POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Porting of a dedicated Linux distribution for IP cameras to the Ingenic T32 processor: development process on the GOAT System-on-Chip

Supervisors

Prof. Luciano LAVAGNO

Matteo MASCIOCCHI

Candidate

Agostino SAVIANO

March 2026

Abstract

Utopia s.r.l. is a company that produces monitoring systems used on machinery and trucks for the purpose of observing their surroundings from inside the cockpit. A viewable feed is composed from many custom built IP cameras, running the open source OpenIPC distribution that provides a ready-made solution with hardware support for the image sensors in use and a streaming software to send data over ethernet to the main monitoring apparatus. The board used for the cameras is equipped with the Ingenic T31 processor, quickly approaching End-of-Life status, so the company was seeking to explore if OpenIPC could be adapted to the new T32 processor, by means of porting it to their development board (codenamed `GOAT`) so that replacement cameras could be manifactured. The thesis explains the approaches, instruments and software used to create a working prototype sporting the aforementioned T32 processor that can be, with the necessary future verification and integration into Utopia's process, a drop-in replacement for their cameras used in production.

Acknowledgements

A Zio Natale, per aver creduto in me.
Alle persone che mi sono più care, per aver fatto lo stesso.

Table of Contents

List of Figures

List of Listings

Chapter 1

Introduction

This thesis presents the prototyping efforts conducted in Utopia s.r.l. over the course of approximately three months for the porting of OpenIPC, a Linux based distribution for IP cameras. The company in question produces monitoring equipment placed inside trucks and other machinery; the apparatus is composed of a main body that is connected to a number of IP cameras that send a stream of video data through Ethernet that is then composed into a single feed and shown to the operator. Cameras currently in production are equipped with the T31 CPU manufactured by Ingenic Semiconductor, though this model is approaching End-of-Life status. To prevent potential supply line issues and to future-proof the cameras, Utopia opted to explore the path of upgrading to the next generation counterpart, the T32, and subsequently porting OpenIPC to it, because as the project currently stands, that model is not officially supported.

Ingenic provides a development board codenamed `GOAT` that has been used as the backbone of the project and that was to be converted into a prototype to base the next series of production cameras on. Starting with a thorough study of the build system used by OpenIPC, the same was done with the Ingenic SDK, that contained a functioning Linux kernel and drivers for image sensors shipped with the board, along with many code samples. The bulk of the effort was spent integrating the Ingenic-provided kernel with the OpenIPC build system, complemented with a customization package for Utopia that follows the specification adopted by OpenIPC and the necessary software to extract a video feed. The resulting camera prototype is able to quickly boot into program execution and provide a feed that has been proven to work with the systems already in use by Utopia.

The methods and reasoning behind the process are discussed in detail according to the following outline:

- **Background** - Main software projects utilized throughout the thesis and functional details on some of their specific features; overview on how a generic embedded Linux device boots up.
- **Development process** - Incremental steps on how the project was brought to completion: rationale on the integration direction to follow; board bring-up procedure; resolution of after-boot issues and enhancement of resulting system image.
- **Future works** - Later development proposals and requirements for production-ready deployment.
- **Conclusions and final thoughts** - Closing thoughts and comments on the project development experience.

Chapter 2

Background

2.1 OpenIPC

OpenIPC is an open source operating system for IP cameras based on Linux, designed to replace opaque and closed firmwares on an ample range of devices that all have in common the CPU architecture, MIPS and ARM, though in this thesis focus is placed on the former as both T31 and T32 CPUs incorporate the MIPS XBurst-1 microarchitecture. It is distributed as flashable pre-compiled binary files, with included drivers for image sensors and other hardware, for example Wi-Fi dongles and motors, but also per-vendor features by means of including libraries from their respective SDKs. To complement the system, it comes pre-installed with a streamer software called Majestic, a lightweight ssh server for remote access to the machine and many utilities, most of which provided by Busybox.

OpenIPC is the foundation of the whole thesis project as it provides a ready-made and quite featured system, whose only deficiency is missing support for the T32 CPU and lack of a working streaming software, all points of discussion that will be clarified in later chapters.

2.2 Buildroot: OpenIPC's build system

OpenIPC as a distribution is built using Buildroot, a tool that simplifies and automates the process of building a complete Linux distribution for an embedded system, using cross-compilation. It is able to generate, independently if needed, a cross-compilation toolchain, a root filesystem, a Linux kernel image and a bootloader for a chosen target [1]. For this project, only the second and third are needed. This section will be focused on the features used by OpenIPC to build the distribution, specifically those that have been interacted with to bring the project to completion, and not on general usage.

Buildroot is composed of many Makefiles that obtain the source code for a specific software, configure and compile it with the correct options. Each software package has its own Makefile, the structure of which will be described further down this section. Makefiles are separated by directory according to the following schema [2], though not all will be relevant for the project's purposes:

- `toolchain/`: Makefiles and associated files related to software responsible for building cross-compilation toolchains. Since Ingenic's SDK already includes a x86_64 to MIPS toolchain, this feature will not be adopted.
- `arch/`: definitions for all CPU architectures supported.
- `package/`: Makefiles and related files for user-space utilities and libraries that can be compiled by the build system and subsequently added to the target filesystem. Notable because this is where the package for customizations made for Utopia will be put. One subdirectory is present for every package.
- `linux/`: Makefiles and associated files for the Linux kernel.
- `boot/`: Makefiles and associated files for all bootloaders supported by Buildroot. Unused, U-Boot (the bootloader of choice) will be built separately.
- `system/`: system integration files (skeleton of root filesystem, selection of init system).
- `fs/`: Makefiles and associated files for the generation of the target root filesystem image (including the choice of which format to use e.g. UBIFS, SquashFS and similar).

The aforementioned hierarchy lives inside OpenIPC's `output` folder and is generated when running its main Makefile that downloads it from a known source before passing control to Buildroot itself.

For each directory there are at least 2 files:

- `example.mk`: the Makefile that downloads, configures, compiles and installs the package `example`.
- `Config.in`: partial description file for the configuration tool, it describes the options, dependencies and info about the package.

This is exactly how Buildroot packages are integrated into the build system. Taking *divinus*, a streamer software from the same developers of OpenIPC, as an example, one can see in Listing 2.1 and correlated figure the correspondence between `Config.in` and its presentation in the configuration utility `menuconfig`. In Listing 2.2 instead the Makefile is listed with a header section, toolchain options,

the build commands and those for installing the resulting objects into the target's filesystem.

```
config BR2_PACKAGE_DIVINUS
    bool "Divinus streamer"
    help
     Multi-platform open source
    streamer
     https://github.com/openipc/
    divinus

```

Listing 2.1: divinus Config.in

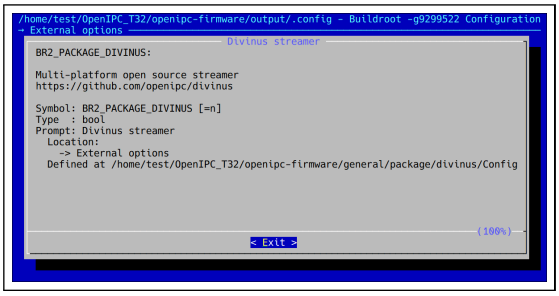

Figure 2.1: Menuconfig entry for divinus.

```
DIVINUS_SITE = $(call github,openipc,divinus,$(DIVINUS_VERSION))
DIVINUS_VERSION = HEAD
DIVINUS_LICENSE = MIT
DIVINUS_LICENSE_FILES = LICENSE

ifeq ($(BR2_TOOLCHAIN_USES_GLIBC),y)
DIVINUS_OPTIONS = "-rdynamic -s -Os -lm"
else
DIVINUS_OPTIONS = "-rdynamic -s -Os"
endif

define DIVINUS_BUILD_CMDS
$(MAKE) CC=$(TARGET_CC) OPT=$(DIVINUS_OPTIONS) -C $(@D)/src
endef

define DIVINUS_INSTALL_TARGET_CMDS
$(INSTALL) -m 755 -d $(TARGET_DIR)/etc
$(INSTALL) -m 644 -t $(TARGET_DIR)/etc $(@D)/divinus.yaml

$(INSTALL) -m 755 -d $(TARGET_DIR)/usr/bin
$(INSTALL) -m 755 -t $(TARGET_DIR)/usr/bin $(@D)/divinus
endef

$(eval $(generic-package))
```

Listing 2.2: Divinus Makefile (divinus.mk)

OpenIPC makes extensive use of Buildroot's `br2-external trees`, which is its mechanism to manage project-specific customizations without polluting its own source tree. Observing the root directory [Fig. 2.2] of the repository, one can see many external trees divided by chip vendors;

```
.
├── br-ext-chip-allwinner/
├── br-ext-chip-ambarella/
├── br-ext-chip-anyka/
├── br-ext-chip-fullhan/
├── br-ext-chip-goke/
├── br-ext-chip-grainmedia/
├── br-ext-chip-hisilicon/
├── br-ext-chip-ingenic/
├── br-ext-chip-novatek/
├── br-ext-chip-rockchip/
├── br-ext-chip-sigmastar/
├── br-ext-chip-ti/
├── br-ext-chip-xiongmai/
├── general/
├── LICENSE
├── Makefile
├── output/
└── README.md
```

Figure 2.2: Directory listing of OpenIPC's root folder

instead inside the `general` subfolder the necessary files to configure the logic of the external trees are present, mainly:

- `external.desc`: contains an unique name for the tree and a brief description.
- `Config.in`: defines models and options extracted from each vendor's tree to be used later in the process (SoC model and family, sensor model, flash size etc.).
- `external.mk`: Makefile recipe that extracts the aforementioned models and options from *defconfigs* (see below) and sets up the external tree logic together with recipes from the packages into the main Makefile logic.

In addition, an `overlay` folder with files that will be placed on top of the generated root file system for the target and a `scripts` folder for build scripts used later in the process. The `package` folder has already been discussed previously.

With regard to the external trees instead, `br-ext-chip-ingenic` is taken as an example. Subdivided by board model, it contains two types of configuration

files, one for the Buildroot (default configuration or *defconfig* as they are called) and the other for the Linux kernel. The first contains options regarding how the system is built: CPU architecture (microarchitecture, hardware floating point support); cross compilation toolchain (version, libc selection, support for various compiler options); kernel (location of source tarball and configuration file, supported compression algorithms for in-memory unpacking of the image); filesystem image (format chosen e.g. SquashFS and relative compression, also part of userspace tooling with Busybox); SoC informations like model and flash size; packages to be compiled and included in the image. The second instead (*.config*) contains configuration options for the Linux kernel. It is so customizable as to contain many thousands of key-value pair lines of text that follow the Kconfig specification [3], that instruct which kernel features to build and whether they are to be compiled directly inside the kernel or as separate, loadable modules, while also managing dependencies between said features. The *.config* file is usually interacted with a specifically made utility called `menuconfig` [Fig. 2.3], or `xconfig` for graphical environments, that presents an interactive menu in which to explore and select features to enable, along with a brief help text to understand their scope and functionality. Kconfig files themselves manage how the menu is presented, and there are many scattered throughout the whole kernel tree.

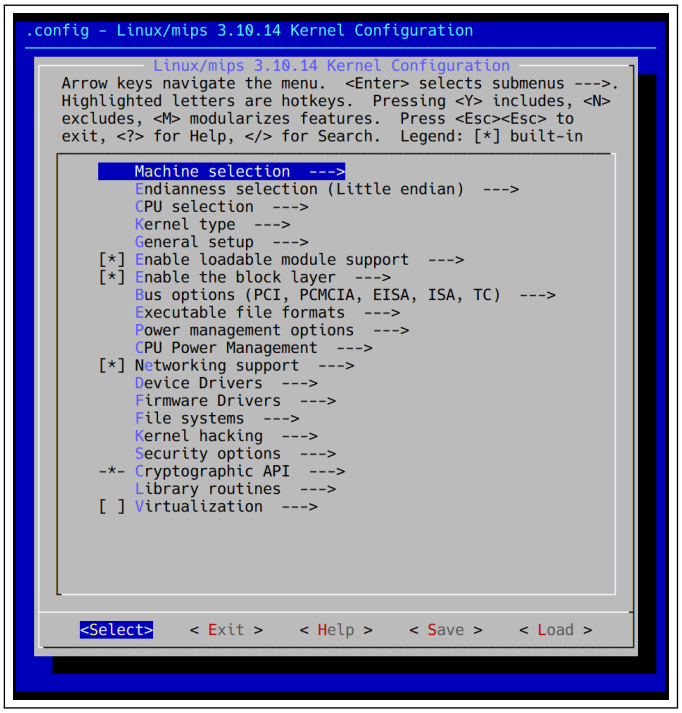

Figure 2.3: `menuconfig` showing all configurable option categories for Linux. A similar menu exists for Buildroot's defconfigs.

2.3 Linux 3.10.14: noteworthy features encountered during development

Camera systems such as those discussed in this thesis usually run older versions of the Linux kernel. For the case in point, version 3.10.14 was released back in October 2013. As for the reasons of using such an old kernel, there are many. First and foremost, OpenIPC on the T31 used this version, so it was decided that keeping the same one would result in less compatibility issues. In addition, in the embedded landscape, value is given to stability instead of being bleeding edge. The ecosystem is quite mature and proven approaches exist that have already been well tested on the field: both developers and system designers know their flaws and warts and how to tackle them. New features also tend to bloat the size of the kernel image and are usually unneeded for tasks as simple as reading data from an image sensor and streaming it to an Ethernet line.

Some of the features though are essential for the correct operation of the device and have been interacted with in a more prevalent fashion, and for this reason they will be discussed in further detail in the following subsections.

2.3.1 Memory Technology Device (MTD) subsystem

There exist a plethora of flash types and technologies: NOR, NAND and others related to those. The MTD subsystem exists to concile all of them and to present a single interface to system developers [4] that handles all the low level details. For example, though NOR flash memories can be directly byte addressable and even mapped into the CPU address space for execution, NAND flashes are read, written and erased in blocks that can be many KiB in size; moreover they are susceptible to bit flips when writing and erase blocks become unreliable over the course of thousands of erase cycles [5]; some even integrate Error Control Codes (ECC), all requiring conscious management on the software side. Chips usually have integrated microcontrollers to handle this complexity; the MTD subsystem is only concerned with unhandled, raw flash memories.

MTD consists of three layers: a set of essential functions, a set of drivers for various types of chips, and user-level drivers that present the flash memory either as a character device or a block device. Chip drivers live at the lowest level and directly interface with flash chips. NOR flash, being simpler, requires a small numbers of drivers, enough to cover the *Common Flash Memory Interface* standard and variations. For NAND flash, drivers for the controller are needed and they are usually supplied as part of the board support package. Some of them are included in the current mainline kernel [6].

Flash memories are usually partitioned in a number of areas; in the project's case to provide space for a bootloader, a kernel image, a root filesystem and additional

data to be memorized. There are several ways to specify partitions in MTD, the one used is by passing their definition through the kernel command line. Taking an example from literature for a chip of 128 MiB to be divided into five partitions:

```
mtdparts=:512k(SPL)ro,780k(U-Boot)ro,128k(U-BootEnv),4m(Kernel),-(
Filesystem)
```

The line starts with the *mtd-id* that identifies the chip. After that, there is a comma-separated list of partitions together with their sizes in bytes and names in parenthesis. The *ro* suffix is used to mark a partition read-only, while the dash specifies that the last partition takes all the remaining space [6].

2.3.2 OverlayFS

OverlayFS is the main implementation on Linux of a so-called *union filesystem*, that is a filesystem that permits the merger of two or more filesystems. There are different existing implementations of this concept, namely `aufs` and `UnionFS`, but the former exists out-of-tree and can either be built as a loadable kernel module or provided patched together with the whole Linux mainline git tree [7]; development of the latter instead has stopped since 2014 with the last release being version 2.5.13 for Linux 3.14.0-rc7 [8], though a `FUSE` (Filesystem in Userspace) implementation is currently maintained and can be found on recent package repositories [9] (for example on Ubuntu 24.04.3 LTS, which is the version used for the development of this project).

OverlayFS proves to be the dominant solution on this front, having been included in the mainline Linux kernel since version 3.18 and being extensively employed in high profile projects like Android and Docker.

Regarding its operation, an overlay-filesystem tries to present a filesystem which is the result over overlaying one filesystem on top of the other [10]. It combines two of them, an *upper* and a *lower* one. When a name is present in both filesystems, the object in the upper is visible while the one in the lower is either hidden or, in the case of directories, merged with the upper object. There is no limitation that prevents the filesystems to rather be directory trees, since they can live on the same filesystem and there is no requirement for the root of a filesystem to be either upper or lower [11].

Overlaying involves directories first and foremost. If a given name is present in both upper and lower filesystems and refers to a non-directory in either, then the object in the lower is hidden and the name refers only to the upper object. If both upper and lower objects are directories, a merged directory is formed [12].

In the case of removing files or directories without changing the lower filesystem, there needs to be a record of which have been removed in the upper filesystem. For

this purpose, *whiteouts* and *opaque directories* are utilized. A whiteout corresponds to a character device with device number `0/0`. If a whiteout is found in the upper level of a merged directory, any matching name in the lower level is ignored together with the whiteout itself, that gets hidden. A directory instead is made opaque with a specific extended attribute and if one is found in the upper filesystem, any directory in the lower one with the same name gets ignored [13].

2.4 Kernel Modules

A Linux kernel module is precisely defined as a code segment capable of dynamic loading and unloading within the kernel as needed [14]. This can be achieved with utilities like `insmod` and `rmmod`; these modules enhance kernel capabilities without necessitating a system reboot. One context in which they prove themselves to be very useful is the generic *device driver* module, employed to facilitate kernel interaction with hardware components present in the system that have no ready working support for them. They make life easier for developers, that do not require to repeatedly compile the whole kernel but instead can focus only on the specific driver part. Indeed, were it not for kernel modules, the prevailing approach would steer toward a style of kernel known as *monolithic*, that requires directly modifying it to add new functionalities into the kernel image [14].

Another advantage, though that can be said only for parties that do not wish for their industry knowledge to be made public, is additional kernel functionality for proprietary hardware and software. As it is customary in the embedded world, open source kernels are enhanced with closed source extensions for, usually, hardware accelerators and other features that compose a distinctive asset not available to competitors. These are distributed as pre-compiled `.ko` files to be loaded as needed on the platform, with documentation to make it work properly, but no source code[1].

[1]This conforms to the *"Linux syscall"* exception clause of the GPL 2.0 license, that permits non disclosure of software pieces that merely "use kernel services by normal system calls" as that "is merely considered normal use of the kernel" [15].

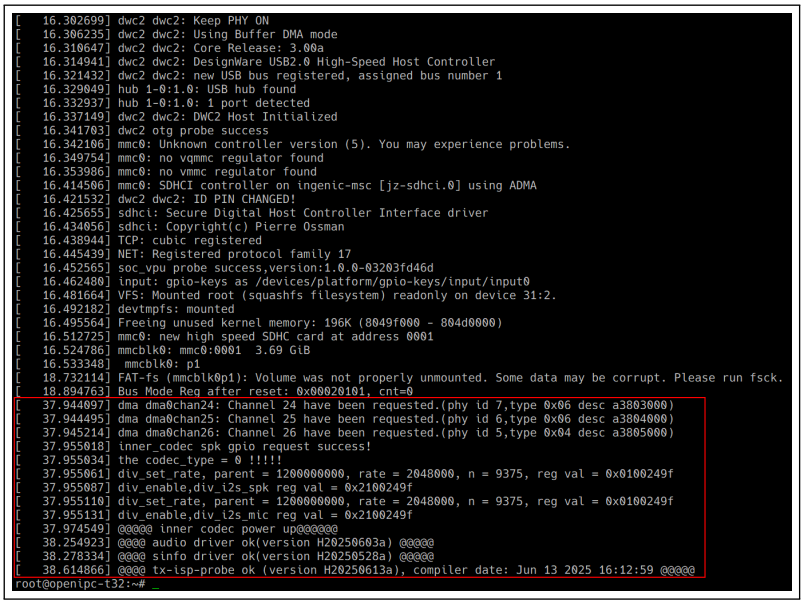

Figure 2.4: `dmesg` output after loading Ingenic's proprietary kernel modules.

2.5 Boot process for Embedded Linux

A fundamental part of running any embedded operating system is the bootloader, that is executed at the very start when powering up the board. Its responsibility is preparing the system's bare resources, though only those strictly needed for loading a kernel image and passing control to it. Chris Simmonds and Frank Vasquez provide a straightforward view of the whole boot process [16] that applies in general to all Linux embedded systems, including the one utilized in this thesis work.

The boot process can be quite simple for devices that have a NOR flash with a linear address space, but when other types of memories and peripherals are present, a multi-stage approach is usually needed.

When first powering up the board, the first code that gets executed is stored on-chip directly on the SoC, and that is the `ROM code`. It is is proprietary and burnt during manufacturing, meaning that it cannot be replaced. Early in the boot process, in this phase access to DRAM is not available since it has to be initialized by the memory controller first, and so the only one that can be used is SRAM. This kind of memory is quite expensive and so very little is provided on-board, usually in the range of a few KiB. Its job is limited to loading a small piece of code from predefined locations into SRAM, usually either from an external flash or an SD card, and in some cases even from Ethernet, USB or serial interface, though these options are normally reserved for ease and speed of development or during production. The code to load is an intermediate loader called `secondary program`

`loader (SPL)`.

Jumping to the SPL now loaded into SRAM, its capabilities are more or less the same of the ROM code (which can be thought of as the primary program loader), but in addition to that it also initializes DRAM according to the process described by the manufacturer, and also prepares any other peripheral needed for loading the `tertiary program loader (TPL)`, for example the serial interface, indeed it is normal for the SPL to show some information like version and DRAM parameters along with progress messages, but no user interaction can be had yet. At the end of this phase, the TPL is loaded into DRAM and jumped to.

The TPL now running is the bootloader proper, in this case U-Boot. It is a fundamental piece of the project, so it is discussed in its own section.

```
U-Boot 2013.07-H20250320a (Nov 18 2025 - 15:47:13)

apll_freq = 1200000000
mpll_freq = 900000000
vpll_freq = 1188000000

sdram init start
DDR clk rate 900000000
80000e5c 00000000 00000000
sdram init Finished
image entry point: 0x80100000
Board: PRJ007 (Ingenic XBurst PRJ007  SoC)
DRAM:  128 MiB
Top of RAM usable for U-Boot at: 88000000
Reserving 432k for U-Boot at: 87f90000
Reserving 32784k for malloc() at: 85f8c000
Reserving 32 Bytes for Board Info at: 85f8bfe0
Reserving 124 Bytes for Global Data at: 85f8bf64
Reserving 128k for boot params() at: 85f6bf64
Stack Pointer at: 85f6bf48
Now running in RAM - U-Boot at: 87f90000
serial_initialize done
mem_malloc_init done
MMC:   MSC: 0
SF: Detected ZB25VQ128, flash size: 16MB, manufacturer id: 5e

In:    serial
Out:   serial
Err:   serial
Net:   Jz4775-9161
Hit any key to stop autoboot:  0
SF: Detected ZB25VQ128, flash size: 16MB, manufacturer id: 5e

--->probe spend 5 ms
SF: 2621440 bytes @ 0x40000 Read: OK
--->read spend 843 ms
## Booting kernel from Legacy Image at 80600000 ...
   Image Name:   Linux-3.10.14__isvp_goat_1.0__-t
   Image Type:   MIPS Linux Kernel Image (lzma compressed)
   Data Size:    1649853 Bytes = 1.6 MiB
   Load Address: 80010000
   Entry Point:  8037c220
   Verifying Checksum ... OK
   Uncompressing lzma Kernel Image ... OK

Starting kernel ...
```

Figure 2.5: U-Boot startup process.

2.6 U-Boot

U-Boot is the bootloader of choice for this project, the final piece of the boot process whose job is to perform the last remaining initialization tasks before loading a kernel image into memory and jumping to it. It is a rich and featureful program, that can accomplish quite a lot considering its small-footprint nature; it supports many hardware platforms and is easy to port to new devices.

After starting, U-Boot shows a console that can be typed into via serial interface. It is not as versatile as bash, but considering the low level of its scope, it is more than sufficient. Commands (relevant to the project) are present for:

- Booting a kernel image from different sources: `bootm` to boot from memory, `bootp` to boot via network with TFTP protocol.

- Writing, reading, comparing and copying RAM contents: `mw`, `md`, `cmp`, `cp`.
- Interacting with subsystems for reading from and writing to various mediums (MMC cards, external NOR and NAND memories): `fatload mmc` to load generic files (mostly images) from SD to memory, `sf/nand probe/erase/write` to detect, erase and write to NOR or NAND memories.
- Interacting with the U-Boot environment: `printenv` to print current values of variables to console, `setenv` to set them, `saveenv` to save variables to persistent storage. The environment's working is explained more in detail in the next paragraph.
- Miscellaneous utility: `run` to run commands stored in environment variables, `crc32` to verify the contents of a range in RAM, `reset` to shutdown and restart the CPU, `help` to print all available commands and for help on specific ones.

Many other commands are present but the ones mentioned are the most utilized throughout the project.

The U-Boot environment is a multi-purpose mechanism used to store and pass information between functions and even to create scripts [17]. The variables are key-value pairs that get stored in a specific area of memory and may contain any value representable by a string. If U-Boot is configured to support it, the environment can survive reboots by keeping it non-volatile memory like flash by means of the `saveenv` command; a checksum is also appended to make sure that it is read back correctly. Variables can be used to store an entire semicolon-separated command sequence which can then be run with `run`. This method is widely employed to memorize the command to read into memory and load the kernel image while also passing command line arguments to it.

```
PRJ007# printenv
baudrate=115200
bootargs=console=ttyS1,115200n8 mem=64M@0x0 rmem=64M@0x4000000 init=/init_t32
rootfstype=squashfs root=/dev/mtdblock2 rw mtdparts=sfc0_nor:256k(boot),2560k(
kernel),5120k(root),-(appfs)
bootcmd=sf0 probe;sf0 read 0x80600000 0x40000 0x280000;bootm 0x80600000
bootdelay=2
ethact=Jz4775-9161
ethaddr=00:11:22:56:96:68
filesize=4db000
gatewayip=192.168.2.1
ipaddr=192.168.2.200
loads_echo=1
netmask=255.255.255.0
serverip=10.3.2.4
stderr=serial
stdin=serial
stdout=serial

Environment size: 514/16380 bytes
PRJ007#
```

Figure 2.6: Output of U-Boot `printenv` command showing some variables with their default values and their occupied size.

```
PRJ007# help
?       - alias for 'help'
base    - print or set address offset
boot    - boot default, i.e., run 'bootcmd'
boota   - boot android system
bootd   - boot default, i.e., run 'bootcmd'
bootm   - boot application image from memory
bootp   - boot image via network using BOOTP/TFTP protocol
bus_debug- open or colse the bus debug
cmp     - memory compare
coninfo - print console devices and information
cp      - memory copy
crc32   - checksum calculation
echo    - echo args to console
env     - environment handling commands
ethphy  - ethphy contrl
fatinfo - print information about filesystem
fatload - load binary file from a dos filesystem
fatls   - list files in a directory (default /)
go      - start application at address 'addr'
help    - print command description/usage
```

Figure 2.7: Truncated output of U-Boot `help` command, showing some of the available commands.

Chapter 3

Development process

Development was conducted over the course of approximately three months, with a laptop equipped with Windows 10 for research and editing, plus a virtual machine with Lubuntu 22.04 LTS which is the suggested version of the distribution by OpenIPC developers, to build kernel and flashable images for the board. Port of OpenIPC runs on Ingenic's T32 development board codenamed `GOAT` and shown in Fig. 3.1. The process was incremental, starting with study and decision of the build system, then effort went into reaching successful boot into OpenIPC, after that there was correction of significant errors and bugs that prevented the system from fully functioning, and finally refining the result with on-board software needed for a complete prototype that can be integrated into Utopia's production.

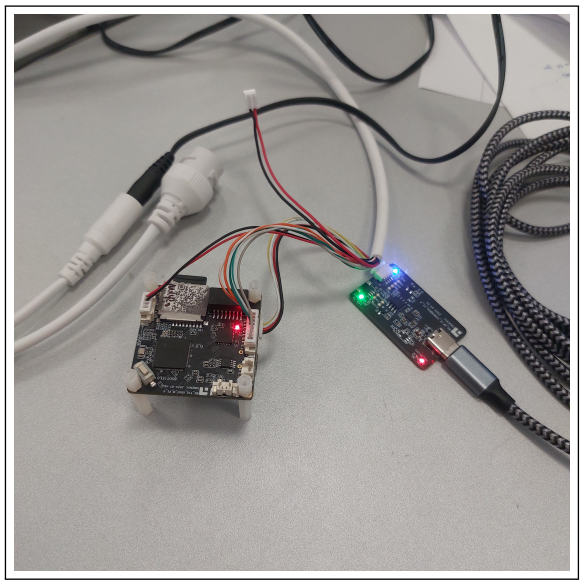

Figure 3.1: GOAT board, along with development setup.

3.1 Comparison between building processes

During the initial stage of the project, one of the first hurdles to overcome was choosing the building system to use for porting OpenIPC to the working board. Ingenic's development kit includes also a software side with the following components: source code for two different versions of the Linux kernel (3.10.14 and 4.4.94) and for U-Boot, BusyBox and drivers for supported image sensors; statically pre-compiled open source tools for debugging with two different C libraries (glibc and uclibc); pre-compiled closed source SDK libraries for interfacing with the board advanced peripherals (video encoding, neural processing unit, watchdog etc.) along with example code snippets; a custom build system comprising of essential files for the root image, a pre-compiled cross-compilation x86_64 to MIPS GCC toolchain and a complex build script `Tassadar_image_mk.sh`; many others irrelevant to the project.

Arrived at this point, a consideration has to be made on the differences between the two build systems: the one from OpenIPC and the one from Ingenic. OpenIPC's build system *Buildroot* has been thoroughly discussed in section 2.2: it is standardized, widely used in the industry, open source and extensible by following its manual. In contrast with the one from Ingenic, that is custom, and extending its function means modifying the script itself; it does build a working system (that reproduces their provided pre-built images), but trying to customize it means having to invest into either creating an infrastructure around it or relying on brittle modifications that can start simple enough but that can evolve into a confusing monolith, wasting precious development time that could have just been used in lesser extent to study Buildroot and leaving the rest for the remainder of the project.

Integration can go into one of two directions: either mixing OpenIPC's userspace with Ingenic's kernel and into their build script, or discarding Ingenic's build system altogether and bringing their working kernel into Buildroot, adapting its configuration files and expanding packages to accomodate for Utopia's customizations. It is clear that the direction of integration to be followed was the latter, an approach that favours reusability of components, as Buildroot is much easier to extend, and that follows industry's best practice of integrating what already exists and is proven to work reliably.

3.2 Booting bare OpenIPC on the GOAT

The first step in building the prototype is having OpenIPC to boot on the GOAT in whichever state possible to provide a base for further advancements. The process is not immediate and requires some preliminary work, detailed in the following phases.

3.2.1 Flashing U-Boot

The GOAT was delivered with U-Boot already burned on the on-board NOR flash, but it cannot be assumed that this is always the case. It was indeed flashed unproperly during development multiple times preventing it from powering on correctly without external intervention, and even not taking these scenarios into account, defining a bootstrapping process for the bootloader itself can prove to be useful in the long term if needs arise for a construction line, though that specifically will be discussed in chapter 4.

To bring a blank board to the bootloader stage, two versions of U-Boot need to be compiled: one for booting from SD card and the other for booting from on-board NOR flash. The available source code from Ingenic provides pre-existing Make targets for the specific variant of CPU present on the GOAT, `T32NQ`, and for that, boot from both mediums: `make PRJ007_nq_msc0` and `make PRJ007_nq_sfcnor` respectively. Both targets produce file *u-boot-with-spl.bin*, for clarity's purpose one will be called *u-boot-with-spl-sd.bin* and the other *u-boot-with-spl-flash.bin*. An SD card has to be prepared carefully, by first repartitioning it, then formatting it and writing *u-boot-with-spl-sd.bin* to a specific offset. The commands are:

```
# repartitioning SD card
sudo fdisk /dev/sdb # assuming block device "sdb"
o                   # create a new partition table
n                   # add partition n
p                   # primary partition type
[Enter]             # default partition number
[Enter]             # default first section value
[Enter]             # default last sector value
w                   # write partition table to SD card

# formatting sdb1 after plugging and unplugging the SD
mkfs.vfat /dev/sdb1

# writing bootloader into 17KB offset
dd if=u-boot-with-spl-sd.bin of=/dev/sdb bs=1024 seek=17
```

The SD boot card is ready. Mounting it and copying *u-boot-with-spl-flash.bin* in the root finalizes the preliminary step before insertion into the board.

With the SD present into the slot, powering up the board while pressing `bootsel0` button instructs it to boot from the card, quickly presenting its interactive console. At this point the resident bootloader is ready to be burned onto the NOR flash chip. The procedure consists in loading the U-Boot image file into RAM, erasing the first 256KiB of flash and then writing the image:

```
# load image from SD into a safe RAM range
fatload mmc 0 0x80600000 u-boot-with-spl-flash.bin

# detect NOR flash chip in use
sf probe

# erase 256KiB of flash starting from address 0
sf erase 0 0x40000

# write 256KiB of data from RAM address 0x80600000 to flash starting
from address 0
sf write 0x80600000 0 0x40000
```

At this point the SD card can be removed and the board, when reset, will automatically start into U-Boot, pre-configured to execute a boot command after a short delay. This command is defined at compile time and will fail because no kernel image is present in memory. Such an image must be prepared alongside a root filesystem.

3.2.2 Building kernel image and root filesystem

Buildroot is in charge of building a kernel image plus a root filesystem and packaging everything to be readily flashed onto the board. To do so, it employs defconfigs, as explained in section 2.2. Since T32 is T31's successor, its defconfig has been taken as a starting point because of the many similarities they share. Ingenic provides together with the SDK the kernel, its .config and the toolchain for cross-compilation, all components that are configured inside the defconfig. To obtain a bare functioning systems, all packages that are not strictly needed have to be disabled using the `=n` suffix, starting from userspace packages that compile and integrate mostly utility software, their needed libraries and drivers.

After that, to set up a functioning toolchain, options to download them from known repositories online must be changed to point to external pre-installed Ingenic one; additional options must be added for Buildroot to detect correct version, headers, C library and supported features (for example, whether the toolchain

supports Fortran and OpenMP[1]).

The last meaningful modification is changing the location where the kernel is obtained from. Buildroot detects automatically from the string whether a download from internet has to be performed or if it refers to a local file: the GitHub URL present is changed to a path with prefix `file://` pointing to the kernel source code specifically compressed in tar.gz format [2]. The .config file points to *PRJ007_goat_defconfig* [3], provided by Ingenic in the Linux sources.

```
# Toolchain
BR2_TOOLCHAIN_EXTERNAL=y
BR2_TOOLCHAIN_EXTERNAL_CUSTOM=y
BR2_TOOLCHAIN_EXTERNAL_PREINSTALLED=y
BR2_TOOLCHAIN_EXTERNAL_PATH="$(TOPDIR)/../../utopia-custom/mips-gcc540-
    glibc222-r3.3.7.mxu2.cve2"
BR2_TOOLCHAIN_EXTERNAL_GCC_5=y
BR2_TOOLCHAIN_EXTERNAL_FORTRAN=y
BR2_TOOLCHAIN_EXTERNAL_OPENMP=y
BR2_TOOLCHAIN_EXTERNAL_CUSTOM_PREFIX="mips-linux-gnu"
BR2_TOOLCHAIN_EXTERNAL_HEADERS_4_2=y
BR2_TOOLCHAIN_EXTERNAL_CUSTOM_GLIBC=y
BR2_TOOLCHAIN_EXTERNAL_CXX=y
```

Listing 3.1: Fragment of T32 defconfig showing modified options for toolchain

Miscellaneous OpenIPC related options are adjusted accordingly, values like SoC family and model, to match the T32 variant; these are used in the build system to select correct files for the CPU, but since OpenIPC does not support this model yet, they remain mostly vestigial and have an effect only on the naming of end-of-compilation artifacts and in-system banners and hostname, as far as has been observed.

All relevant files for this part of the process have been relegated to a single unified folder *utopia-custom* that lives in the root of the OpenIPC repository.

Building the kernel image and root filesystem as configured in the defconfig is done by executing `make BOARD=t32_lite`. Buildroot will proceed to download the source code both for compilation tools and for the software specified in the configuration file, then will compile the first and use them to build the second. On a standard work machine like the one used for this project, time taken is

[1]Even if these features are not used, compilation tools complain about mismatch with declared and detected features, so these must necessarily be added.

[2]This format is commonly referred to as a *tarball*

[3]PRJ007 is the internal name used by Ingenic to refer to the GOAT board.

30 minutes to an hour. At the end of the process, a kernel image and a root filesystem are produced in different formats: relevant ones are `uImage.t32` and `rootfs.squashfs.t32`.

3.2.3 Populating residual MTD partitions

Flashing the two previously produced artifacts is straightforward as it follows the same method used to flash U-Boot, but with a few precautions. Exactly as before, the two images need to be first copied to the SD card. Powering up the board, now equipped with U-Boot for NOR flash, the following commands are then executed (with fictitious write sizes because they differ ever so slightly at every compilation):

```
# detect NOR flash
sf probe

# erase flash to make space for kernel image
sf erase 0x40000 0x280000

# load kernel image from sd into RAM
fatload mmc 0 0x80640000 uimage.t32

# write kernel image to flash
sf write 0x80640000 0x40000 0x192cfd

# erase flash for root filesystem
sf erase 0x2c0000 0x500000

# load root filesystem from sd into RAM
fatload mmc 0 0x808c0000 rootfs.squashfs.t32

# write root filesystem to flash
sf write 0x808c0000 0x2c0000 0x4db000

# erase remainder of flash from end of root filesystem to end of flash
sf erase 0x7c0000 +0x840000
```

These commands could be condensed, but have been shown separated for clarity. The '+' before erase size in the last command is used to round up to the next sector because the command is generated by a helper script that does not know that size in advance, and U-Boot can only erase in multiples of erase sectors.

Instructing U-Boot to load the kernel is only a matter of executing the boot command with `run bootcmd`. Doing so, however, would result in a kernel panic due to it not being able to correctly mount the root filesystem, underlying reason being that Ingenic's U-Boot source code includes predefined boot command arguments

that partition the flash in such a way that the root filesystem generated by Buildroot does not fit. The exact define directive responsible for this behaviour is the following:

```
// original define
#define CONFIG_BOOTARGS BOOTARGS_COMMON " init=/linuxrc rootfstype=
squashfs root=/dev/mtdblock2 rw mtdparts=sfc0_nor:256k(boot),2560k(
kernel),2048k(root),-(appfs)"
```

Listing 3.2: Excerpt from `uboot/include/configs/PRJ007.h`

The predefined MTD partitioning was intended for Ingenic's smaller root filesystem and it's defined in the U-Boot configuration header file for the GOAT board `PRJ007.h`. This means that it can be changed at compile time, but just performing a variable change and environment save step suffices for the project's purposes:

```
# set correct bootargs variable
setenv bootargs console=ttyS1,115200n8 mem=64M@0x0 rmem=64M@0x4000000
init=/linuxrc rootfstype=squashfs root=/dev/mtdblock2 rw mtdparts=
sfc0_nor:256k(boot),2560k(kernel),5120k(root),-(appfs)

# save environment to flash so it persists reboots
saveenv
```

Finally, executing `run bootcmd` yields a boot screen [Fig 3.2] with OpenIPC's login prompt.

Figure 3.2: First boot of OpenIPC on GOAT board.

3.3 Correcting and refining the resulting system image

Figure 3.2 reveals many errors issued before the login prompt; others have been observed while testing system functions during early stages of development after obtaining a booting OpenIPC. In order to reach an acceptable quality level for the final prototype, supplemental provisions have to be made: some are of notable importance and are absolutely needed for correct operation, while others are either infrastructural and offer aid in development, or are for completing its feature set.

3.3.1 Enabling write support with kernel patching

One of the first issues that have been noticed was a lack of write permissions on the root filesystem. Any attempt of writing a file was followed by an error reporting

`"Read-only file system"`. Investigating the init script, whose responsibility is mounting the root filesystem, it was found that an early exit condition was triggered by the absence of OverlayFS in the list of supported filesystem types. OpenIPC requires it for its architecture: a small user writeable partition (corresponding to the residual flash memory after all the others), on top of a read-only root filesystem formatted as a SquashFS image.

```
1   # init consults /proc/filesystems to query the kernel for supported
    filesystem types
2   grep -q overlay /proc/filesystems || exit 1
3
4   # [...]
5
6   # partial logic responsible for correctly mounting the overlayfs
    filesystem
7   if grep -q overlayfs /proc/filesystems; then
8       if ! mount -t overlayfs overlayfs -o lowerdir=/,upperdir=/overlay,ro
     /mnt; then
9           umount /overlay
10          exit 1
11      fi
12  else
13      overlay_rootdir=/overlay/root
14      overlay_workdir=/overlay/work
15      mkdir -p $overlay_rootdir $overlay_workdir
16      if ! mount -t overlay overlay -o lowerdir=/,upperdir=
    $overlay_rootdir,workdir=$overlay_workdir /mnt; then
17          umount /overlay
18          exit 1
19      fi
20  fi
```

Listing 3.3: Fragment from OpenIPC init script

This feature was merged in Linux 3.18, while both OpenIPC's and this project's kernel lack support for it, being on version 3.10.14. OpenIPC developers address this problem by applying an out-of-tree OverlayFS patch, and to bring Ingenic's kernel to specifications the same manual patchwork is needed. Commit history for the `ingenic-t31` branch of OpenIPC's Linux fork [18] shows no attribution for the patch's origin, so in this case it is extracted directly from the repository and applied to the kernel source with the following commands:

```
# download OpenIPC's Linux fork and cd into it
git clone https://github.com/OpenIPC/linux.git
cd linux

# extract patch from the commit that introduces OverlayFS functionality
git format-patch -1 e7666aa11e90295bf232148916be91fb884dca2e --output=
openipc-overlayfs.patch

# copy patch to right folder
cp openipc-overlayfs.patch ../kernel-3.10.14

# cd into Ingenic's kernel folder and apply patch
cd ../kernel-3.10.14
patch -p1 < openipc-overlayfs.patch
```

The kernel source needs recompaction into tarball to be usable again by Buildroot.

The feature first needs enabling to be built into the kernel. This can be achieved by either adding the line `CONFIG_OVERLAYFS_FS=y` to the .config file or by virtue of the `menuconfig` utility.

3.3.2 Creating a customizations package

Buildroot provides an extensive selection of software in the form of packages, as explained in section 2.2: if it is present in the repository, it is always advised to use this facility for installing it in the final resulting image. For binaries and files of external making, br2-external trees are also an option, but to achieve the most granular and flexible control on how they are produced and placed in the image (e.g. to manage permissions and symlinks), indeed a custom package is the best alternative.

For this project, the `utopia-customizations` has been created. It resides in the `general/package` folder inside OpenIPC's root and respects the specifications necessary to be detected by Buildroot's makefiles. It is employed to copy binaries, configuration files, scripts, proprietary kernel modules and during development it has also been utilized to symlink libraries for the correct operation of specific software.

This approach of making modifications localized, possibly keeping everything in the same folder and avoiding altering already existing files, applied generally throughout the project, allows further adjustments for Utopia to be built as only additions to the original repository, without affecting files that may prove useful in the future. This way, the project is intended as a supplemental patch applicable to the OpenIPC repository.

Below are adjustments that were implemented and later integrated through the aforementioned package.

init script substitution

OpenIPC's original init script expects MTD partitions to be called a certain way. Nothing prevents editing the boot command to match it, or even the init script itself, but following the *only additions permitted* approach explained above, an alternative version of the script has been devised. It consists simply in a copy of the original init with MTD partitions modified, but, being separated, modifications can be done without affecting the original.

The new init is called `init_t32` and resides in the `files` folder of the utopia-customizations package. On U-Boot, the `bootargs` variable's init argument needs to be corrected to `init=/init_t32`.

Enabling userspace read/write support for U-Boot environment

OpenIPC makes heavy use of the U-Boot environment to store and retrieve data about the SoC itself: total available RAM, sensor mounted on the board, pre-loaded ethernet address and many others; userspace commands `fw_setenv` and `fw_printenv` are used for this purpose. Normally these tools interact directly with the MTD subsystem, by reading and writing from flash partitions specified at compile time[4]; OpenIPC itself, in its boot command arguments, specifies an MTD partition just for the U-Boot environment. Though, they are also predisposed to check the presence of configuration file `/etc/fw_env.config`, so this method is used to instruct them on the environment's location. The file uses a basic format and every line has (for NOR flash based systems): device name for the MTD partition, offset from the device, environment size, flash erase sector size and number of sectors, but the last value can be omitted. To compute the correct offset and environment size, defines `CONFIG_ENV_OFFSET` and `CONFIG_ENV_SIZE` from `PRJ007.h` have been consulted, with the first being assigned value $240 * 1024 = 245760$ bytes, while the other $16 * 1024 = 16384$ bytes. Final configuration file has this single line:

```
1 # MTD device name       Device offset   Env. size       Flash sector size
        Number of sectors
2 /dev/mtd0               0x3c000         0x4000          0x4000
```

[4]Source code for these tools is included with U-Boot in subdirectory `/tools/env`.

3.3.3 Bringing the system to full functionality

Up until this stage of development, work had been focused on solving issues about system startup, all the bugs and errors that prevented a clean boot from happening. This last sub-section will instead revolve around completing its feature set, bringing it close to the specifications provided by Utopia: providing a drop-in replacement for their cameras used in production systems.

Miscellaneous software packages

In order to speed up development and compilation time, nearly all software packages specified in T32's defconfig had been disabled. For most of them, to regain functionality it is sufficient to re-enable them with suffix `=y`; they will be compiled and added to the produced image. In some cases, instead, some have been removed as they consisted of either SDK libraries intended for T31 or firmware/drivers for network adapters left unused. Those that remain are utility packages that provide SSH access through a small footprint server, tools like `curl` for generic internet data transfer and `ipctool` to check camera's hardware, various codecs for audio formats, and CLI tools and libraries to parse JSON and YAML data.

To aid development and for experimentation on the board itself, packages for editor `nano` and command `tree` to display folder contents in a tree-like manner were also enabled.

Evaluating streaming solutions

The very last, and arguably most important, addition to the system is a streamer software that can interface with the on-board sensor and provide an RTSP feed that can then be parsed by Utopia's production machines. OpenIPC by default adopts Majestic, made by the same developers. It is tightly integrated with the cameras it is able to run on, by leveraging the specific SDK libraries provided by the vendors for each of the CPU they are based on. The main issues lie in it being a closed source endeavour, indeed it is distributed as a pre-built binary with different versions for each SoC, and also not supporting the T32 altogether.

An attempt has been made during research to bring it to a functional status, by trying to include all the necessary SDK components and libraries [5], but the effort did not prove fruitful as the software managed to start up and display part of a hardware probing routine, before becoming completely unresponsive.

[5]Including the musl C library, which is the one OpenIPC uses. This project relies instead on the GNU C Library since it is widely used and also one of the only supported ones (together with uClibc) by Ingenic's build tools.

An analysis has also been performed with tools `strace` and `gdb`. The first was used to intercept and record syscalls to discover at which point the program was stalling; it was found that it was stuck in an infinite loop of calling `ioctl` and receiving an error code, likely indicating incompatibilities between T31 SDK and GOAT board when initiating communication with hardware peripherals. gdb was instead used to perform runtime inspection, but its use was deemed impractical as Majestic enables an hardware watchdog when starting, meaning that debugging sessions could only last 5 minutes before being interrupted and requiring restarting.

These difficulties prompted the search for another solution, which was found in Ingenic's library of pre-compiled tools: the `Carrier` server. This also is a closed source solution, but it was shipped with T32 support and it works sufficiently well for the purposes of a prototype. It recognizes the image sensor correctly, provides primary and secondary RTSP streams with high and low quality respectively and its options (bitrate, resolution, color settings and many others) can be configured through a control board software provided in the same bundle. Reliance on an external program that has to run on a pc to control its working severely limits its usefulness in a production environment, but this and other discussion points are addressed in chapter 4.

Together with a basic bash script to load on startup the necessary kernel modules and drivers for the image sensor, Carrier can start and provide the streaming service. This marks the end point for this project.

Chapter 4

Future works

The work and research in this thesis were performed to create a prototype that could serve Utopia to bring their cameras to the next generation of Ingenic processors and to avoid supply line issues. By its very nature, some steps are required to bring the project to production-ready status.

At the time of writing, preliminary testing has been performed on both the GOAT board provided by Ingenic and an in-house board developed by Utopia. Every successive iteration of the prototype will need to be stress-tested and checked for compliance with all requested functional requirements, as full integration with Utopia's production systems has yet to be achieved. A plan for mass fabrication also must be devised, as the process of flashing the board up until now has been mostly manual, with only the help of a script to generate the necessary U-Boot commands and the rest performed step-by-step on a workstation. The necessity of loading each camera with custom per-board configurations (e.g. IP address, sensor drivers) may be equally satisfied by specifically designed equipment.

On the software side, Carrier server is a valid choice for a prototype, but it would show its weaknesses in a production setting. The options that can be configured at startup through command line arguments are very few and only cover the essentials for a streamer software, like FPS and frame dimensions. To obtain finer control without depending on external software, documentation for Carrier's API endpoints is needed as the one provided is severely lacking. Utopia also requested for future developments if edge AI features could be obtained, but a separate SDK (*Magik AI*) is required to access the board Neural Processing Unit, or NPU, and models that can run on. Achieving fast development for a software solution that could both integrate its AI features and provide easily configurable parameters for the feed produced by the camera can be made possible, Ingenic willing, with Carrier's source code. If it proves not available though, more development resources will need to be diverted to building a software that correctly utilizes Ingenic's SDK capabilities.

Finally, given that OpenIPC is an open source project, a consideration could made about sharing the results of this project back to the community by submitting pull requests to the original repository. This way, as OpenIPC was employed in the beginning to work with T31 based boards, support could be offered to users that need it on T32 based variants. Nevertheless, it is not a direct process, as the developed code requires clearance from a legal advisor[1] to be published and the codebase reworking to adhere to OpenIPC contribution standards.

[1]Some of the processes and informations utilized throughout the project are covered by Non-Disclosure Agreements.

Chapter 5

Conclusions and final thoughts

Working with embedded Linux systems is a known and well-trodden field. Open source and standardized tooling enable porting a distribution to an unsupported board with relative ease and quick development time, as has been demonstrated.

Though, something that is not immediately apparent from the thesis is how much time was spent on debugging efforts. Many issues that were encountered during development did not present obvious solutions, requiring careful considerations in how to proceed forward and what approach would be best. The software used, however open it was, revealed itself to be sometimes obtuse, and many hours were spent researching, reading documentation and comparing different options. The architecture of open source project is often not clear from comments, documentation and commit messages alone, leaving much to interpretation and testing.

However, all of this is customary in this context, and gaining experience can only happen by interacting with the gritty details of reality. What has been achieved in this thesis work can definitely be improved, but poses solid bases to solve the real problem of ensuring a stable supply line by upgrading working but obsolete hardware with the new T32 component. Integration with standard operating procedures ensures a clear line for further developments, bringing the project from prototype to fully incorporated in Utopia's solutions.

Bibliography

[1] Buildroot developers. *Buildroot 2025.11 manual*. 2025. Chap. 1: About Buildroot. URL: https://buildroot.org/downloads/manual/manual.html (cit. on p. 3).

[2] Buildroot developers. *Buildroot 2025.11 manual*. 2025. Chap. 15: How Buildroot works. URL: https://buildroot.org/downloads/manual/manual.html (cit. on p. 4).

[3] The kernel development community. *The Linux Kernel documentation*. Chap. Kconfig Language. URL: https://docs.kernel.org/kbuild/kconfig-language.html (cit. on p. 7).

[4] *General MTD documentation*. Chap. MTD overview. URL: http://linux-mtd.infradead.org/doc/general.html (cit. on p. 8).

[5] Chris Simmonds and Frank Vasquez. «Mastering Embedded Linux Programming». In: 2021. Chap. 9: Creating a Storage Strategy, Section: Storage options. ISBN: 978-1789530384 (cit. on p. 8).

[6] Chris Simmonds and Frank Vasquez. «Mastering Embedded Linux Programming». In: 2021. Chap. 9: Creating a Storage Strategy, Section: Accessing flash memory from Linux - Memory technology devices. ISBN: 978-1789530384 (cit. on pp. 8, 9).

[7] Junjiro R. Okajima. *aufs website*. Chap. 2. Download. URL: https://aufs.sourceforge.net/ (cit. on p. 9).

[8] *UnionFS website*. URL: https://unionfs.filesystems.org/ (cit. on p. 9).

[9] *unionfs-fuse repository*. URL: https://github.com/rpodgorny/unionfs-fuse (cit. on p. 9).

[10] Neil Brown. *OverlayFS documentation from Linux Kernel tree, branch linux-3.18.y*. Chap. Overlay Filesystem. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/filesystems/overlayfs.txt?h=linux-3.18.y (cit. on p. 9).

[11] Neil Brown. *OverlayFS documentation from Linux Kernel tree, branch linux-3.18.y*. Chap. Upper and Lower. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/filesystems/overlayfs.txt?h=linux-3.18.y (cit. on p. 9).

[12] Neil Brown. *OverlayFS documentation from Linux Kernel tree, branch linux-3.18.y*. Chap. Directories. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/filesystems/overlayfs.txt?h=linux-3.18.y (cit. on p. 9).

[13] Neil Brown. *OverlayFS documentation from Linux Kernel tree, branch linux-3.18.y*. Chap. whiteouts and opaque directories. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/filesystems/overlayfs.txt?h=linux-3.18.y (cit. on p. 10).

[14] Peter Jay Salzman, Michael Burian, Ori Pomerantz, Bob Mottram, and Jim Huang. «The Linux Kernel Module Programming Guide». In: 2026. Chap. 1.3 What Is A Kernel Module? (Cit. on p. 10).

[15] Linus Torvalds. *Linux Syscall clause of the GPL 2.0 license*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/COPYING?h=linux-3.14.y (cit. on p. 10).

[16] Chris Simmonds and Frank Vasquez. «Mastering Embedded Linux Programming». In: 2021. Chap. 3: All about Bootloaders. ISBN: 978-1789530384 (cit. on p. 11).

[17] Chris Simmonds and Frank Vasquez. «Mastering Embedded Linux Programming». In: 2021. Chap. 3: All about Bootloaders, Section: U-Boot - Using U-Boot - Environment variables. ISBN: 978-1789530384 (cit. on p. 14).

[18] OpenIPC Project. *Linux Kernel Source: `ingenic-t31` branch*. https://github.com/OpenIPC/linux/tree/ingenic-t31. Accessed: 2026-03-07. 2026 (cit. on p. 24).