Method 12
   3.1 Building images for the CL-SOM-iMX7 using Yocto 12
   3.2 Early debugging 14
   3.3 Xenomai’s Cobalt Core on an i.MX7D 15
CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4 Creating the SDK for Cross-Compilation</td>
<td>16</td>
</tr>
<tr>
<td>3.5 Measuring Latency</td>
<td>17</td>
</tr>
<tr>
<td>3.6 Performance Improving The Xenomai Cobalt Core</td>
<td>18</td>
</tr>
<tr>
<td>3.7 A Xenomai Yocto Layer</td>
<td>20</td>
</tr>
<tr>
<td>3.8 Excluding Licenses</td>
<td>21</td>
</tr>
<tr>
<td>3.9 The Xenomai Mercury Core With PREEMPT_RT</td>
<td>22</td>
</tr>
<tr>
<td>4 Result</td>
<td>24</td>
</tr>
<tr>
<td>4.1 Xenomai Stressed Cobalt Core</td>
<td>24</td>
</tr>
<tr>
<td>4.2 Xenomai Cobalt vs Mercury Core</td>
<td>26</td>
</tr>
<tr>
<td>5 Discussion</td>
<td>27</td>
</tr>
<tr>
<td>5.1 Latency: cobalt and mercury</td>
<td>27</td>
</tr>
<tr>
<td>5.2 Latency: idle CPU with cobalt core</td>
<td>27</td>
</tr>
<tr>
<td>5.3 Latency: Stressed CPU with cobalt core</td>
<td>28</td>
</tr>
<tr>
<td>5.4 Latency: comparison with other hardware</td>
<td>29</td>
</tr>
<tr>
<td>5.5 Latency: PREEMPT_RT</td>
<td>30</td>
</tr>
<tr>
<td>5.6 Multiple Cores</td>
<td>30</td>
</tr>
<tr>
<td>5.7 Open-source licenses</td>
<td>31</td>
</tr>
<tr>
<td>5.8 The Yocto Layer</td>
<td>31</td>
</tr>
<tr>
<td>6 Conclusion</td>
<td>32</td>
</tr>
<tr>
<td>A Appendix - patch for gpc.c</td>
<td>i</td>
</tr>
<tr>
<td>B Appendix - patch for gpcv2.c</td>
<td>v</td>
</tr>
<tr>
<td>C Appendix - patch for power configuration</td>
<td>ix</td>
</tr>
</tbody>
</table>
CONTENTS

D Appendix - linux-compulab_4.9.11.bbappend xi
E Appendix - xenomai_3.0.8.bb xiii
F Appendix - local.conf xiv
List of Abbreviations

API  Application Programming Interface
ARM  Advanced RISC Machine
BSP  Board Support Packages
CPU  Central Processing Unit
DRAM Dynamic Random Access Memory
eMMC embedded Multi Media Card
gpc general power controller
GPIO General Purpose Input Output
GPL GNU General Public License
IoT Internet of Things
I-Pipe Interrupt Pipeline
IRQ Interrupt Request
MIT Massachusetts Institute of Technology
OS Operating System
POSIX Portable Operating System Interface
RTAI Real-Time Application Interface
RTOS Real-Time Operating System
SDK Software Development Kit
SMP symmetric multiprocessing
SoC System on Chip
SoM System on Module


1 Introduction

Bosch Rexroth in Mellansel manufactures hydraulic drive systems which are used all over the world. These drives are controlled by the Hägglunds Spider which gives total control over the Hägglunds DU drive unit. \[1\] Since the release of this control system, new hardware and software have been developed which could improve the Spider control system. This thesis will analyse one viable SoM for upgrading the Spider.

At the base of any complex computer is an operating system which controls the use of the computer’s hardware. When handling industrial drives the operating system on the control system has to be able to guarantee a given task will be executed in the right amount of time. In order to accomplish this a real-time Linux operating system will be implemented on the new hardware. The RTOS should implement Xenomai in order to be compatible with previous projects.

The hardware analysed in this thesis is a CL-SOM-iMX7 which has a NXP (Freescale) i.MX 7 dual as CPU. \[2\] This processor was released in 2015 meaning tools for developing on the i.MX7D are available and have been tested by software developers. Due to hardware and licensing constraints, the operating system has to be as minimal as possible. This rules out major distributions such as Fedora, Raspbian and Ubuntu as these distributions come with additional packages and might have licensing issues.

During development with the CL-SOM-iMX7, packages could have to be added to the operating system and existing packages updated. Therefore the Yocto Project is used for building the operating system. This project consists of a number of tools which can be used for creating a custom Linux distribution regardless of the hardware architecture due to its’ layered design. It is primarily used by embedded system developers which have unique requirements for their operating systems. Using Yocto will ensure the ability for the company to change hardware, add functionality to the RTOS and maintain an up to date SDK. Furthermore, all tools used by the Yocto project are open-source allowing for transparency of which software is run on the system.

1.1 Business Application

The current control system used by Bosch Rexroth is Spider 2. This unit is configured to each individual drive application using either the display or via a serial connection to an external computer. Since this unit was released in 2001, newer and more advanced hardware and software are available. This means an upgrade of the Spider should be done in order to stay up-to-date with current technology.

Another application for the module is the condition monitoring being developed at Bosch Rexroth. Here an IoT-gateway is needed which can gather the necessary data from the drive units and send this information to a server for analysis.
1.2 Purpose and Goal

The purpose of this thesis is to create a RTOS using Xenomai for the i.MX7D which could be used for the next control system by Bosch Rexroth. If the RTOS is successfully created, the SoM will probably be used for the new condition monitoring of the drive units no matter the latency. However, if the latency is low enough, the module could be used in the next control system of these drive units. Therefore this thesis will answer the question of what latency Xenomai has on an i.MX7D.

The thesis should conclude whether the Yocto Project can be used to build an RTOS on a CL-SOM-iMX7 using an i.MX7D and analyse the performance of the real-time operating system. A latency comparison should be done with other hardware. The partial goals for the thesis are summarized in section 1.3.4.

1.3 Specifications

In collaboration with Bosch Rexroth in Mellansel a specification of requirements for the project was written.

1.3.1 Real-time operating system and benchmarking.

In industrial applications, real-time communication is necessary in order to assure appropriate action is taken in time. The real-time part of the operating system should guarantee time critical tasks will get enough CPU-time in order to complete before deadline. The Spider control system has control and monitoring tasks which are executed every 100 ms. Therefore the deadline of the system is 100 ms. An exact limit for the latency cannot be given as the time required for each task is unknown.

Previous projects done by the company use the Xenomai POSIX interface and Xenomai’s serial driver. In order not to rewrite code, the new SoM should also have Xenomai implemented. Improvement and security updates are rolled out by Xenomai and the Linux kernel continuously, therefore the latest stable version of Xenomai and the Linux kernel which are compatible should be implemented.

Xenomai includes a test suite which can be used to measure latency. The result of this test should be discussed and the real-time capabilities of the board should be analysed using this test.
1.3 Specifications

1.3.2 Yocto

A minimal Linux distribution should be built as to not include packages with a Copyleft license and in order to reduce overhead which will consume CPU-time. Here the Yocto Project should be used so all none hardware specific packages can be used in other builds when the SOM is changed. The final distribution should be built with Yocto and the base distribution for the build should be either Poky or Ångström.

1.3.3 Licenses

Due to this thesis being done in relation to a corporate environment, the system should only use tools with a license favourable to corporate guidelines for open-source. This excludes commercial licenses and GPLv3 (together with its’ varieties). Other Copyleft licenses should also be excluded apart from the GPLv2 as the kernel itself and Xenomai use this license.

1.3.4 Summary

<table>
<thead>
<tr>
<th>Requirement:</th>
<th>Explanation:</th>
<th>Priority:</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTOS</td>
<td>A RTOS should be made for the CL-SOM-iMX7 made with the latest stable Xenomai and kernel versions.</td>
<td>1</td>
</tr>
<tr>
<td>Benchmark</td>
<td>Xenomai’s latency should be benchmarked on the i.MX7D with the testsuite.</td>
<td>2</td>
</tr>
<tr>
<td>Latency</td>
<td>The maximum latency should be less than 100 ms.</td>
<td>3</td>
</tr>
<tr>
<td>Yocto</td>
<td>The RTOS should be built using the Yocto project.</td>
<td>4</td>
</tr>
<tr>
<td>Licenses</td>
<td>Only cooperate friendly licenses should be used in the final image.</td>
<td>5</td>
</tr>
<tr>
<td>Comparison</td>
<td>The latency on the NXP i.MX7D should be compared to another SOM, for example a TI AM4379 or previous hardware used.</td>
<td>6</td>
</tr>
</tbody>
</table>
2 Theory

There are many components used in this thesis which are explained in this section.

2.1 Linux

The Linux operating system is a derivation of the Unix operating system and open-source code written by the GNU Project (and other code contributions by other developers). Since Linus Torvalds released the Linux kernel in 1991 and the Linux operating system was assembled, the OS gained popularity due to the code being open-source. Open-source is when the code of a project is made public and depending on which license the code is released under, anyone can use it. Since Linux is freely available to anyone, an even larger community grew around the open source movement of software with everybody contributing to accelerate the operating system and countless other software projects.

One of the tasks of an operating system is to manage the hardware on which it is installed, such as for example a desktop computer. Two of the biggest operating systems for desktop computers are Windows by Microsoft and MacOS by Apple Inc. For embedded systems, the market looks different as these two operating systems are not open source. This means developers cannot customize the OS to fit their hardware without paying a fee. Here Linux has the advantage. Linux can be customized by anyone making it easier and free for companies to create an OS for their hardware.

There are different components to a Linux operating system, the actual Linux kernel, a bootloader and then other packages. On most hardware there is a small section of code which tells the processor how to start up essential hardware and in turn the entire operating system. This code is called a bootloader. There can be multiple types of bootloaders for each operating system, also depending on which hardware is used. For Linux one common bootloader for embedded systems is Das U-Boot, "the universal bootloader".

After the bootloader starts, the kernel will be started and begins managing the hardware on the system. After years of development the Linux kernel has become a large project for different architectures such as x86 and ARM. This thesis will discuss the ARM architecture considering this architecture is common for embedded systems.

Drivers for different components of the system are included in the kernel. These drivers reside in kernel-space giving them direct access to the hardware of the system. Programs executed on the system using these drivers are instead run in user-space and do therefore not receive uncontrolled access to the hardware.

1The GNU libraries, Linux kernel and other contributions are called "the Linux operating system" or just "Linux" in this thesis. Unless mentioned otherwise.
2.2 Real-time Operating system

Some programs commonly used, such as a terminal, are not part of the kernel yet they are required for a usable operating system. These programs are usually added to the system after the kernel and are commonly installed in the form of packages. Programs can depend on other programs or libraries which should be installed in advance. This can lead to complex dependency trees and are usually resolved using a package manager. However a package manager is not always included on an embedded system. There are different reasons for this, one of which is safety as updating packages can cause crashes of the system. [7]

2.2 Real-time Operating system

Multiple tasks can be executed on an OS even though the hardware can only support one process on the CPU at a time. This is done by dividing up a task and allowing different segments of this task to run on the CPU. When a segment is complete, another task’s segment will be run on the CPU. The segmentation, known as process scheduling, is done by the scheduler in the Linux kernel and will schedule which timeslot a task receives and how long this timeslot is. Processes which are critical for the system have a higher priority and are thus scheduled accordingly. Different schedulers use different algorithms for scheduling CPU-time. The default scheduler in the kernel does not guarantee all processes are treated fairly and some processes will get more CPU-time than others depending on for example hardware resources.[8]

One way of making sure the tasks are executed prior to other tasks is by changing their nice value, which is a type of priority value. [9] A lower nice value gives a higher priority in the Linux scheduler. Therefore hardware related tasks such as fan control have lower nice values. When a lot of tasks are being handled by the CPU, there is a risk that user-space tasks will be delayed longer than allowed. This delay between when some stimulus is received and the corresponding task receives CPU-time is called latency.

There are two different types of real-time systems, hard and soft real-time systems. A hard real-time system is when missed deadlines are considered a system failure. Some application of these types of systems could be the automotive industry or music industry. When the deadline is reached, the consequences could be fatal. [10]

On the other hand there are soft real-time systems. The tasks in a soft real-time system can miss their deadlines. Then the system does not fail when the deadline is exceeded and can continue on afterwards. However the tasks should still hold their deadlines for the system to work properly. During industrial applications it is more important for the Spider to continue on controlling the system than to fail if a deadline is reached. Therefore the Spider is considered to be a soft real-time system.
2.3 Xenomai as a Real-time Operating System

Xenomai is one way of creating real-time Linux and has some commonly used APIs. In order to create a RTOS, Xenomai offers two possibilities. The first one is using the real-time capabilities of the Linux kernel. Depending on the real-time needs of the applications which run on the system, the kernel might have to be patched with the `PREEMPT_RT`. One of the things this patch does is change all the spinlocks of the kernel to reduce latency. This option the Xenomai developers named the `mercury` core and does not have complete support of all the Xenomai options such as kernel drivers. [11]

The second possibility Xenomai offers for real-time is a dual-core configuration, called the `cobalt` core. This core takes over the interrupt handling and the scheduling of the real-time threads. Cobalt has a higher priority than the native kernel meaning the real-time threads can be scheduled to be completed in time. Implementing the `cobalt` core is done using the I-Pipe patch. [12]

Events such as interrupts and system calls are first registered with the Xenomai co-kernel and then Xenomai decides where to dispatch it. If the task is meant to be real-time Xenomai will schedule it, else the task is dispatched to the Linux kernel. If a real-time task were to fail, Xenomai can pass the task on to the Linux kernel allowing for the normal fault handlers to be used. [13]

Real-time tasks can be used with the RTOS API and supports user-space real-time tasks. Due to the dual-core environment, care must be taken when implementing Xenomai that the real-time kernel does not call on any normal kernel code. If this were the case, an unsafe entry could occur which could harm the entire system. [14]

As the Xenomai kernel and the normal Linux kernel are two separate independent kernels, the spinlocks implemented by the regular kernel can be preempted by the Xenomai kernel. This means code which should only be run by one thread (such as some device drivers) could be run by the Xenomai kernel even if the regular kernel holds the lock. As this could cause major faults, the decision was made that a real-time thread is only handled by one kernel at a time. If the thread does not call on any normal kernel system calls, the thread stays real-time. However if the real-time thread were to use normal system calls, the thread is moved over to the normal kernel during that period and the real-time guarantee is forfeited as normal locks would apply. Application developers should therefore develop the code accordingly. [15]

One of the advantages of implementing Xenomai is the possibility to use skins. These skins allow developers who have used other RTOS implementations before, to port their applications to Xenomai and still maintain the same API. Some of these skins are POSIX, RTAI and VxWorks. [16][17][18]

The entire programming interface and additional information can be found in Xenomai’s documentation. [19]
2.4 The Yocto Project

The Yocto Project is an umbrella for a collection of tools used for creating a custom embedded Linux distribution. [20] Some of the components included in a distribution are the bootloader, the kernel and devices drivers. However other things need to be considered such as life cycle management and how software can be developed for the system. The Yocto Project offers tools which can help with all of these steps.

A custom distribution can be based on the default Yocto distribution called Poky and is maintained by the OpenEmbedded community. Different distributions can be used as a base, one of which is the Ångström distribution. [21]

Embedded systems have different hardware solutions, each with their own architecture. A commonly used architecture is ARM which has different types of cores and since ARMv8 a 64-bit alternative for embedded ARM systems is also available. In order to support all these different hardware types, the Yocto project has implemented a layered structure. The top layer can be shared and used for different hardware, whereas the lower hardware layer is changed depending on which hardware is used.

The Yocto framework uses its' own terminology which is explained in their mega-manual. This manual includes all information about the Yocto Project needed for most embedded Linux development. [22]

- Metadata contains the information used to build the distribution. This data includes for example recipes, configuration files and build instructions.

- Recipes contain the settings and instructions for the packages used to build the binary image. This can include where to download the source code from and which patches should be applied. Dependencies are also described here.

- Layer is a collection of recipes which are related to each other. The layers are hierarchical and can be used to customize the distribution.

- OpenEmbedded-Core is essentially meta-data containing classes and files which are common among OpenEmbedded-derived systems. These core recipes are "tightly controlled and quality-assured" by the OpenEmbedded developers.

- Poky is the reference distribution providing the basic functionality of a distribution and can be customized. This is essentially an integration layer on top of the OpenEmbedded-Core.

- Build System - "Bitbake" is the engine which takes care of the scheduling, parsing of the recipes and the creation of the distribution image. The build system creates a dependency tree in order to schedule the build process.
2.5 The Hardware

The chosen hardware for this project is a CL-SOM-IMX7 from Compulab and is mounted on a carrier board. [23] On this module is a NXP (Freescale) i.MX 7Dual ARM Cortex-A7 at 1GHz together with an ARM Cortex-M4 co-processor at 200MHz. [24] A DRAM of 1GB is included and an eMMC of 16GB.

Both processors have ARM Cortex cores. The main processors are based on the Cortex-A series, which is designed by ARM as a power-efficient high performance core. ARM’s A7 can host an operating system with multiple complex tasks meaning it can be used for a number of applications. The architecture of this core is the ARMv7-A. [25]

The co-processor is based on the ARM Cortex-M4, which is meant as a low cost and low power signal controller in embedded devices. ARMv7E-M Harvard is the architecture chosen for the Cortex-M4. In the i.MX7D, this core is used for real-time signal handling. However this thesis only focuses on the Linux real-time latency of the CPU and therefore the M4 core is not taken into account. [26]

2.6 Toolchain for cross-compilation

A toolchain is a set of tools used to create an executable binary file. When a toolchain is used to create executable binary files for another architecture than the host system, the toolchain is called a "cross development toolchain". An example for when these toolchains are required is when building a distribution for an i.MX7D as the host is a amd64 and the SoM is an ARMv7-A.

A compiler is not the only part of a toolchain as it includes libraries, an assembler, a linker and some other tools as well. The compiler for C code includes a preprocessor which for example removes comments and replaces macros. This tool does not compile the code and only adds/removes C code. However some macros are platform dependent and therefore the preprocessor also has to be compatible with the architecture for which you want to compile.

The actual C compiler is also platform dependent as it translates the C code to assembler code, which can be different for every machine. This assembler code is later translated into a binary object by the assembler. Here variables are listed in the object file which can be shared to other object or when they have to be included from other objects. These objects are later linked together by the linker. Here external objects, such as from a C libraries, are also linked into the program to create the final executable file. [27]

All of the above mentioned steps are platform-specific and therefore require a toolchain tailored to the target platform. These tools can be taken from different sources as long as they are able to work together. All the tools must be able to work together forming
a chain which the source code is passed through.

Additionally, the libraries linked into the source code needed by the program must be compiled to object files for the correct platform in advance. Without dynamically linked libraries, everything has to either be built with the entire source code or the program will simply not work. In this paper, this is done using the Yocto Project.

2.7 Related Literature

Real-time operating systems are a must for some applications and have been researched and implemented by different actors from open-source communities, to private companies, to universities. As different tools can be implemented for creating a RTOS a need arose for a comparison of the different systems. A comparison was made by Huang, C.-C. et al. and found Xenomai 3 having shorter latency times than for example a preempt_rt patched kernel. [49] Another such comparison was made by the Brown and Martin. [50] They researched the best real-time implementation depending on the real-time needs of a system.

Brown and Martin use an external method of measuring the latency. This is done to verify the time objectively as tools provided by the real-time implementation might improve their own times. This method was also used by Gustav Johansson for measuring the latency of Xenomai on a Raspberry Pi 3B. [48] The results from the related literature mentioned here will be compared with the results in this thesis.

When testing a real-time system, the CPU is sometimes stressed in order to simulate the applications which will be run by the system. In order to simulate this stress, the previous mentioned papers use the stress program. [31] However Xenomai uses its’ own script to stress the CPU called dohell. In a thesis written by Andréas Hallberg, the two ways of stressing the CPU are used when measuring real-time tasks. [32] Hallberg found periodic latency peaks when using dohell, but not with stress. These peaks were explained as being caused by the $ls -lR /$ command used for simulation a load in the dohell script. The mean latency for the real-time tasks is also higher when running dohell instead of stress, with a difference of 132 microseconds (13%).

2.8 Open-source licences

According to the Open Source Initiative software is open source if ten conditions are complied with. These ten conditions are as follows, [33]

1. Free Redistribution
2. Source Code Distribution
2.8 Open-source licences

3. Allow Derived Works
4. Integrity of The Author’s Source Code
5. No Discrimination Against Persons or Groups
6. No Discrimination Against Fields of Endeavor
7. Distribution of License
8. License Must Not Be Specific to a Product
9. License Must Not Restrict Other Software
10. License Must Be Technology-Neutral

This definition allows for different licenses to be classified as open source licenses. Within the open source licenses, there are two different categories as well, copyleft and permissive. Copyleft are licenses were the derived work also has to be published under the same license. This means the license forces programs using the open-source software to also be open-sourced. One such license is the GNU General Public License. [34] Permissive licenses do not require derivative software to be published as open source. The MIT license is an example of a permissive license which allows for the open source code to be used and compiled together with closed source code. [35] However the open source code used has to be distributed together with the closed source program and the MIT license.

Permissive licenses are used during this thesis as using code under this license will not require future work by the company to be open-sourced. However some critical components are licensed under the GPLv2, such as the Linux kernel which force this licence to be allowed. The later version of this license, GPLv3, has some changes specific for embedded systems in order to for example prevent "Tivoization". [36] As this thesis is done in cooperation with a commercial company, the decision was made to not include Copyleft licenses other than the GPLv2.
3 Method

The first step in creating the RTOS for the i.MX7D is trying to create the demonstration image of the SoM supplier. This step confirms that the Yocto build system works as described by the developers and will show how Compulab made their Yocto layer.

After the demonstration image is built and successfully booted on the hardware, a minimal image is built without packages such as a desktop environment. These types of packages only increase build-time and the size of the image. In addition, these packages might increase the latency of the system with unnecessary tasks. The core-image-minimal defined in the Poky layers only builds with the necessary packages for an image to boot.

After the minimal image is built, the I-Pipe patch is altered to be compatible with the kernel used by the Compulab layer. Then Xenomai is implemented and booted on the hardware.

When Xenomai is implemented in the minimal image and booted successfully on the hardware, latency tests can be executed.

3.1 Building images for the CL-SOM-iMX7 using Yocto

The company Compulab has created Yocto layers for their CL-SOM-iMX7. These layers can be used for creating a distribution and depend on the Yocto layers from NXP. Compulab has instructions on their website on how to set-up the build environment. A short version of these instructions is given in this section. The build is done on a Ubuntu 16.04 amd64 host computer and the packages required by Yocto are installed using the following command.

```bash
$ sudo apt-get install gawk wget git-core diffstat unzip \
texinfo gcc-multilib build-essential chrpath socat cpio \
python python3-pip python3-pexpect xz-utils \
debianutils iputils-ping libstdc++-dev xterm
```

First the NXP building environment has to be downloaded. This is done by using the repo tool developed by Google to make the use of Git easier. [37][38] The repo tool can be installed using the following commands.

```bash
$ mkdir -/bin
$ curl \ 
  http://commondatastorage.googleapis.com/git-repo-downloads/repo
```
3.1 Building images for the CL-SOM-iMX7 using Yocto

```bash
> ~/bin/repo
$ chmod a+x ~/bin/repo
$ export PATH=~/bin
```

When `repo` is downloaded, it in turn can be used for downloading the NXP build environment.

```bash
$ repo init -u \
  git://source.codeaurora.org/external/imx/imx-manifest.git \
  -b imx-linux-rocko -m imx-4.9.88-2.0.0_ga.xml
$ repo sync
```

Now the Compulab BSP meta-layer can be added to the `sources` folder which was created when setting up the NXP build environment. There is a slight difference between the versions here. The NXP distribution layer is on version 4.9.88, while the Compulab layer is on 4.9.11. However this has not been shown to be an issue and in a message conversation with Compulab, their intent to release a 4.9.88 version within six months was confirmed.²

```bash
$ git clone -b master \
  https://github.com/compulab-yokneam/meta-compulab-bsp.git \
  sources/meta-compulab-bsp
```

When the Compulab BSP layer is downloaded, the following variables are exported in accordance with the Compulab instructions.

```bash
$ export DISTRO=fsl-imx-x11
$ export MACHINE=cl-som-imx7
$ BUILD_DIR=build-x11
```

Afterwards the set-up scripts are sourced.

```bash
$ source fsl-setup-release.sh -b ${BUILD_DIR}
$ source ../sources/meta-compulab-bsp/tools/setup-compulab-env
```

²The question was asked in April 2019
3.2 Early debugging

When the scripts have been sourced, the build environment has been set-up and all variables have been added to the terminal’s PATH variable. This means the `bitbake` command can be used to build the distribution image together with other commands. Different images are supported by the included layers. The image recommended for testing the hardware is the `fsl-image-validation-imx` image. Building this image can be done using the following command.

```
$ bitbake fsl-image-validation-imx
```

The image used during this project is the `core-image-minimal`. This is an image defined in the layers of the Poky distribution and only compiles the absolutely necessary packages needed for creating a bootable image.

### 3.2 Early debugging

When the kernel starts, some tasks are performed before the kernel activates the console. This means that when the kernel has a panic before the console is started, no prints will be sent to for example a serial connection or printed on a screen. In order to debug the kernel before this point, a special configuration has to be enabled when the kernel is configured. These prints are only used during the debug phase and should be turned off to improve boot time afterwards.

In the kernel source code, there are print functions before the console is started. The print functions are called `early printk` and getting these prints requires two steps. First the kernel has to be configured so the early printk are compiled into the kernel. Where this information is sent also needs to be configured as shown in figure 1.

![Figure 1](image1.png)

**Figure 1** – The options selected in order to get early debugging information from the kernel on an iMX7D.

The kernel can be configured using Yocto with the following command.

```
$ bitbake -c menuconfig virtual/kernel
```
3.3 Xenomai’s Cobalt Core on an i.MX7D

If an image is made with Bitbake, the packages other than the kernel will not have to be downloaded again when only the kernel is altered. The kernel used by Bitbake is defined in the Compulab layers as a vendor kernel maintained by NXP instead of the mainline Linux kernel maintained by Linus Torvalds.

In addition to a vendor kernel, Compulab also applies some patches to this kernel for better support for their hardware. As the I-Pipe patch is derived from the mainline kernel, there might be some incompatibilities with these patches and the I-Pipe patch. Therefore it is recommended to start with a clean kernel when applying the I-Pipe. The i.MX vendor kernel can be found in the `tmp/work-shared/cl-som-imx7/kernel-source` folder after the following commands. This kernel is based on the mainline version of 4.9.11 and maintained by NXP.

```
$ bitbake -c cleansstate virtual/kernel
$ bitbake -c do_unpack virtual/kernel
```

The I-Pipe patches should be applied to the same kernel version as there might be differences in the kernel which can increase latency or the kernel is so different that the patch cannot be applied. There is however no I-Pipe patch which has the same minor version as the vendor kernel. The closest version is the `ipipe-core-4.9.24-arm-2.patch`. This patch can be applied to a mainline kernel of version 4.9.11, if a fuzz level of 3 is given to the `patch` command. A fuzz level dictates how different the code can be in order for the patch to still be applied.

However, the patch cannot be applied directly to the vendor kernel without alteration. One file has been altered by the NXP developers in places required by the I-Pipe and therefore the patch does not recognize the code of the kernel. The file is named `gppc.c` and has to be changed manually. Changes made to the file in order to be compatible with the I-Pipe can be found in appendix A.

When booting into the kernel after these changes, the kernel will not boot. In order to receive more information early debugging has to be enabled which is described in section 3.2. These prints reveal a kernel panic in functions of the `gpcc.c` file. On the Xenomai wiki pages is a guide for how to port the I-Pipe to new SoC. All the steps in this guide were checked and every topic has been altered by the I-Pipe patch after some changes. These topics are the hardware timer, high resolution counter and the interrupt controller. [39]

When checking the `gpcc` file, a `gpccv2.c` was found which is not in the mainline kernel. The documentation describes this file as only being for the i.MX7D. There is no big difference between the essential functions which need to be altered in `gpcc.c` and `gpccv2.c`, which is why the I-Pipe changes can be ported to this file almost directly. The changes can be
3.4 Creating the SDK for Cross-Compilation

found in appendix B.

With these changes to the I-Pipe patch for both gpc files, Xenomai can be added to the vendor kernel. Without creating a Yocto layer, Xenomai can be implemented using the following commands. The first command opens a shell in the kernel-source folder and the second command calls on a Xenomai script to implement itself on the given kernel.

```bash
$ bitbake -c devshell virtual/kernel
$ {path_to_xenomai}/scripts/prepare-kernel.sh \ 
  --ipipe={path_to_ipipe_with_imx_changes} --arch=arm
```

After calling the `prepare-kernel.sh` script, the kernel is automatically configured with Xenomai and the I-Pipe. Both Xenomai and the I-Pipe can be turned off and the kernel can be compiled without them if required.

According to the Xenomai developers, Xenomai will print kernel messages if the patched kernel is booted up successfully. These messages can be seen using the `dmesg` and `grep` commands. The result of these commands is shown in figure 2.

![Figure 2 - Kernel prints showing Xenomai is active on the CL-SOM-IMX7.](image)

3.4 Creating the SDK for Cross-Compilation

In order to measure the latency with Xenomai’s test suite, the tools have to be cross-compiled for the i.MX7D. A toolchain is needed for cross-compiling to an embedded system from an amd64 host. Using the Yocto Project, a SDK including a toolchain can be built with the `bitbake` command below.

```bash
$ bitbake meta-toolchain
```

When this build is done, an installation script for the SDK is created in the `tmp/deploy/sdk` directory. Inside of this directory there is a script to install the SDK which includes the
3.5 Measuring Latency

toolchain. Alongside the SDK directory, is the \texttt{deb} directory. Here are all the \texttt{.deb} files stored which were used to create the SDK. The SDK built by Yocto is larger than just a simple toolchain as there are many different packages ranging from compilers, to locale settings, to scripts.

Bitbake creates an installation script which will check different settings such as if the current host system is compatible with the toolchain. A location needs to be given to where the toolchain will be installed. The environment script need to be run once before the \texttt{$PATH$} variable is updated with the location of the toolchain. This is done with the commands below.

\begin{verbatim}
$ source /opt/fs1-imx-x11/4.9.88-2.0.0/environment-setup-
  cortexa7hf-neon-poky-linux-gnuabi
\end{verbatim}

3.5 Measuring Latency

An RTOS is implemented in order to improve latency and decrease the time needed before executing specific tasks. This time is usually measured in microseconds and there are different ways of measuring this delay. Xenomai has their own tests which measures the latency on the device it is executed on.

In order to install these test-programs, Xenomai must be configured with the toolchain used for compiling programs on the specific embedded device. When Xenomai is downloaded as a compressed file, this configuration has already been done. However not with the correct toolchain for a distribution build with the Yocto Project. Therefore Xenomai has to be reconfigured with the correct toolchain. Assuming the toolchain has been installed and all variables are sourced, the command below will configure Xenomai correctly for an i.MX7D.

\begin{verbatim}
$ ../xenomai-3.0.8/configure --with-core=cobalt --enable-smp
  --host=arm-poky-linux-gnuabi CFLAGS="-march=armv7ve -mfpu=neon\n  -mfloat-abi=hard -mcpu=cortex-a7" LDFLAGS="-march=armv7ve \n  -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a7"
\end{verbatim}

Apart from which compiler should be used for compilation, two other options are passed to the configuration script. The first flag \texttt{--with-core=cobalt} is which core to use and can be either \texttt{cobalt} or \texttt{mercury}.

The other option passed to the configuration script is whether or not to enable SMP. This reduces the required memory as the cores share memory. [40] Another advantage is
that this configuration should improve performance when using multiple cores.

All the tests and libraries for Xenomai can be installed to the given root filesystem using the following command.

```
$ make DESTDIR={build_path}/tmp/work/cl_som_imx7-poky-linux-\
gnueabi/core-image-minimal/1.0-r0/rootfs install
```

The latency program is one of the tools used for measuring the latency of a task. This is done by getting the time when the thread is created and when the thread is scheduled. Afterwards the time difference is compared.

In order to simulate a load on the CPU, the dohell script uses four tasks.

- `cat /proc/interrupts`
- `ps w`
- `dd if=/dev/zero of=/dev/null`
- `ls -IR /`

These commands are general tasks and in order to get an accurate latency measurement the latency program should be executed together with the real load on the system. After the program and latency are run together for about a week, all the latency peaks should be shown.

3.6 Performance Improving The Xenomai Cobalt Core

Embedded systems can have different applications and some of these applications are battery-driven. In order to increase the lifetime of these devices, different power-saving functionality has been added to the Linux kernel. This functionality does increase latencies, which is the main priority for a RTOS to reduce. Therefore different kernel configurations have to be turned off in addition to some other configurations. These configurations are summarized in table 1.

Xenomai urges developers to turn off CPU frequency scaling which allows the CPU frequency to be changed during run-time. A higher frequency allows the kernel to operate faster, however this also increases the power consumption of the device. When the CPU frequency is changed during runtime Xenomai’s timing can be affected. Additionally, when the frequency is changed to something higher, it can take many cycles before the CPU has reached full speed. [41]
When the CPU is in an idle mode, it will take a few cycles before the CPU can process the interrupt that woke it up. This latency increases the general latency for the CPU, which in turn affects the real-time tasks. Therefore it is recommended to turn off the configuration which allows the kernel to enter a low-power state. The CPU\_idle configuration allows the CPU to enter a deep sleep mode and should be turned off. In addition, the SUSPEND configuration allows for a sleep mode to be entered were the memory is still powered, however waking up from this mode will also affect latency.\[^{42}\]

In order to turn these configurations off, the kernel has to be patched due to some bugs. Bugs such as a structure being defined when CONFIG\_CPU\_FREQ is defined, but the structure still being used even if this configuration is turned off, cause errors when compiling the kernel. The changes made to the kernel are shown in appendix C and are not intended for any other CPU than the i.MX7D. In addition, the CONFIG\_PM also has to be turned off for the patch to work.

An additional recommendation by the Xenomai developers is to turn off the page migration used by the Linux kernel. This allows the kernel to move physical pages in the memory closer to the processor which accesses this memory. The processor will not notice this as the virtual address for the memory is still the same, however the move could increase latency.\[^{43}\] When this happens, the real-time kernel might still have the old addresses which will cause page faults and in turn a higher latency for the real-time task as it checks for the correct address.\[^{44}\]

In general, debugging any process will slow it down as data has to be collected and printed to a destination. Therefore the debugging of Xenomai and the I-Pipe should be turned off when the main goal for the system is performance. However when an error occurs, there will be no possibility to see debugging prints for information about what happened where.

The frequency of when the hardware timer interrupts the kernel can affect the latency of the kernel. During this wake-up, the kernel does internal time management. This configuration also sets an upper bound for the kernels internal timers. Therefore it is argued this configuration will increase effectiveness while increasing power consumption. The configuration variable for this is CONFIG\_HZ and has been set to 1000Hz as reducing power consumption is not as important as reducing latency for this control system.\[^{45}\]
3.7 A Xenomai Yocto Layer

Table 1 - Kernel configurations which have to be turned off to reduce latency of the real-time tasks.

<table>
<thead>
<tr>
<th>Config</th>
<th>Explanation</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_FREQ</td>
<td>CPU Frequency scaling allows you to change the clock speed during run-time.</td>
<td>Critical</td>
</tr>
<tr>
<td>CPU_IDLE</td>
<td>CPU idle is a generic framework for supporting software-controlled idle processor power management.</td>
<td>Critical</td>
</tr>
<tr>
<td>SUSPEND</td>
<td>Allow the system to enter sleep states in which main memory is powered and thus its contents are preserved.</td>
<td>High</td>
</tr>
<tr>
<td>CMA</td>
<td>Contiguous Memory Allocator</td>
<td>High</td>
</tr>
<tr>
<td>COMPACtion</td>
<td>Allow for memory compaction.</td>
<td>High</td>
</tr>
<tr>
<td>MIGRATION</td>
<td>Allow memory page migration.</td>
<td>High</td>
</tr>
<tr>
<td>IPIPE_DEBUG</td>
<td>Allow I-pipe debugging.</td>
<td>low</td>
</tr>
<tr>
<td>DRM</td>
<td>Kernel-level support for the Direct Rendering Infrastructure.</td>
<td>low</td>
</tr>
<tr>
<td>IMX_IPUV3_CORE</td>
<td>Image Processing Unit for i.MX5/6.</td>
<td>low</td>
</tr>
<tr>
<td>FTRACE</td>
<td>Kernel tracing infrastructure.</td>
<td>low</td>
</tr>
<tr>
<td>Lock Debugging</td>
<td>All configurations under this section are turned off.</td>
<td>low</td>
</tr>
<tr>
<td>STACKTRACE</td>
<td>This option causes the kernel to create a /proc/pid/stack for every process.</td>
<td>low</td>
</tr>
<tr>
<td>PCI</td>
<td>The PCI should not be needed for this processor.</td>
<td>low</td>
</tr>
</tbody>
</table>

3.7 A Xenomai Yocto Layer

With the Yocto Project, all the previous steps of adding Xenomai to an image can be automated. Two main recipes are needed. One for adding the cobalt kernel and another for adding the Xenomai libraries together with the tests. A layer requires a special layout so it can be parsed by bitbake and this layer is presented in figure 3. This layer contains two recipes, one for the kernel and one for the Xenomai libraries and program. This layer used for both the Xenomai Mercury core and the cobalt core can be found on the authors Github. ^3

^3https://github.com/braco/meta-xenomai-imx7d
Xenomai is added by this layer in the form of a `bbappend` which essentially appends the file to the original `bb` file. This means the original Compulab kernel file is extended with what is in the `bbappend`. In the append file one additional task is added, the `do_add_xenomai`. Here Xenomai is downloaded and the `prepare_kernel.sh` script is executed as mentioned in section 3.3 with the I-Pipe patch from the layer.

Due to some bugs in the kernel when the power management is turned off, a patch to fix this is needed and presented in appendix C. This patch is found in the meta-layer and is added to the kernel together with the other patches from Compulab. Another file in this recipe is the configuration file made with the steps in section 3.6.

The Xenomai libraries are added separately from the `cobalt` kernel. Instead of being added in the `bbappend`, these libraries are added using packages. The recipe creates three packages, one normal `xenomai` package with the required libraries and tests for Xenomai to work. Another `xenomai-dev` package is created with the header files and other shared libraries. The last package is `xenomai-demo` which contains demonstration programs.

When building the kernel with a cobalt core, the `xenomai` package should be added so that the required libraries for real-time applications are installed. If the image is to be used as a development image, the `xenomai-dev` should be installed.

### 3.8 Excluding Licenses

Excluding licenses can be done with the Yocto Project using the `INCOMPATIBLE_LICENSE` variable in the `local.conf` file. All layers have a variable with the license for the layer.
3.9 The Xenomai Mercury Core With PREEMPT_RT

These licenses are parsed and checked if they are compatible with the configuration of the image. In order to exclude the GPLv3, the following line was added to local.conf.

```
INCOMPATIBLE_LICENSE = "GPL−3.0 LGPL−3.0 AGPL−3.0"
```

Due to the GPLv3 being written by the GNU project, most of the software tools under the GNU project use this license. This means packages such as readline, which are required for a minimal-base-image, will not be used in the image. In order to still be able to compile an image without the necessary packages which use GPLv3, a special layer was made by the OpenEmbedded developers. The meta-gplv2 contains older versions of necessary packages before they were updated to the GPLv3. [46]

During the build, commercial licenses were excluded as to not include packages with permissive licenses other than open-source licenses. This is done by not whitelisting these types of licenses. When the NXP build environment is setup, the default is that commercial licenses are whitelisted. This is changed by removing the LICENSE_FLAGS_WHITELIST from the local configuration as shown below.

```
#LICENSE_FLAGS_WHITELIST = "commercial"
```

3.9 The Xenomai Mercury Core With PREEMPT_RT

Since Xenomai 3, the option to have a single kernel configuration has been available. This essentially offers the Xenomai API while only using the real-time capabilities of the current system. This could be either an unchanged mainline kernel, or a kernel patch with for example the PREEMPT_RT patch. As the mainline kernel has unbounded latency, the PREEMPT_RT patch was applied to the i.MX vendor version of the Linux kernel 4.9.11.

The patch can be downloaded from their website and can be applied directly to the kernel with the patch command. [47] No changes to the kernel code are required for this patch to work together with the i.MX vendor kernel and the Compulab patches.

As Xenomai does not state any special configurations for the mercury core, the general kernel configurations which affect latency listed in table 2 were turned off. The mercury core depends on the systems latency and therefore, the kernel should also be configured for the PREEMPT_RT patch. In order to do this, the PREEMPT_RT_FULL has to be turned on. Another option that can be used instead is the PREEMPT_RTB which is the lighter real-time version. After this, the code for the PREEMPT_RT patch should be compiled when the kernel is compiled.
Table 2 – Kernel configurations which have to be turned off to reduce latency of the real-time tasks for the mercury core.

<table>
<thead>
<tr>
<th>Config</th>
<th>Explanation</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_FREQ</td>
<td>CPU Frequency scaling allows you to change the clock speed during runtime.</td>
<td>Critical</td>
</tr>
<tr>
<td>CPU_IDLE</td>
<td>CPU idle is a generic framework for supporting software-controlled idle processor power management.</td>
<td>Critical</td>
</tr>
<tr>
<td>SUSPEND</td>
<td>Allow the system to enter sleep states in which main memory is powered and thus its contents are preserved.</td>
<td>High</td>
</tr>
<tr>
<td>CMA</td>
<td>Contiguous Memory Allocator</td>
<td>High</td>
</tr>
<tr>
<td>COMPACTION</td>
<td>Allow for memory compaction.</td>
<td>High</td>
</tr>
<tr>
<td>MIGRATION</td>
<td>Allow memory page migration.</td>
<td>High</td>
</tr>
<tr>
<td>DRM</td>
<td>Kernel-level support for the Direct Rendering Infrastructure.</td>
<td>low</td>
</tr>
<tr>
<td>IMX_IPUV3_CORE</td>
<td>Image Processing Unit for i.MX5/6.</td>
<td>low</td>
</tr>
<tr>
<td>FTRACE</td>
<td>Kernel tracing infrastructure.</td>
<td>low</td>
</tr>
<tr>
<td>Lock Debugging</td>
<td>All configurations under this section are turned off.</td>
<td>low</td>
</tr>
<tr>
<td>STACKTRACE</td>
<td>This option causes the kernel to create a /proc/pid/stack for every process.</td>
<td>low</td>
</tr>
<tr>
<td>PCI</td>
<td>The PCI should not be needed for this processor.</td>
<td>low</td>
</tr>
</tbody>
</table>

The installation process for the libraries and test programs is the same for the mercury core and the cobalt core. Which core Xenomai should be configured for is indicated with `--with-core=mercury`. Some tests such as `xeno-test` are not included as they require components from the cobalt core. The `dohell` script is not included either.
4 Result

The results for Xenomai’s latency tests are shown in this section. Two different tests were carried out for two hours. The first test is a benchmark of the latency for the cobalt core with the dohell load, while the second test is a comparison between the cobalt and mercury core.

A latency graph for a kernel without real-time implemented is not presented in this thesis as Xenomai’s latency program fails when the latency of the system is too high. Furthermore, the graphs are only plotted correctly when the latency is less than $300 \mu s$ and a non-RTOS can have above a few milliseconds.

4.1 Xenomai Stressed Cobalt Core

Figure 4 is logarithmic graph showing the latencies of the user-space, kernel-space and interrupt real-time tasks. The minimum, average and maximum latency is shown in table 3. Here the IRQ tasks are clearly scheduled the fastest and therefore have the lowest latency. Afterwards, the kernel tasks have the lowest latency. The user-space tasks have the highest latency.
4.1 Xenomai Stressed Cobalt Core

Table 3 shows a negative time. When this happens, Xenomai developers recommend to recalibrate Xenomai so the latency does not turn negative. However as the system has been calibrated using the autotune program from the Xenomai project, the system was not recalibrated a third time. When the negative value first occurred the system was recalibrated and the test was rerun and gave the current results.

**Table 3** – The minimum, average and maximum time of the latency tests run for 2 hours with dohell load active.

<table>
<thead>
<tr>
<th>Type</th>
<th>Minimum ($\mu$s)</th>
<th>Average ($\mu$s)</th>
<th>Maximum ($\mu$s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>user-space</td>
<td>7.500</td>
<td>39.869</td>
<td>268.250</td>
</tr>
<tr>
<td>kernel-space</td>
<td>0.625</td>
<td>21.153</td>
<td>204.125</td>
</tr>
<tr>
<td>interrupt</td>
<td>-0.750</td>
<td>6.528</td>
<td>76.500</td>
</tr>
</tbody>
</table>
4.2 Xenomai Cobalt vs Mercury Core

When testing the mercury core, high latency was measured during the idle CPU test. The latency command saves the values for the graphs shown in this section. However, after the mercury tests it was found this command only saves the latency up to 300µs. The measured latency for the mercury core is higher than shown in figure 5. Table 4 shows the result of the test in numbers. There the maximum latency for mercury is 412.375µs.

Table 4 – Key numbers from the latency measurements on an idle CPU. Both cobalt and mercury have been tested.

<table>
<thead>
<tr>
<th>Type</th>
<th>Minimum (µs)</th>
<th>Average (µs)</th>
<th>Maximum (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cobalt</td>
<td>0.500</td>
<td>2.826</td>
<td>155.750</td>
</tr>
<tr>
<td>Mercury</td>
<td>36.875</td>
<td>46.537</td>
<td>412.375</td>
</tr>
</tbody>
</table>

Figure 5 – The latency on an idle CPU measured with both the cobalt and mercury core. The increase around 300µs could be because all values after this time are saved as 300µs by the latency program.
5 Discussion

A comparison to other papers is done in this section and the results of the latency in this thesis could be considered as high. Some reasons as to why are discussed here however further research is needed to establish the exact cause of the high latency.

5.1 Latency: cobalt and mercury

When the CPU is idle, most tasks have a low latency compared to when the CPU is loaded. The latency peak of the cobalt kernel on an idle CPU is close to zero, while the peak for the mercury kernel is closer to 50µs. Figure 5 shows both peaks have a tail of tasks which took longer to schedule.

The difference between the maximum values of the cobalt core and the mercury core shown in table 4 is 256.625µs. This is a significant difference as the maximum latency for the cobalt core is only 155.750µs. The PREEMPT_RT patch only allows for user-space threads, therefore there are no measurement with a mercury core for IRQ and kernel-space tasks in this report.

As latency increases when the CPU is loaded, the conclusion can be drawn the Xenomai cobalt core has better real-time properties than the Xenomai mercury core with the PREEMPT_RT patch. Nevertheless, depending on how the software of the control system loads the CPU, the mercury core could still be used for the new control system.

5.2 Latency: idle CPU with cobalt core

The cobalt core has a low latency when the CPU is idle. However after the peak shown in figure 5, there are tasks with a latency between 25µs and 50 µs which break this trend. The steady decline continues again after 50 µs. No explanation why these tasks have a higher latency has been found. The processes on the CPU were examined with the top command and no processes loading the CPU were found which could explain this increase latency. The interruption also occurs just before 50 µs, which also is where the latency peak can be found for a loaded CPU.

One possible solution is adding the PREEMPT_RT patch to the cobalt core. The I-Pipe patch does not change all the spinlocks in the kernel. Since the real-time thread sometimes is moved to the normal kernel, the spinlocks in the normal kernel can have a high latency affect as they cannot be preempted. A spinlock is when a thread blocks another thread from accessing the locked resource and the blocked thread is "just spinning". Blocking the other thread can be better for the overall latency of the system than having to reschedule multiple times. However, when the blocked thread needs to be real-time,
these spinlocks can be a problem as they have to wait for a lower priority thread. The \texttt{PREEMPT\_RT} patch makes these spinlocks preemptable by replaces them with a \texttt{rt\_mutex}. Therefore, the latency could be improved if both the I-Pipe and the \texttt{PREEMPT\_RT} patch are applied.

The overall latency of the Linux kernel might be increased, however the latency of high priority tasks will be decreased when the spinlocks are preempted. The main task of a RTOS is to reduce the maximum latency of tasks and therefore applying both should be tested. However, the I-Pipe patch and the \texttt{PREEMPT\_RT} patch have to be rewritten for both to be applied to the i.MX vendor kernel.

Even if the \texttt{mercury} core also has a decrease in decline after the latency peak, the solution could still work. Nevertheless, more research is needed as it is not sure combining the I-Pipe and the \texttt{PREEMPT\_RT} patch will drastically improve the latency compared to just the I-Pipe patch.

5.3 Latency: Stressed CPU with cobalt core

The interesting latency for a control system is the latency when the CPU is stressed. When the i.MX7D will be used for the new Spider control system, the CPU will always be executing different and sometimes complex programs. Then the latency has to be short enough so the monitoring and control tasks can be executed every 100 ms.

Three different types of latency are measured with Xenomai's \texttt{latency} program and the results are shown in figure 4. IRQ latency, kernel-space latency and user-space latency. The monitoring and control tasks are run in user-space meaning the highest latency measured for these tasks, as shown in table 3, is 268.250\(\mu s\). This latency should leave enough time for the monitoring and control tasks to be executed every 100 ms.

None of the three curves shown in figure 4 have outlier latency peaks which are not connected to the curve. Therefore it is assumed the no latency peaks will suddenly occur with an extremely high latency. Not having these outlier latency peaks is typical for an RTOS meaning the \texttt{cobalt} core should have been implemented correctly.

In order to guarantee the latency of the monitoring and control tasks, the \texttt{xeno-test} should be run with the software of the entire control system running. This will measure the latency with the \texttt{dohall} load active together with the running system. If this test is done for one week with the highest load, the latency should be able to be guaranteed to be less than the maximum latency of the test.
5.4 Latency: comparison with other hardware

As the CM-T43 requires an older version of the Yocto Project which uses a different layer structure, the Xenomai layer was not ported to the CM-T43. The i.MX7D is only compared to the results of other white-papers mentioned in section 2.7.

Johansson G. measured the latency of a Raspberry Pi 3B "overnight" which resulted in a minimum latency of $-5.98 \mu s$, an average of $6.23 \mu s$ and a maximum latency of $82.00 \mu s$. This latency test was done with stress-ng active and Xenomai’s latency test. [48] It is assumed the time measured is for user-space tasks. During this test, Xenomai was probably not configured correctly as the minimum latency is negative with a magnitude almost as large as the average latency for the test. The Raspberry Pi 3B supports a 4 core 1.2GHz ARM Cortex-A53 64-bit architecture. This is faster than the i.MX7D’s CPU which could explain the lower latency on the Raspberry Pi 3B.

Another white-paper by Huang C.-C. used a BeagleBone Black with a single 1GHz ARM Cortex-A8 CPU processor. They also measured the latency using Xenomai and stress-ng. Here the loaded BeagleBone showed a minimum latency of $8.296 \mu s$, an average of $8.853 \mu s$ and a maximum of $33.023 \mu s$. [49] These values show the latency on the BeagleBone Black are also lower than on the i.MX7D.

The last white-paper for comparison is written by Brown H. J. and Martin B. in which they measured the latency of a BeagleBoard Rev C4 OMAP3. The measurement was done by sending a stimulus to the board and measuring how long it takes for a GPIO pin to be toggled. [50] The latency measured in this thesis is therefore both the scheduling latency and the latency for toggling a GPIO pin on the board. The authors measured a minimum of $26 \mu s$, an average of $59 \mu s$ and a maximum latency of $90 \mu s$. These results were without a load meaning the average latency on the i.MX7D is lower, however the maximum latency is higher.

The latency on the i.MX7D is high compared to the latency on other devices, sometimes more than $200 \mu s$. This could have multiple reasons. The I-Pipe patch and the PREEMPT_RT patch could have to be optimized. The kernel of the real-time system could be missing a configuration which will reduce the latency. Additionally, the previous mentioned white-papers use the mainline kernel while during this thesis the i.MX vendor kernel is used. This vendor kernel could also have a higher latency than the mainline kernel.

The previous mentioned hardware types, Raspberry Pi and BeagleBone have a large community who have implemented Xenomai and debugged it. For the i.MX7D no community was found for comparing latency and debugging code.
5.5 Latency: PREEMPT_RT

The latency of the PREEMPT_RT patch is measured using Xenomai’s latency test. As these projects have different developers, there could be some incompatibility with the patch and Xenomai’s latency test. Therefore the latency of the patch might be better than shown in this thesis. However due to the Xenomai’s API being used for the business application of this project, the latency of Xenomai’s test should still be the latency for a Xenomai task. Therefore no other latency test is compiled and tested.

I order to really benchmark the performance of the i.MX7D, multiple independent latency tests should be done. Other tests can be found in the rt-test package. These tests should be independent of both the PREEMPT_RT patch and Xenomai. [51]

5.6 Multiple Cores

The i.MX7D has a dual core and SMP enabled. This allows for the CPU load to be divided by the two cores. Doing this can cause deadlocks and race conditions if the kernel is not properly thread protected. Using two cores does however increase the performance when the CPU is loaded as the load can be shared.

Adding Xenomai to a multi-core CPU is possible as shown in this paper. Nonetheless, no reference to the latency for a i.MX7D can be found in the Xenomai mailing lists. There is however a thread on the mailing lists about high writing latency for the i.MX6Q which has 4 cores. Here Gerum P., one of the developers of Xenomai, mentions that “the fewer the cores, the better the results with the i.MX6 series”. [52] This statement means the number of cores could also have an effect on the real-time latency of i.MX7D.

This could have something to do with the real-time scheduler of the cobalt core needing to check the two cores. The Xenomai wiki also mentions the mercury configuration should be chosen over the cobalt configuration when the system uses more than 4 cores due to SMP scalability. It also states the problem is not with the physical cores, however with the number of cores actually running real-time threads. [53]

In order to test whether or not the number of cores affects the latency, one core could be turned off in the kernel. This would also require the Xenomai libraries to be reconfigured without the --enable-smp flag.

An additional co-processor in the form of an Cortex-M4 is also implemented into the i.MX7D. This core can utilize the same memory and peripherals as both Cortex-A7 cores. This hardware design might have some latency impacts, however more research on this topic is needed.
5.7 Open-source licenses

A minimal image is built using only permissive licenses and GPLv2. The necessary packages needed can as mentioned be included using a special layer. In the information for this layer, the OpenEmbedded developers recommend to rethink the strategy for not including GPLv3 in embedded builds. This is due to these packages not being updated with security updates and maintenance.

Security updates are a critical component of every control system as security breaches could lead to profit loss or even worse, personal injuries if the system does not work properly. However, the packages listed in the meta-gplv2 layer are not all safety risks. For example the which tool only displays the path to a shell command and is therefore not seen as a safety risk.

It was found that Bitbake checks the licenses for packages which are used for building the image, not only the packages included in the image. The make package is required for building the core-image-minimal, however it is not included in the image and Bitbake still flagged the package as having an incompatible license.

If packages with a Copyleft license are not distributed, they could be used for building the image. As the software is not distributed, the license is not violated either. When only using and not distributing the software, a Copyleft license would not have to be flagged as an INCOMPATIBLE_LICENSE. However this option would require developers to be absolutely certain of which packages they include in the image.  

5.8 The Yocto Layer

The Yocto layer created during this thesis uses a .bbappend which Bitbake directly adds to the .bb file of the same name if the layer is added to the bblayers.conf file. If the cobalt core should not be added to the i.MX kernel, the entire layer should be removed. In order to avoid removing the layer, a .bb file could be created specifically for adding a cobalt kernel which could then be chosen using the PREFERRED_PROVIDER_virtual/kernel variable in the environment’s local.conf.

It was chosen to use a .bbappend as this would not require copying of the entire existing Compulab kernel recipe with all patches and now only a single task is added which implements Xenomai. Furthermore, the layer only includes Xenomai related packages so if Xenomai is not required, the layer might as well be removed.

---

4This thesis is not meant as legal advice. Consult a professional legal advisor when distributing open-source code. Neither is this thesis meant as ethical advice. If you use open-source code, contribute back to the community.
6 Conclusion

A RTOS using Xenomai’s cobalt core or the PREEMPT_RT patch can be implemented on the CL-SOM-iMX7 using the i.MX7D with the i.MX vendor kernel. The I-Pipe patch for the cobalt core requires some changes before it can be applied to the kernel and the PREEMPT_RT patch can be applied directly. Both of these methods are done with the Yocto Project and only uses corporate friendly licenses to build the image.

Xenomai’s cobalt core has a lower latency than the PREEMPT_RT patch which makes it the best candidate for a control system. When the CPU is idle, the PREEMPT_RT patch has a latency of 412.375\(\mu\)s while the cobalt core only has a latency of 155.750\(\mu\)s. Nevertheless, the deadline of 100 ms for the control loop should be able to be achieved with both core types.

The maximum latency with the dobell script for the cobalt core is 268.250\(\mu\)s, which is high compared to other studies done with for example a Raspberry Pi 3B. A reason for this high latency was not found and it is concluded both the I-Pipe patch and the PREEMPT_RT patch are implemented correctly as no outlier latency peaks are shown during the latency tests. Outlier latency peaks from the curve would be typical for a normal operating system.

A summary of the partial goals of the thesis are shown in table 5, here it is also stated whether or not the goal was achieved.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Explanation</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTOS</td>
<td>A RTOS should be made for the CL-SOM-iMX7 made with the latest stable Xenomai and kernel versions.</td>
<td>Yes</td>
</tr>
<tr>
<td>Benchmark</td>
<td>Xenomai’s latency should be benchmarked on the i.MX7D with the testsuite.</td>
<td>Yes</td>
</tr>
<tr>
<td>Latency</td>
<td>The maximum latency should be less than 100 ms.</td>
<td>Yes</td>
</tr>
<tr>
<td>Yocto</td>
<td>The RTOS should be built using the Yocto project.</td>
<td>Yes</td>
</tr>
<tr>
<td>Licenses</td>
<td>Only cooperate friendly licenses should be used in the final image.</td>
<td>Yes</td>
</tr>
<tr>
<td>Comparison</td>
<td>The latency on the NXP i.MX7D should be compared to another SOM, for example a TI AM4379 or previous hardware used.</td>
<td>To a degree</td>
</tr>
</tbody>
</table>

Table 5 – Every partial goal was completed except for the comparison with a TI AM4379. However the latency was compared to other white papers.
REFERENCES

References


REFERENCES


34
REFERENCES


REFERENCES


Retrieved from: https://lwn.net/Articles/145973/

https://layers.openembedded.org/layerindex/branch/master/layer/meta-gplv2/

https://mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/

Master Thesis. KTH Royal Institute of Technology.
Retrieved from: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-235484

Real-Time Linux Workshop.

Xenomai and Linux for real-time applications. In proc. of the 12th Real-Time Linux
Workshop (RTLWS'12) (pp. 1-17).
Retrieved from: https://pdfs.semanticscholar.org/9eb5/1db823034e80b8664d828
1996d2a9659e.pdf


#user-content-single-or-dual-kernel-configuration
A Appendix - patch for gpc.c

```c
diff --Naur old--gpc.c patched--gpc.c
--- old--gpc.c 2019-04-17 08:07:14.655156000 +0200
+++ patched--gpc.c 2019-04-17 08:47:49.023213853 +0200
@@ -15,6 +15,7 @@
 #include <linux/io.h>
 #include <linux/irq.h>
 #include <linux/irqchip.h>
+#include <linux/ipipe.h>
 #include <linux/of.h>
 #include <linux/of_address.h>
 #include <linux/of_irq.h>
@@ -92,7 +93,8 @@
 static u32 gpc_saved_imrs[IMR_NUM];
 static u32 gpc_mf_irqs[IMR_NUM];
 static u32 gpc_mf_request_on[IMR_NUM];
-#define SPI_DEFINE_SPINLOCK(gpc_lock);
+// #define SPI_DEFINE_RAW_SPINLOCK(gpc_lock);
 static struct notifier_block nb_pcie;
 static struct pu_domain imx6q_pu_domain;
 static bool pu_on;  /* keep always on i.mx6qp */
@@ -229,6 +231,7 @@
 void imx_gpc_pre_suspend(bool arm_power_off)
 {
     void __iomem *reg_imr1 = gpc_base + GPC_IMR1;
+    unsigned long flags;
     int i;
     if (cpu_is_imx6q() && imx_get_soc_revision()\
         == IMX_CHIP_REVISION_2_0)
@@ -243,15 +246,20 @@
         if (arm_power_off)
             imx_gpc_set_arm_power_in_lpm(arm_power_off);
+    flags = hard_cond_local_irq_save();
+    for (i = 0; i < IMR_NUM; i++) {
        gpc_saved_imrs[i] = readl_relaxed(reg_imr1 + i * 4);
        writel_relaxed(^gpc_wake_irqs[i], reg_imr1 + i * 4);
```
+ hard_cond_local_irq_restore(flags);
}

void imx_gpc_post_resume(void)
{
    void __iomem *reg_imr1 = gpc_base + GPC_IMR1;
    unsigned long flags;
    int i;
    if (cpu_is_imx6q() && imx_getsoc_revision() == IMXCHIP_REVISION_2_0)
        IMR_259_13 + 267_18 @@
    /* Keep ARM core powered on for other low-power modes */
    imx_gpc_set_arm_power_in_lpm(false);
    /* Keep M/F mix powered on for other low-power modes */
    if (cpu_is_imx6sx() || cpu_is_imx6ul() || cpu_is_imx6ull())
        IMR_290_22 + 303_32 @@
    flags = hard_cond_local_irq_save();
    for (i = 0; i < IMR_NUM; i++)
        IMR_4
        writel_relaxed(gpc_saved_imrs[i], reg_imr1 + i * 4);
    hard_cond_local_irq_restore(flags);
}

static int imx_gpc_irq_set_wake(struct irq_data *d, unsigned int on)
    IMR_290_22 + 303_32 @@
void imx_gpc_mask_all(void)
{
    void __iomem *reg_imr1 = gpc_base + GPC_IMR1;
    unsigned long flags;
    int i;
    flags = hard_cond_local_irq_save();
    for (i = 0; i < IMR_NUM; i++)
        IMR_4
        gpc_saved_imrs[i] = readl_relaxed(reg_imr1 + i * 4);
writel_relaxed(~0, reg_imr1 + i * 4);
}
+
hard_cond_local_irq_restore(flags);
+
void imx_gpc_restore_all(void)
{
    void *reg_imr1 = gpc_base + GPC_IMR1;
    unsigned long flags;
    int i;
    flags = hard_cond_local_irq_save();
    +
    for (i = 0; i < IMR_NUM; i++)
        writel_relaxed(gpc_saved_imrs[i], reg_imr1 + i * 4);
    +
    hard_cond_local_irq_restore(flags);
}

void imx_gpc_hwirq_unmask(unsigned int hwirq)
{
    static void imx_gpc_irq_unmask(struct irq_data *d)
    {
        unsigned long flags;
        raw_spin_lock_irqsave(&gpc_lock, flags);
        imx_gpc_hwirq_unmask(d->hwirq);
        __pipe_spin_unlock_irqbegin(&gpc_lock);
        irq_chip_unmask_parent(d);
        __pipe_spin_unlock_irqcomplete(flags);
    }

    static void imx_gpc_irq_mask(struct irq_data *d)
    {
        unsigned long flags;
        raw_spin_lock_irqsave(&gpc_lock, flags);
        /* Parent IC will handle virtual locking */
        imx_gpc_hwirq_mask(d->hwirq);
        __pipe_spin_unlock_irqbegin(&gpc_lock);
        irq_chip_mask_parent(d);
```c
+__ipipe_spinnlock_irqcomplete(flags);
+
+} #ifdef CONFIG_IPIPE
+
+static void imx_gpc_hold_irq(struct irq_data *d) {
+  raw_spin_lock(&gpc_lock);
+  imx_gpc_hwiirq_mask(d->hwiirq);
+  raw_spin_unlock(&gpc_lock);
+  irq_chip_hold_parent(d);
+
+}
+
+static void imx_gpc_release_irq(struct irq_data *d) {
+  unsigned long flags;
+  raw_spin_lock_irqsave(&gpc_lock, flags);
+  imx_gpc_hwiirq_unmask(d->hwiirq);
+  raw_spin_unlock_irqrestore(&gpc_lock, flags);
+  irq_chip_release_parent(d);
+
+} #endif /* CONFIG_IPIPE */
+
+static struct irq_chip imx_gpc_chip = {
+  .name = "GPC",
+  .irq_eoi = irq_chip_eoi_parent,
+  .irq_set_affinity = irq_chip_set_affinity_parent,
+  #ifdef CONFIG_SMP
+  .irq_set_affinity = irq_chip_set_affinity_parent,
+  #endif
+  #ifdef CONFIG_IPIPE
+  .irq_hold = imx_gpc_hold_irq,
+  .irq_release = imx_gpc_release_irq,
+  #endif
+};
+
+static int imx_gpc_domain_translate(struct irq_domain *d,
```
B  Appendix - patch for gpcv2.c

diff −Naur old−gpcv2.c patched−gpcv2.c
--- old−gpcv2.c 2019−04−17 08:07:14.655156000 +0200
+++ patched−gpcv2.c 2019−04−18 10:56:01.067175892 +0200
@@ −14,6 +14,7 @@
#include <linux/io.h>
#include <linux/irq.h>
#include <linux/irqchip/arm−gic.h>
+include <linux/ipipe.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/of_irq.h>
@@ −114,7 +115,8 @@
static u32 gpcv2_saved_imrs_m4[IMR_NUM];
static u32 gpcv2_mf_irqs[IMR_NUM];
static u32 gpcv2_mf_request_on[IMR_NUM];
−static DEFINE_SPINLOCK(gpcv2_lock);
+static PIPE_DEFINE_RAW_SPINLOCK(gpcv2_lock);
+static DEFINE_SPINLOCK(gpcv2_lock);
+static struct notifier_block nb_mipi, nb_pcie, nb_usb_hsic;
void imx_gpcv2_add_m4_wake_up_irq(u32 hwirq, bool enable)
@@ −154,21 +156,31 @@
void imx_gpcv2_mask_all(void)
{
    void __iomem *reg_imr1 = gpc_base + GPC_IMR1_CORE0;
    +unsigned long flags;
    int i;

    flags = hard_cond_local_irq_save();
    +
    for (i = 0; i < IMR_NUM; i++) {
        gpcv2_saved_imrs[i] = readl_relaxed(reg_imr1 + i * 4);
        writel_relaxed(~0, reg_imr1 + i * 4);
    }
    +
    hard_cond_local_irq_restore(flags);
}

void imx_gpcv2_restore_all(void)
{
void __iomem *reg_imr1 = gpc_base + GPC_IMR1_CORE0;
+ unsigned long flags;
    int i;
+ flags = hard_cond_local_irq_save();
+ for (i = 0; i < IMR_NUM; i++)
    write1_relaxed(gpcv2_saved_imrs[i], reg_imr1 + i * 4);
+ hard_cond_local_irq_restore(flags);
} }

void imx_gpcv2_hwirq_unmask(unsigned int hwirq)
@@ -195,14 +207,25 @@
static void imx_gpcv2_irq_unmask(struct irq_data *d)
{
    + unsigned long flags;
    +
    + raw_spin_lock_irqsave(&gpcv2_lock, flags);
    + imx_gpcv2_hwirq_unmask(d->hwirq);
    + __ipipe_spin_unlock_irqbegin(&gpcv2_lock);
    + irq_chip_unmask_parent(d);
    + __ipipe_spin_unlock_irqcomplete(flags);
}

static void imx_gpcv2_irq_mask(struct irq_data *d)
{
    + unsigned long flags;
    +
    + raw_spin_lock_irqsave(&gpcv2_lock, flags);
    + /* Parent IC will handle virtual locking */
    + imx_gpcv2_hwirq_mask(d->hwirq);
    + __ipipe_spin_unlock_irqbegin(&gpcv2_lock);
    + irq_chip_mask_parent(d);
    + __ipipe_spin_unlock_irqcomplete(flags);
}

void imx_gpcv2_set_slot_ack(u32 index, enum imx_gpc_slot m_core,
@@ -519,6 +542,7 @@
void imx_gpcv2_pre_suspend(bool arm_power_off)
{
    void __iomem *reg_imr1 = gpc_base + GPC_IMR1_CORE0;

    vi
84 + unsigned long flags;
85     int i;
86
87     if (arm_power_off) {
88         imx_gpcv2_set_lpm_mode(STOP_POWER_ON);
89     }
90
91     flags = hard_cond_local_irq_save();
92     +
93     for (i = 0; i < IMR_NUM; i++) {
94         gpcv2_saved_imrs[i] = readl_relaxed(reg_imr1 + i * 4);
95         writel_relaxed(~gpcv2_wake_irqs[i], reg_imr1 + i * 4);
96     }
97     +
98     hard_cond_local_irq_restore(flags);
99 }
100 }
101
102 void imx_gpcv2_enable_wakeup_for_m4(void)
103 {
104     void __iomem *reg_imr1 = gpc_base + GPC_IMR1_CORE0;
105     + unsigned long flags;
106     int i, val;
107
108     /* only external IRQs to wake up LPM and core 0/1 */
109     writel_relaxed(val, gpc_base + GPC_MLPCR);
110 }
111
112 + flags = hard_cond_local_irq_save();
113 +
114     for (i = 0; i < IMR_NUM; i++)
115     writel_relaxed(gpcv2_saved_imrs[i], reg_imr1 + i * 4);
116 +
117     hard_cond_local_irq_restore(flags);
118 +
119     imx_gpcv2_set_lpm_mode(WAIT_CLOCKED);
120     imx_gpcv2_set_cpu_power_gate_by_lpm(0, false);
121     imx_gpcv2_set_plat_power_gate_by_lpm(false);
122 +
123     imx_gpcv2_enable_rbc(false);
+static void imx_gpcv2_hold_irq(struct irq_data *d)
{  
  raw_spin_lock(&gpcv2_lock);
  imx_gpcv2_hwirq_mask(d->hwirq);
  raw_spin_unlock(&gpcv2_lock);
  irq_chip_hold_parent(d);
}

+static void imx_gpcv2_release_irq(struct irq_data *d)
{  
  unsigned long flags;
  
  raw_spin_lock_irqsave(&gpcv2_lock, flags);
  imx_gpcv2_hwirq_unmask(d->hwirq);
  raw_spin_unlock_irqrestore(&gpcv2_lock, flags);
  irq_chip_release_parent(d);
}

+#endif /* CONFIG_IPIPE */

static struct irq_chip imx_gpcv2_chip = {
  .name = "GPCV2",
  .irq_eoi = irq_chip_eoi_parent,
  @ @ -663,6 +717,10 @@
  #ifdef CONFIG_SMP
  .irq_set_affinity = irq_chip_set_affinity_parent,
  #endif
  
  +#ifdef CONFIG_IPIPE
  +.irq_hold = imx_gpcv2_hold_irq,
  +.irq_release = imx_gpcv2_release_irq,
  +#endif

  
};

static int imx_gpcv2_domain_xlate(struct irq_domain *domain,
C Appendix - patch for power configuration

```c

diff --git a/arch/arm/mach-imx/anatop.c b/arch/arm/mach-imx/anatop.c
index e6e5572..6a612ca 100644
--- a/arch/arm/mach-imx/anatop.c
+++ b/arch/arm/mach-imx/anatop.c
@@ -145,6 +145,7 @@ void imx_anatop_pre_suspend(void)
       return;
 }

+if defined(CONFIG_SOC_IMX6Q) || defined(CONFIG_SOC_IMX6SL)
  if (cpu_is_imx6q() && imx_get_soc_revision() == IMX_CHIP_REVISION_2_0)
    imx_anatop_disable_pt(true);

@@ -60,6 +61,7 @@ void imx_anatop_pre_suspend(void)
@@ -160,6 +161,7 @@ void imx_anatop_pre_suspend(void)
@@ -175,6 +177,7 @@ void imx_anatop_post_resume(void)
    void imx_anatop_post_resume(void)
      return;
   }

+if defined(CONFIG_SOC_IMX6Q) || defined(CONFIG_SOC_IMX6SL)
  if (cpu_is_imx6q() && imx_get_soc_revision() == IMX_CHIP_REVISION_2_0)
    imx_anatop_disable_pt(false);

@@ -190,6 +193,7 @@ void imx_anatop_post_resume(void)
@@ -147,7 +147,7 @@ static int busfreq_notify(enum busfreq_event event)

 static void imx_anatop_usb_chrg_detect_disable(void)
 diff --git a/arch/arm/mach-imx/busfreq-imx.c b/arch/arm/mach-imx/busfreq-imx.c
index 353795c..ce17cad 100644
--- a/arch/arm/mach-imx/busfreq-imx.c
+++ b/arch/arm/mach-imx/busfreq-imx.c
@@ -147,7 +147,7 @@ static int busfreq_notify(enum busfreq_event event)

 return notifier_to_errno(ret);
 }
```

ix
int register_busfreq_notifier(struct notifier_block *nb)
{
    return raw_notifier_chain_register(&busfreq_notifier_chain, nb);
}

int unregister_busfreq_notifier(struct notifier_block *nb)
{
    return raw_notifier_chain_unregister(&busfreq_notifier_chain, nb);
}

EXPORT_SYMBOL(unregister_busfreq_notifier);

static struct clk *origin_step_parent;

static void imx6ull_lower_cpu_rate(bool enter)
{
    int ret;

    if (enter) {
        org_arm_rate = clk_get_rate(arm_clk);
        origin_arm_volts = regulator_get_voltage(arm_reg);
        origin_soc_volts = regulator_get_voltage(soc_reg);
    }

    clk_set_parent(pll1_bypass_clk, pll1_bypass_src_clk);
    clk_set_parent(pll1_sw_clk, pll1_sys_clk);
}

void request_bus_freq(enum bus_freq_mode mode)
{
    mutex_lock(&bus_freq_mutex);
    cur_bus_freq_mode = mode;
    EXPORT_SYMBOL(get_bus_freq_mode);
}

EXPORT_SYMBOL(get_bus_freq_mode);

int get_bus_freq_mode(void)
{
    return cur_bus_freq_mode;
}

EXPORT_SYMBOL(get_bus_freq_mode);

int set_high_bus_freq(int high_bus_freq)
{
    return 0;
}

if defined(CONFIG_CPU_FREQ) && !defined(CONFIG_ARM64)

*/

+// fixme: only works with imx7d

+-/

+ if defined(CONFIG_CPU_FREQ) && !defined(CONFIG_ARM64)

-
static struct map_desc ddr_iram_io_desc__initdata = {
    .virtual = 1, .PFN = 0, .length = SZ_1M,
}

#include "arch/arm/mach-imx/mach-imx7d.c"

static void __init imx7d_init_machine(void)
{
    pr_warn("failed to initialize soc device\n");
}

static struct platform_default_populate
{
    .pm = &pm
}

static struct platform_driver imx6_pcie_driver = {
    .name = "imx6_pcie",
    .of_match_table = imx6_pcie_of_match,
    .pm = &pci_imx_pcie_ops,
}

static struct platform_driver imx6_pcie_driver = {
    .name = "imx6_pcie",
    .of_match_table = imx6_pcie_of_match,
    .pm = &pci_imx_pcie_ops,
}

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init imx7d_pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif

+ static void __init pm_init()
+
+ #ifdef CONFIG_PM
+ imx7d_pm_init();
+ #endif
Appendix - linux-compulab_4.9.11.bbappend

DESCRIPTION = "Provides userspace xenomai support and libraries needed to for
real-time applications using the xenomai RTOS implementation"

LICENSE = "GPLv2"

SECTION = "xenomai"

HOMEPAGE = "http://www.xenomai.org/"

PR = "r0"

SRC_URI += 
"https://xenomai.org/downloads/xenomai/stable/xenomai-3.0.8.tar.bz2;name=xeno"
SRC_URI += "c373261ddb8280d9d7078cdd9cd9646dfb7d70d8cd3aa9603d9148f03990d711"

FILESEXTRAPATHS_prepend = "${THISDIR}/${PN}:":

IPIPatch = "ipipe-4.9.11-imx-arm.patch"

SRC_URI += "file://${IPIPatch};apply=0"

SRC_URI += "file://fix_no_power_saving_modes_imx7d_4.9.11.patch"

SRC_URI += "file://config_imx7d_xenomai;apply=0"

XENOMAI_SRC = "${WORKDIR}/xenomai-3.0.8"

#INHIBIT_DEFAULT_DEPS = "1"

do_add_xenomai () {
    # Prepare kernel
    ${XENOMAI_SRC}/scripts/prepare-kernel.sh --arch=${ARCH} \ 
    --linux=${S} --ipipe="${WORKDIR}/${IPIPatch}" --default
    cp ${WORKDIR}/config_imx7d_xenomai ${B}/.config
}

addtask add_xenomai after do_preconfigure before do_configure
E Appendix - xenomai_3.0.8.bb

1 DESCRIPTION = "Provides userspace xenomai support and libraries needed to for real-time applications using the xenomai RTOS implementation"
2 LICENSE = "GPLv2"
3 LIC_FILES_CHKSUM = "file://include/COPYING;md5=79ed705ccb9481bf9e7026b99f4e2b0e"
4 SECTION = "xenomai"
5 HOMEPAGE = "http://www.xenomai.org/
6 #PR = "r0"
7
8 SRC_URI = "https://xenomai.org/downloads/xenomai/stable/xenomai-{PV}.tar.bz2"
9 SRC_URI[sha256sum] = "c373261ddbb2880d9d7078cdd9c9646dfb7d70d8cd3aa9693d9148f03990d711"
10 S = "$ {WORKDIR}/xenomai-{PV}"
11 inherit autotools pkgconfig
12 includedir = "/usr/include/xenomai"
13 EXTRA_OCONF += "--enable-smp \\n14    --with-core=cobalt \\n15    --prefix="n"
16
17 PACKAGES += "$ {PN}-demos"
18 FILES_${PN}-demos = "/usr/demos"
19 FILES_${PN}-dev += "/dev"
20 FILES_${PN} += " \\n21    /usr/lib/modchk.wrappers \ 
22    /usr/lib/cobalt.wrappers \ 
23    /usr/lib/dynlist.ld \ 
24    "

xiii
F Appendix - local.conf

1  MACHINE ?= 'cl-som-imx7'
2  DISTRO ?= 'fsl-imx-x11'
3  PACKAGE_CLASSES ?= "package_rpm"
4  EXTRA_IMAGE_FEATURES ?= "debug-tweaks"
5  USER_CLASSES ?= "buildstats image-mklib image-prelink"
6  PATCHRESOLVE = "noop"
7  BB_DISKMON_DIRS ?? = ""
8  STOP_TASKS, ${TMPDIR}, 1G, 100K
9  STOP_TASKS, ${DL_DIR}, 1G, 100K
10 STOP_TASKS, ${STATE_DIR}, 1G, 100K
11 STOP_TASKS, /tmp, 100M, 100K
12 ABORT, ${TMPDIR}, 100M, 1K
13 ABORT, ${DL_DIR}, 100M, 1K
14 ABORT, ${STATE_DIR}, 100M, 1K
15 ABORT, /tmp, 10M, 1K
16 PACKAGECONFIG_append pn-qemu-native = " sdl"
17 PACKAGECONFIG_append pn-nativesdk-qemu = " sdl"
18 CONF_VERSION = "1"
19
20 DL_DIR ?= "${BSPDIR}/downloads/
21 ACCEPT_FSL_EULA = "1"
22 CORE_IMAGE_EXTRA_INSTALL += " can-utils ntp dhcp-client memtester 
23 cl-deploy cl-boot cl-cleanup xenomai xenomai-dev xenomai-demos"
24 LICENSE_FLAGS_WHITELIST = "commercial"
25 PREFERRED_VERSION_linux-compulab = "4.9.11%"
26
27 INCOMPATIBLE_LICENSE = "GPL-3.0 LGPL-3.0 AGPL-3.0"
28 BBMASK += "recipes-security/optee-imx/optee-os-imx_git.bb"
29 BBMASK += "recipes-browser/chromium/chromium_%_bbappend"