CS39440 Major Project

Institute of Mathematics, Physics and Computer Science

Aberystwyth University

Improving Player Driver Support for Sony Pan-Tilt-Zoom Cameras

---

Author: Oliver Saunders (ols7@aber.ac.uk)

Supervisor: Dr. Fred Labrosse (ffl@aber.ac.uk)

11 August 2015

Version: 2.0 (Release)

This report was submitted in partial fulfilment of a BSc degree in
Artificial Intelligence and Robotics (GH76)

Declaration of Originality

In signing below, I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for plagiarism and other unfair practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the sections on unfair practice in the Students' Examinations Handbook and the relevant sections of the current Student Handbook of the Department of Computer Science.
- I understand and agree to abide by the University's regulations governing these issues.

Signature  (Oliver Saunders)

Date 11/08/2015

Consent to share this work

In signing below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Signature  (Oliver Saunders)

Date 11/08/2015

Ethics Form Application Number

The Ethics Form Application Number for this project is: **833**

Student Number

Acknowledgements

Throughout this project, I have been extremely fortunate to receive a huge amount of help, support and encouragement from many different people. I would like to take this opportunity to express my gratitude to each and every one of you.

In particular, I would like to thank my supervior Dr. Fred Labrosse, for the incredible amount of support, advice and wisdom he has shared with me over the past four months. The trust he showed in me each time he cheerfully handed me hundreds of pounds worth of hardware was much appreciated! I would also like to express my gratitude to Patricia Shaw for acting as the second marker for this project and encouraging me to consider new ideas when developing the driver. Additionally, I am grateful to the staff and students within the department, particularly to Neil Taylor for guiding the process and to the former students who made their dissertations available for me to read.

My friends and colleagues at *Ultracomida* have made working part-time whilst writing this dissertation a real pleasure, as well as keeping me well-supplied with Manchego cheese. I could not wish for a better place to work. I would also like to thank Dave Price and the other organisers of the Aberystwyth amateur radio training course for offering me a vastly appreciated break from writing once a week, as well as a reason to walk up Penglais Hill rather than sit in front of a monitor.

It would be remiss of me not to profusely thank Brian Gerkey, Brad Tonkes et al. for their work on the original driver that this work is based on. I would also like to extend my appreciation to Damien Douxchamp, the creator of *libVISCA*, which was instrumental in providing a detailed account and reference implementation of the *VISCA* protocol. In a similar vein, I would like to express my sincere appreciation to the wider open source community for providing tools like LaTeX, *Doxygen* and *Vim* which were used in this project.

On a personal note, I would like to thank my family for their support and also my friends Dale, Katie, Niroo, Ben and Rowan for proofreading. Even though they were busy with their industrial years, they took the time to read this over and offer feedback. Most importantly of all; I would like to thank my partner Leah for her selfless support, optimism and patience, without which this dissertation may never have been completed.

Finally, I would like to acknowledge the town of Aberystwyth, a magical place by the sea which has become my home away from home. I was only here for three years, but the memories I made will last a lifetime.

Abstract

Pan-tilt-zoom cameras are used for a wide variety of tasks including CCTV, obstacle detection and automated object tracking. Uses for these cameras range from mundane lab exercises to cutting-edge robotic exploration on the surface of Mars.

This dissertation presents a thorough analysis of and a number of improvements to the current publically available hardware driver for using Sony *EVI* series pan-tilt-zoom cameras with the open source *Player* robotics framework.

A thorough investigation into the software requirements, server architecture, relevant communications protocols and hardware characteristics is followed by a detailed account of the iterative refactoring and development process used to produce a modified driver which represents a significant improvement on the existing system.

The design of this modified driver is comprehensively explained, including justification for significant decisions made during the refactoring process. Following this, implementation details are provided for the improved driver and related software deliverables. This improved implementation includes novel features such as hardware calibration, which are explained in detail.

The challenges of testing, documenting and debugging a hardware driver are examined and measures put in place in response to these are discussed.

Finally, the software and wider project are critically evaluated and consideration is given to future possibilities relating to this work.

Contents

Figures

Tables

Listings

Chapter 1

Background, Analysis and Process

> If you wish to make an apple pie from scratch, you must first invent the universe.
>
> Carl Sagan

The primary aim of this project was to improve the existing hardware driver provided for using Sony *EVI* pan-tilt-zoom (*PTZ*) cameras with the open source *Player* [27] robotics framework, including the production of significant amounts of documentation to assist future development efforts and minor changes to the core server to allow for additional functionality.

Figure 1.1: Front and back views of the *EVI-D70* PTZ Camera. [Sony Electronics Inc.]

This dissertation consists of five chapters. This chapter provides an introduction to the technologies used in this work, explores preliminary research, presents an analysis of the existing system and explains the motivation behind this project. The objectives of the project will be clearly defined and referred back to in subsequent chapters.

After this introduction, the design of the improved driver will be explored and justifications will be provided for significant decisions made during the refactoring process.

Next, implementation details will be discussed and novel implementation features such as automatic hardware calibration will be introduced and explained.

Following this, discussion will turn to the challenges of testing and debugging a hardware driver and how we can minimise these in this and similar projects.

Finally, a comprehensive evaluation of the improved driver and the wider project will be conducted to determine whether the objectives outlined in this chapter have been met. There will also be an overview of what future work in this area might entail and some closing remarks.

1.1 Background

Prior to discussing the improvements made to the existing system, it is important that we familiarise ourselves with the technologies involved before going into greater depth in later chapters. It is also important to gain an understanding of the current state of *Player* drivers, particularly the existing Sony PTZ driver implementation which forms the basis of this work.

Gaining a thorough understanding of what facilities *Player* provides was an important aspect of preliminary research, as well as examining the target hardware and the means by which the hardware, the existing driver and the *Player* server interact.

As previously mentioned, this work is based on and made possible by the existing Sony PTZ driver (known as *sonyevid30* [9]) which offers various degrees of supports for several Sony *EVI* cameras, although it has a number of issues in its current state. Before work began on the improvements to this system, it was important to examine the driver and assess the degree of refactoring required, potential for the addition of new features and the best approach to take regarding the development process.

Initial research demonstrated that the current driver was mostly functional for two of the three target hardware models and provided a suitable basis for a refactoring and development effort which aimed to produce an improved and fully documented system. There are a number of resources which helped enable this process, perhaps the most important being the *libVISCA* [6] library which provides a comprehensive reference implementation for controlling these cameras using software and (alongside the existing source code) provided enough implementation details to make documenting, refactoring and extending the existing driver easier to do.

There are a number of resources outlining the structure of *Player* drivers and providing the necessary information to implement them. Collecting and studying these resources was a key area of research during the early stages of the project. Even extending an existing *Player* driver requires a good degree of knowledge about the inner workings of the relevant software and hardware.

Research into the current driver and other similar systems highlighted a number of concerns, including an abundance of cryptic naming conventions, poorly structured functions and partially implemented features.

A significant source of motivation for this project was the desire to improve the driver to more closely adhere to modern software design principles, as outlined in a number of books [7, 8] which are highly-regarded within the software industry.

As well as preserving existing functionality and providing new features where possible, the improved driver also needed to be easy to understand and easily extendible by third-parties. This was an important aspect of the project as *Player* is open source software (released under the GNU General Public License [24]) and is widely used [5] in post-secondary education and industry throughout the world.

Player could therefore be considered a 'flagship' open source project and with this in mind, improving the driver so that it is easy to extend and adapt was of the utmost importance. This project set out to provide proof that hardware drivers can be clearly-structured, intuitive and easy to modify without sacrificing performance or functionality.

Within the following pages, we will explore the three major technologies used within this project;

the *Player* server, the pan-tilt-zoom camera hardware and the communications protocol linking them together.

1.1.1 *Player*

Player is described [27] as a *robot device interface* which acts as a *hardware abstraction layer* for various robotic hardware; including laser scanners, grippers, PTZ cameras and a vast array of other devices.

Hardware drivers for *Player* implement an interface for a class of devices rather than being written for a specific device, in much the same way that an operating system provides abstraction when writing hardware drivers.

For example, an operating system driver might implement a *mouse* or *USB mass storage* interface which reduces the need to reimplement features shared between all devices of that type.

Figure 1.2: The official logo of *Player*. [Player Project]

Player is often used directly on robotic hardware (e.g. *Pioneer* [26] robots) if sufficient computing resources are available, in this case a full POSIX-compliant operating system. It can also run on a computer connected to robotic hardware via a network or a direct physical connection, which makes the server very flexible and suitable for remote use.

The driver was originally designed for a directly connected camera and was tested as such throughout this project, however controlling a camera remotely is theoretically possible.

1.1.2 Sony PTZ Cameras

The improved driver focuses on supporting three similar models of pan-tilt-zoom camera developed by Sony, these have model numbers beginning with *EVI*. Much of the groundwork for this has already been laid in the existing driver, with the *EVI-D30* and *EVI-D70* being well supported.

Other models of *EVI* camera should work to some extent, however the driver will default to the profile of an *EVI-D30*, allowing only a basic level of control. It is worth noting that a number of cameras from other manufacturers (and Sony themselves) use the same communications protocol as the three supported cameras, expanding the driver to explicitly support these would be simple.

The specific models of hardware supported by the new driver are as follows:

- **EVI-D30**: The first *EVI* camera from Sony, the existing driver was designed for and works well with this camera.
- **EVI-D100**: A newer model of *EVI* camera with greater pan and tilt speeds than the *D30*, the existing driver has some issues with this camera.
- **EVI-D70P**: The newest model of *EVI* camera with a significantly wider pan and tilt range, the existing driver has severe issues with this camera.

We can see that the *D30* is well-supported by the existing driver, but the other cameras were found to have significant issues during testing. This is discussed in more detail later in this chapter.

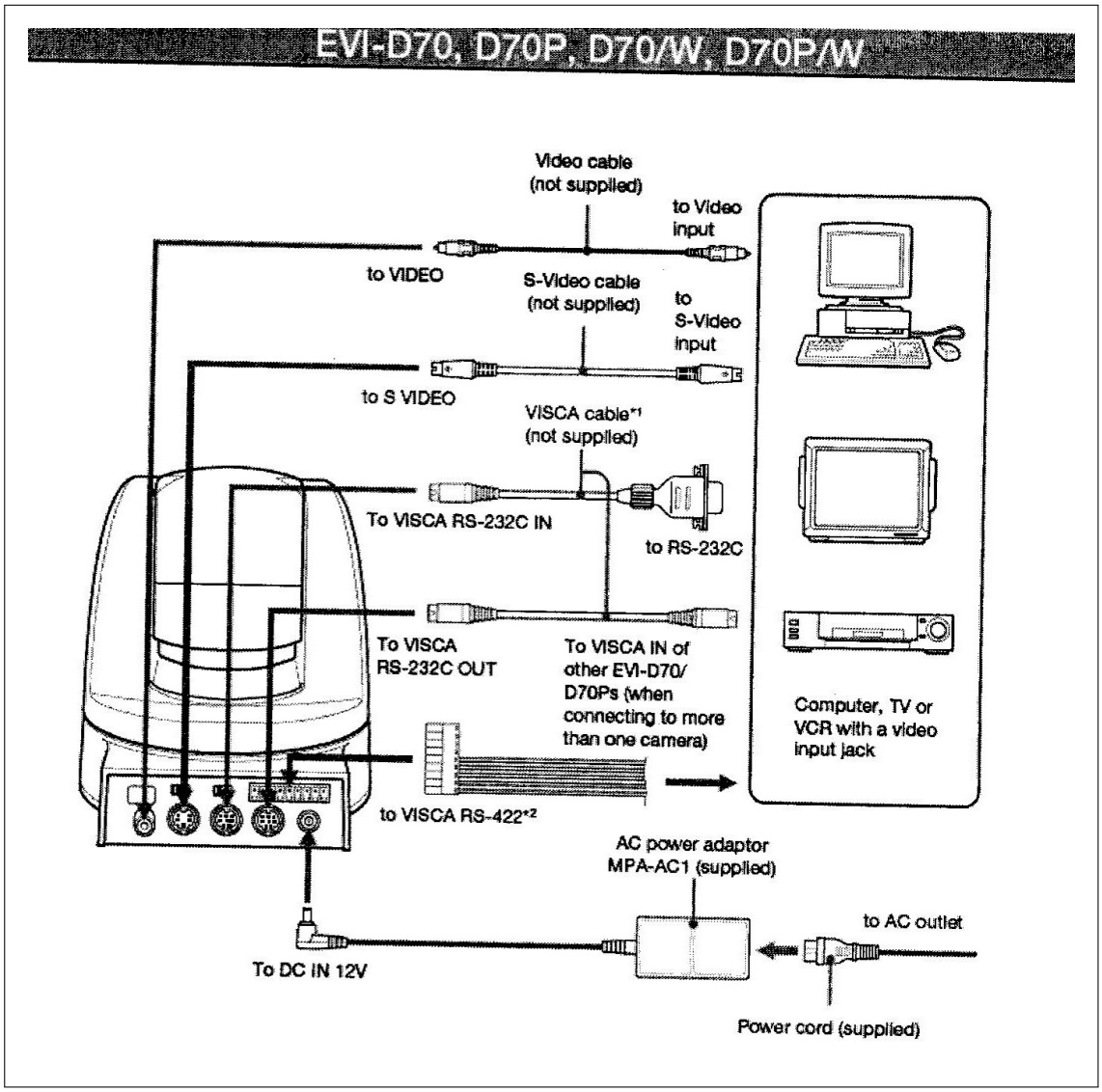


Figure 1.3: Connection diagram for *EVI* cameras. [Sony EVI-D70P Technical Manual, pg. 4]

All *EVI* cameras have a standardised communications interface known as *VISCA*. The physical interface is also shared between cameras, the connection diagram shown in **Figure 1.3** above applies to all three supported models, as well as variants such as the *EVI-D70/W*.

Connection to a computer (*controller* in *VISCA* terms) is achieved using a *VISCA* cable which connects to the input on the camera and terminates in a standard *RS232* (DB-9) serial connector. As many portable computers and workstations developed in the last five years lack a dedicated serial port, it is common to use an *RS232* → USB adapter to connect the controller to the camera.

1.1.3 *VISCA*, (The) Video Systems Control Architecture

As discussed previously, the serial protocol all *EVI* cameras use to communicate is known as *VISCA*, which stands for *Video Systems Control Architecture*.

Table 1.1: *VISCA RS232* Parameters

Parameter	**Value**
Communication Speed	9600bps
Start Bit	1
Stop Bit	1
Data Bits	8
Parity	None
Byte Order	MSB First

This is a relatively simple protocol which is detailed in a specification [20] available online, from which much of the following information is derived.

The required *RS232* parameters for connecting to a *VISCA* enabled camera are listed in **Table 1.1** to the left.

Communication is carried out by sending and receiving hexadecimal packets, which have three components:

- Header (1 Byte): Contains the address of the target. The controller address is always 0, the camera address can range from 1 to 7
- Message (1-14 Bytes)
- Terminator (1 Byte): Always `0xFF`

Given this, we see that a *VISCA* packet can vary from 3 to 16 bytes in size. It is also apparent that up to seven cameras can be daisy-chained (connected in sequence) and used by a single controller. A packet may be broadcast to all connected cameras using the reserved target address `0x88`.

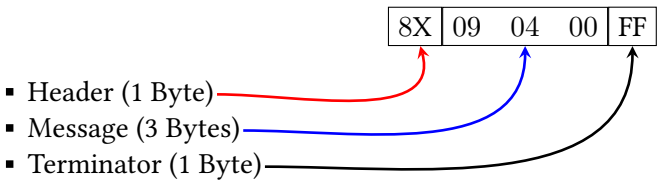
A packet can contain either a *command* or an *inquiry*. An inquiry returns data from the camera (such as the maximum pan speed) whereas a command changes the state of the camera. Commands and inquiries can be related to pan, tilt, zoom or miscellaneous hardware functions such as power status.

By default an *EVI* camera has a command buffer of two commands, if a command is already in progress and a new command is received it will be placed in a queue to be executed once the first command is complete.

If two commands are already queued the camera will not be able to process additional commands, inquiries are not typically restricted by this. "Even when two command buffers are being used at any one time, an [EVI] management command and some inquiry messages can be executed"[1].

The existing driver for *EVI* cameras makes use of the command buffer but the modified driver does not, for reasons which will be discussed in the next chapter.

The anatomy of a *VISCA* packet is relatively simple, let us look at an example of an inquiry sent by the controller to determine whether the camera is powered up:

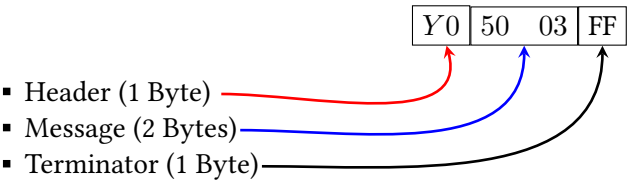


[1] EVI-D70P Technical Manual, pg. 33

- `0x8X` is the target address, where `X` can be any value from 1 to 7 or 8 for a broadcast to all connected hardware.
- `0x09` represents a *VISCA* inquiry, commands are represented using `0x01`.
- `0x04` is for miscellaneous functions (i.e. not pan / tilt / zoom).
- `0x00` is the command to return the power status.

The correct hexadecimal string for various hardware functions can be found using one of several technical manuals provided by Sony. Several pages of the *EVI-D70P* command listings are provided in Appendix A.

An example reply for a power inquiry would be as follows:



- `0xY0` is the header, the target address is set to the controller (0). `Y` represents the socket number (i.e. the buffer) the command or inquiry was sent to.
- `0x50` indicates that what follows is the reply to an inquiry packet.
- `0x03` is an information return and signifies that the camera is on standby, a return of `0x02` would indicate that it was powered up.
- `0xFF` is the terminator which accompanies all packets.

1.2 Objectives

Now that we are familiar with the basic technologies and concepts involved, it is important to define exactly what the objectives of this work are. We have discussed certain aims; namely restructuring, documenting and improving the existing system, but there are a number of additional considerations.

For the work to be considered a success it had to provide functional parity with the existing system and offer additional features, bug fixes or other incentives to justify its use over the unmodified driver included with *Player*, both by end users and developers.

One of the unique features the modified driver possesses is hardware calibration. This fixes a major issue with the existing system which is that there are a number of (often inaccurate) hard-coded values used to represent hardware limits. This means the unmodified driver does not fully exploit the capabilities of all supported hardware.

For example; the *EVI-D70* is limited to a maximum tilt of only 25° with the unmodified driver, with the new system this is correctly set to 90°.

As the software was refactored using agile methodologies, it seemed sensible to derive the requirements for the modified system using a set of *user stories* [4] which represent use cases for the software. Any functionality in the existing system which contradicted or was not required by a user story could safely be altered or removed. Obvious stories include the ability to pan, tilt and zoom at various speeds and to be able to control the camera through a serial connection, most of which *sonyevid30* already does to some degree of success.

The main requirements were discussed with Dr. Fred Labrosse (the project supervisor) and these discussions provided a series of user stories that would define what the modified system was expected to do and what the failings of the current system were.

1.2.1 User Stories: The *Player* Mailing List

As is the case with many open source projects, the *Player* project possesses a mailing list. The initial requirements for the software were established in a series of face to face meetings with Dr. Labrosse, however more were made clear in a post [10] he had sent to the mailing list in 2013.

Within this post, he highlighted several issues with the current driver which can be summarised as follows:

- The *sonyevid30* (existing) driver claims to support the *D30*, *D70* and *D100* cameras, but the *D70* exhibited a number of issues such as zoom being reversed. The *D70* support in *sonyevid30* is implemented differently to the other hardware and this inconsistency causes issues.
- The driver detects the *D100* as being a *D30* which does not allow full access to the capabilities of the hardware.
- There is an issue with small zoom increments being ignored due to using inefficient integer arithmetic with truncation rather than floating point with rounding in a function which converts camera viewing angles from radians to internal camera units.
- There is no support for controlling camera zoom speed despite it being supported by the hardware and by *VISCA*.

An important objective of the refactoring and development process was to eliminate the issues outlined above. An additional requirement gathered from this was that changes needed to be made to the *Player* server and interfaces to support controlling camera zoom speed.

Modifications to an open source project should ideally take the form of a patch which can then be contributed to the project, this highlighted the need for an additional deliverable.

1.2.2 Summary

The time has come to summarise all of the objectives of the project, including those related to the driver itself, development methodology and other deliverables. These objectives will be evaluated in the final chapter of this dissertation to determine the extent to which they have been achieved.

1. To create an improved hardware driver, built on the existing system with all of its core functionality intact.

2. To provide an implementation with a modern, clean and consistent code-base that encourages modification and extension.
3. To implement new features such as hardware calibration which make the modified driver outwardly superior to the existing one.
4. To produce comprehensive documentation for using the driver so it can be easily extended and integrated into projects.
5. To produce a patch file containing changes to *Player* to add zoom speed control and to submit this to the maintainers.
6. To produce a hardware test function which demonstrates the functionality of the modified driver.

These six objectives provided clear guidelines for the work that needed to be done and kept the project on track. The final two objectives were identified during the analysis stage of the project, which we will now discuss.

1.3 Analysis

Once background research had been completed and most of the objectives of the project had been identified, it was time for more detailed analysis. The primary deliverable was the driver, from this point on "the driver" refers to the **modified** driver created as a result of this project, which is (rather unimaginatively) named *sony-ptz* to differentiate it from *sonyevid30*, itself a somewhat inaccurate name as it offers support for more than just the *EVI-D30*.

Player provides an API for writing drivers which specifies the required functions for interacting with the server and how to properly implement the message processing architecture that is a key component of the hardware abstraction layer. The following pages describe relevant topics encountered during the analysis phase of the project.

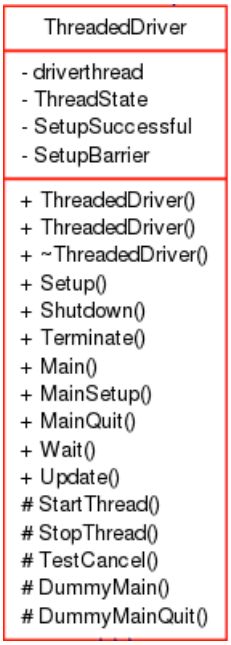


Figure 1.4: Class diagram for the *ThreadedDriver* class.

1.3.1 Player Driver Architecture

Drivers can either be created as shared libraries (*.so*, *.dylib*, *.dll*, etc.) which are loaded at runtime or as static drivers compiled within the server.

It is far more efficient to create a *plugin* driver because static drivers require recompiling the entire source tree (including *Player*) whenever changes are made. One of the earliest design decisions in any driver is the choice between creating a plugin or a static driver, *sonyevid30* is implemented as a plugin driver and it was decided that it was best to keep the modified driver the same way.

The *Player* server is object-oriented and all drivers inherit from the *Driver* class[2].

This class is then extended by the *ThreadedDriver* class, which several drivers (including *sonyptz* and *sonyevid30*) use as their base class. All drivers sharing this base class are multithreaded (using *PThreads*) so that the main function runs in a separate thread to the rest of the application, this allows for non-blocking message processing.

Several functions (or methods, depending on your choice of terminology) are mandatory [13] in any class that extends *ThreadedDriver*, regardless of the purpose of the driver.

Table 1.2: Required functions in a *Player* driver

Function Name	Purpose
`player_driver_init`	C function which acts as the entry point for the driver, calls the registration function.
`SonyPTZ_Register`	Registers the driver with the server, indicating supported interface(s) and the factory function to create a new driver instance.
`SonyPTZ_Init`	Factory function, returns a pointer to a new driver instance, accepts a driver configuration file as a parameter.
`MainSetup`	Runs prior to the Main function, used for general setup tasks such as opening a serial port.
`Main`	Runs in a separate thread, processes messages and runs as long as the driver has subscribers.
`MainQuit`	Called before termination, used for clean-up tasks such as closing serial ports and powering down hardware.
`SonyPTZ`	Mandatory driver constructor, used for instance-specific setup.

Note that where *SonyPTZ* appears in any of the above function names, the name is technically arbitrary. This is not the case for the remaining functions which require exact names in order to be executed by the server. In addition to the functions listed above, a PTZ driver also needs to implement the `ProcessPTZRequest` and `ProcessGenericRequest` functions which should call other appropriate functions in response to requests.

1.3.2 Configuration Files

Every *Player* driver has an associated configuration file [25], these typically have the file extension *.cfg*. Configuration files contain data in key / value pairs and at a minimum must contain the name of the driver and what interface(s) the driver supports.

[2] `http://playerstage.sourceforge.net/doc/Player-svn/player/classDriver.html`

There are a number of optional fields available and driver developers can add custom fields if they require persistent storage. *sony-ptz* makes use of this functionality to allow users to store camera calibration data if they choose to. There is an API provided [22] for reading (but not writing) configuration files. The files are simple text files that are space delimited and can contain strings, tuples and other data types as values.

```
# Camera model is detected at runtime, calibration data is saved at the bottom of this file
driver
(
    name "sony-ptz"
    provides ["ptz:0"]
    plugin "./build/sony-ptz.dylib"
    port "/dev/cu.usbserial-FTEIZ7XW"
EVI-D70P [63263 2273 65125 1197 24 23]
)
```

Listing 1: Player configuration file for the *sony-ptz* driver

As can be seen in the above listing; the file contains a driver name, defines a provided interface, sets the value of the optional 'plugin' field and defines two custom fields. The first defines the default serial port to connect to and the other contains calibration data for a camera.

1.3.3 Target Hardware

We explored an overview of the target hardware in a previous section but did not discuss an important aspect. As previously mentioned, all three supported cameras use the same communications protocol, the major differences lie in the minimum and maximum values for pan, tilt and zoom and in the accepted pan and tilt speeds.

The following table contains minimum and maximum values for the three supported cameras, which are reflected in *sony-ptz*. These values were taken from the technical manuals and independently verified using the hardware.

Table 1.3: Hardware limits of the three supported cameras



	EVI-D30	EVI-D70	EVI-D100
Minimum Pan	-100°	-170°	-100°
Maximum Pan	+100°	+170°	+100°
Minimum Tilt	-25°	-30°	-25°
Maximum Tilt	+25°	+90°	+25°
Maximum Pan Speed	1.39 rad/s	1.74 rad/s	5.23 rad/s

As we can see, the *EVI-D70* has the widest range of movement and the *D100* is the fastest.

It is worth noting that despite the apparent progression in model numbers, the *D70* is in fact newer than the *D100*.

1.3.4 Analysis of *sonyevid30*

We have already determined that the existing Sony PTZ camera driver known as *sonyevid30* [9] has a number of issues both with external functionality and internal consistency. It was important that *sony-ptz* rectified these issues with the original codebase.

An important part of the analysis focused on clarifying exactly what the issues were with the current system before beginning the refactoring process.

- Lack of consistency: The function to send a *VISCA* inquiry is called `Send` and the one to send a command is called `SendCommand`, inconsistencies like these occur throughout the driver.
- Significant amounts of 'dead' code: There are a number of unnecessary variables and partially implemented functions (such as `SendStepPan`) which are not used at any point and cause confusion when reading the source code.
- Lack of support for the *D100*: A *D100* is recognised as a *D30* by the `GetCameraType` function and anything specific to the *D100* (e.g. the increased speed) is ignored by the driver.
- Issues with the *D70*: The camera sends a "buffer full" response at times when the buffer is not full, the maximum tilt is set to only 25° despite 195° being supported by the hardware.
- Lack of a sensible naming convention: Variables have names like *numread* and *EPS* and vary from function to function.
- Lack of abstraction: The driver often uses "magic numbers" with no explanation of what they represent. The hexadecimal constant `0x0370` appears in the `GetCameraModel` function, there is no way to know what this value is without consulting the technical manual for the *EVI-D30*. `0x0370` is actually the internal camera representation of a pan position of +100°.

These are the most obvious issues but the evidence is clear that the driver has been built over time in a way that makes it unwieldy.

1.3.5 Server Modifications

The changes that were required to be made to the *Player* server are concentrated in three areas of the source code. The first involves minor changes to *libplayerc*, the core C library which all drivers interact with, either directly or through a wrapper written in a different programming language. There is a function in this library named `playerc_ptz_set_ws` which is used to set the speed of a PTZ camera, this lacks a parameter for zoom speed so one needed to be added.

The next change needed to be in *libplayerc++*, the C++ *Player* interface most drivers use. The function `PtzProxy::SetSpeed` already accepts a zoom speed parameter but this was discarded when the function called the `playerc_ptz_set_ws` function. This call would need to be updated to reflect the previously described change to the C library.

Finally, *libplayerinterface* which defines data structures (implemented as C structs) for interfaces (classes of device) needed to be updated to reflect the other changes, specifically the `player_ptz_data_t` and `player_ptz_cmd_t` structures.

Making these three modifications added full support for zoom speed control to *Player*, allowing new or modified drivers to make use of this functionality.

It is important that changes made were as minimal as possible because this improves the chances of the patch being accepted by the project maintainers and reduces the scope of potential issues. It is worth noting that changes to the data structures described above may have implications on existing drivers and this would need to be accounted for by the *Player* maintainers when considering the inclusion of the patch produced during this project.

1.4 Development Process

With sufficient research and analysis completed, it was almost time to begin the refactoring and development process. The project followed agile design principles in order to be responsive to change and to allow for the quick, decisive implementation of new features.

Some would argue [17] that a driver is not a suitable project for an agile development methodology due to the clearly defined scope and the need to avoid the possibility of hardware damage. In practice, an agile approach worked extremely well when paired with the use of defensive coding practices, risk to the hardware was mitigated by re-using proven methods for interacting with hardware wherever possible.

It is safe to say that camera driver and nuclear power station software are very different things.

Development requires both a process and a selection of tools. Some of these tools (such as *Player* and the use of C or C++) were mandated by the requirements and the existing system, other decisions such as the host operating system and development environment were a matter of preference or technical merit, more detail on the decisions made will be discussed in the next chapter.

1.4.1 Iterative Development Methodology

Any software project of significant size requires a great deal of thought about development methodology. This is particularly important for a project such as this one, which could potentially be used by hundreds or thousands of students, researchers and professionals.

The choice of a development methodology was a key decision and impacted the entire project. It was quickly decided that the traditional waterfall software development method, which was actually first described as an example of an ineffective way to develop software [15], would be too inflexible. As this project is predominantly a rewrite / refactoring of the existing driver it makes sense to evaluate development methodologies which encourage refactoring.

Some well-known open source projects (such as the Linux kernel) do not have a clearly defined development methodology due to their highly distributed development model, but are still hugely successful despite (or even because of) this. With that said, any project aiming to create high-quality software can benefit from a structured approach.

Whilst the project requirements seemed relatively static, details could (and did) change during development. It was decided that development would consist of a number of iterations; accelerated development cycles consisting of the design, implementation / refactoring, testing and integration of a discrete set of functionality. The new system was developed piece by piece taking elements from the original driver where appropriate, refactoring or altering these as required and then testing functionality. In cases where new features were added which are not present in the original driver, a similar approach was taken with design, implementation and testing. This led to a lean and well-tested system. These iterations are described in the *Implementation* chapter.

The project methodology was heavily influenced by the *eXtremeProgramming* (XP) approach to software development, with some modifications to allow for a single-person project. Additionally, research was conducted into the *Scrum* framework and *Kanban* was assessed and used as a visual progress tracking tool. These decisions affected all areas of development and shaped the design, implementation and testing of the software, as will be seen in those chapters.

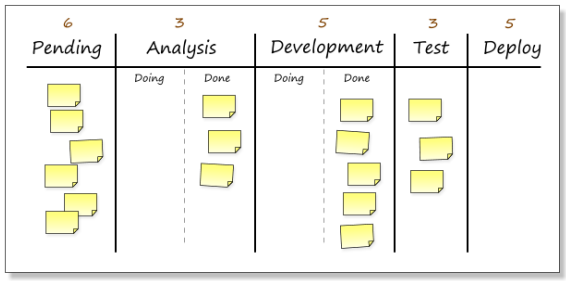


Figure 1.5: A *Kanban* board. [kanbanblog.com]

eXtreme Programming [1] is a well known agile approach, with an emphasis on reactivity, refactoring and pair programming.

The latter is obviously not feasible for a project conducted by a single person, however other focuses of XP such as a focus on testing and regular code-reviews are extremely useful and benefited the process greatly.

One of the hallmarks of XP is the daily 'stand-up' team meeting, which isn't feasible with a team of one. However, like agile development itself, the ideas behind XP are very adaptable.

Time was set aside at the beginning of each day of development to review completed work from the previous day, plan remaining work and assess how the project was progressing overall. This time was also used to record any potential issues or areas to revisit during a scheduled time for review at the end of each iteration.

These daily reviews provided a clear idea of what research, analysis, development, refactoring or testing was required that day. Refactoring and testing were balanced with time spent implementing features to ensure that functionality was considered "done done" [16] and features were completed tested and integrated quickly. There were weekly progress reviews which usually took place during a meeting with the project supervisor.

Scrum was assessed as a possible alternative to the modified XP approach outlined here, but it is important to note that it is not a methodology in the sense that XP is. It is better described as a project framework, although the two overlap when used in a software engineering context.

Player has a code-centric approach to documentation, which meshes well with the focus on "working software over comprehensive documentation" [2] espoused in the *Agile Manifesto*. Documentation in *Player* drivers is embedded within the code itself and the overall framework is already comprehensively documented, so the need to focus on documentation as an end in itself was significantly reduced.

Testing was a crucially important aspect of the development process, a test-driven development approach was considered but was decided against due to the challenges associated with obtaining full test coverage in a system that interacts with hardware.

Regular reflection on completed, current and future work was important in keeping the project on track.

Initial research into methods to track project progress resulted in various approaches being considered, such as keeping a spreadsheet of completed tasks or a regularly updated blog.

Kanban [18] was discovered during this time and seemed much more suitable for tracking what hoped to be a dynamic, fast-paced and responsive process. A *Kanban* board was created with headings including 'Pending', 'Refactoring', 'Development', 'Testing' and 'Deployed', which helped keep development moving at a rapid pace.

Chapter 2

Design

> Design is not just what it looks like and feels like. Design is how it works.
>
> Steve Jobs

This chapter describes the overall design of the modified driver created during this project.

We will begin with a summary of the design objectives and move on to discussing potential design considerations during the refactoring process, providing justifications for why elements were kept the same or changed from the original driver.

After this we will explore an overview of the software, including the chosen development environment, libraries and constants used and the structure of *sony-ptz* and how it compares to *sonyevid30*.

Finally, some insight into the design of key data structures and methods will be provided and data flow within the system will be presented visually and analysed.

2.1 Design Objectives

The design of the modified driver is intended to be clean, easy to understand and similar enough to existing *Player* drivers and its predecessor that it is intuitive to current driver developers. A key focus of the design was a minimalistic approach, taking only what was required from *sonyevid30* and limiting unnecessary layers of abstraction.

The improved system is intended to be as dynamic as possible and generate or query data instead of using stored constants wherever feasible. This provides improved hardware compatibility and makes the source easier to read and understand than *sonyevid30*, which is more static.

2.2 Design Considerations

There were a number of possible design choices which were considered during the refactoring process. These included decisions affecting changes to file layout, data representation and command queuing behaviour.

2.2.1 File Structure

One of the most important design decisions regarded the file structure of the system. There were three potential options, each with their own advantages and disadvantages.

The original driver uses a single file approach. Consideration was given to whether or not to change this structure when refactoring the system. The three possible structures are as follows:

1. **Single file**: All driver code is stored in a single file, the majority of the drivers currently distributed with *Player* follow this design.
2. **Single file with header**: Driver code is stored in a single file with all constants and function prototypes stored in a separate header file.
3. **Multiple files**: A very popular object-oriented design in which each class and any related functionality is stored in its own file, linked together by headers and a main class file.

All three of these are technically valid approaches and have no externally visible impact on the functionality of the driver.

The advantage of a single file approach is that all constants are easily accessible within the file and their definitions are close to where they are used. Another advantage is that the entire system is compact and easy to install. The primary disadvantage of this choice is that, unless suitable documentation and structure are in place, the system can become more difficult to read and understand as it becomes larger, an issue that negatively affects the existing system.

The benefit of using a single file with a separate header is that the file containing the implementation will be cleaner and all constants will be kept together in one place. An issue with this design is that it decouples the value of constants from their usage, resulting in a need to consult a separate file to find the value of a constant.

sonyevid30 has issues with values which do not have an immediately visible meaning, this makes it difficult to read and maintain. There was a possibility that using this design would exacerbate these issues.

With many object-oriented designs, the obvious choice is to logically separate the various classes and objects into different files, e.g. a file with a class to represent a PTZ camera, one to represent a *VISCA* packet and so on and so forth. This is the approach used by *libVISCA*.

However, it is important to understand that the core of *Player* is written in C which has no concept of objects. This is not a standalone software product, there are conventions which *Player* drivers are expected to adhere to by users and developers.

No driver currently included with *Player* uses this approach, perhaps because it is better suited to applications than it is to ”firmware”. Essentially, *Player* drivers are object-oriented islands in a sea of imperative frameworks, which is reflected in their design.

sony-ptz was kept as a single-file, single class system the same as its predecessor, with a focus on ensuring a logically and clearly defined set of methods, data structures and utility functions.

The advantage of this is that all constants and method signatures are clearly visible and any changes made will immediately be propagated to the entire system. It is also possible that there is a performance advantage to operating on a single object rather than several, although this has not been verified.

With this design, the driver remains very compact and can be distributed in the way developers working on the *Player* server and drivers have come to expect.

2.2.2 Dynamic vs. Stored Constants

The unmodified driver stores a number of constants for properties such as the minimum and maximum pan and tilt limits.
A huge disadvantage of this approach is that these numbers are often inaccurate and do not properly reflect the capabilities of the hardware. An obvious example of this is the artificial limit on the tilt of the *D70* to +25° rather than the correct value of +90°.

There are some advantages to using stored constants. Appropriate use of stored constants can improve readability and make testing easier by giving developers easier access to expected values. Sensible use of stored constants can also reduce software complexity by reducing the scope of a calibration function like the one included in *sony-ptz*.

As discussed in the overview of the project objectives in the previous chapter, it was important to reduce the number of stored constants wherever possible to create a more dynamic driver. Dynamically generated values were used in all possible cases, the driver queries the minimum and maximum pan and tilt values and speeds whenever it is connected to a new camera rather than saving these as static constants.

Certain values remain static, either because there is no method to generate them dynamically or because the values do not change between cameras. An example of the former is the list of valid pan and tilt speeds. These are linear on the *D30* but not linear on either of the other two models so cannot be generated as a mathematical series. Without introducing a complex calibration process for speeds, it seemed more sensible to continue storing these values (obtained from the official technical manuals) within the source code.

An example of values which do not change between cameras are the definitions for the various *VISCA* packets. It can be argued that these do not count as "magic numbers" because they are only used once within the system, their purpose is clearly defined in the technical manuals and it is easier for a developer to read the packet directly rather than piece together a series of constants. "`0x81, 0x09, 0x04, 0x00`" is easier to search for in the manual than finding and searching for the values of "`ADDRESS_CAMERA_1, VISCA_INQUIRY, VISCA_INQUIRY_MISC, VISCA_INQUIRY_POWER`".

With that said, it would not be difficult to change the modified driver to use named constants like these if desired.

2.2.3 Buffered vs. Immediate Operation

A key difference between *sony-ptz* and *sonyevid30* is the way buffers are used. The former uses the two command buffer discussed in the previous chapter, however the *D70* in particular has issues with this and occasionally sends back a "Command Not Executable" message when receiving commands, even if the buffer is not full.

sony-ptz is intended for real-time operation, which means that when a command is received it will immediately be executed regardless of the status of any other commands. This solves the issue with the buffer being full because the buffer is ignored. For pan and tilt movement commands there is an option to wait until execution completes before processing the next command, which is useful for programming movement sequences.

Some *sonyevid30* users may require a buffer for certain applications, this is one of the reasons (as well as the systematic renaming of functions and variables) the improved driver created in this project is presented as an alternative for the existing driver, rather than a drop-in replacement.

2.3 Design Overview

Before detailing specific aspects of the modified design we will look at a general overview of the system and the development environment used to create it.

2.3.1 Development Environment and Tools

The development system was a Mac running OS X 10.10. *Player* development focuses on Linux but because OS X is a fully UNIX-compliant operating system, all of the tools required for development were either included (e.g. *svn*) or could easily be obtained.

The build system used was *CMake*, this was selected because it is recommended in the official *Player* documentation [12] for cross-platform build automation. The machine used for both project demonstrations was running Linux, so the driver itself is platform agnostic.

Player is written in a mixture of C and C++, client libraries exist for Java, Python and other languages, but all of the drivers included with the project are written in C++ and the core is written in C. The deciding factor is the fact the existing codebase is entirely in C++ and there is no compelling reason to rewrite it in another language in order for refactoring and extension to proceed.

An issue occurred when attempting to compile *Player* (an error concerning use of the `extern` keyword) which was solved by installing `g++-4.9`. OS X claims to have an installation of *gcc* but this is actually just a symbolic-link to the *clang* compiler. The latest version of *Player* (*3.1.0 svn*) does not compile correctly using *clang*, so *gcc* was required.

CMake is designed for cross-platform use and modules for building drivers are included with *Player*. All compilation options are included in a `CMakeLists.txt` file in the root directory of the project. This made compiling the driver for both OS X and Linux very simple. The same settings can be used for any supported operating system, as long as *CMake* is executed before compiling.

```
# CMake configuration for building the sony-ptz driver, requires CMake >= 2.7
project (Sony-PTZ)
cmake_minimum_required (VERSION 2.7 FATAL_ERROR)

SET (CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR}/cmake/modules)
SET (CMAKE_SHARED_LIBRARY_PREFIX "") # Do not build driver with the standard 'lib' prefix

# SET(CMAKE_BUILD_TYPE Debug) # Include symbol table for debugging

INCLUDE (UsePlayerPlugin)

PLAYER_ADD_PLUGIN_DRIVER (sony-ptz SOURCES sony-ptz.cc)
```

Listing 2: `CMakeLists.txt` used to compile the *sony-ptz* driver

Once the compiler issue had been resolved, no further problems were encountered with the development environment. The specific editor and other tools are down to personal preference, *MacVim* with C++ semantic completion was used for all the editing in this project, but a suitably configured IDE or other editor could also have been be used.

Version control was used throughout this project, with all resources being hosted in a central *git* repository (`http://www.github.com/OMCS/sony-ptz-player`). *Git* was chosen because it is an easy to use, powerful VCS with free private repositories available on *GitHub* for students. Commits were pushed every time a significant amount of code was added or modified, which allowed for greater flexibility when working on multiple computers. As Player itself is open source, *git* provides an excellent platform to allow for other developers to submit modifications.

The open-source documentation framework *Doxygen* [21] was used to generate a number of the diagrams used in this dissertation

In addition to the software previously listed, two tools were used to perform static analysis on code and will be discussed further in the *Testing* chapter.

2.3.2 Dependencies

The modified driver was intended to be self-contained and have as few external dependencies as possible, although certain software and libraries are required. For compilation; an installation of *CMake* with the *Player* modules is necessary, as well as a recent version of the *gcc* compiler, which is also used to compile *Player* itself. It is possible that compiling the driver with *clang* also works but this is not officially supported.

In terms of libraries, the driver uses two standard C++ libraries for stream manipulation, five standard C libraries present on UNIX systems for serial communications, multi-threading and mathematical calculations and the *Player* driver API provided by *libplayercore*. This is an improvement on *sonyevid30* which has thirteen dependencies. For portability, no non-standard libraries except for `libplayercore` and `unistd.h` (standard on all UNIX systems) are required.

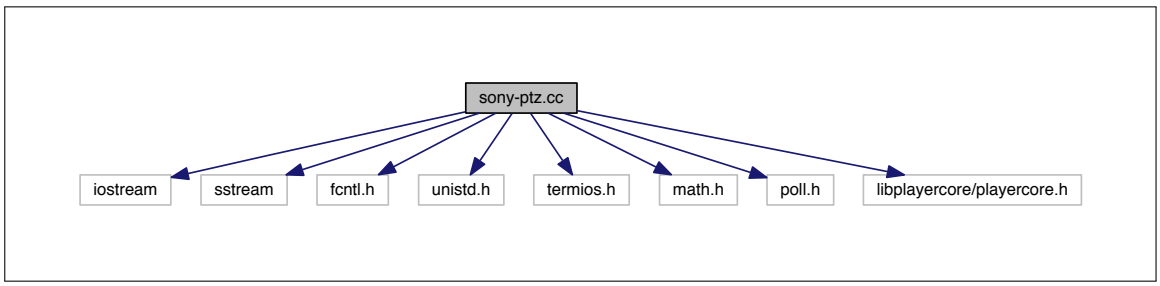


Figure 2.1: Dependency diagram of libraries required by *sony-ptz*. [Generated by *Doxygen*]

Table 2.1: Libraries required by *sony-ptz*

Library	Type	Used for
`iostream`	C++ (Standard)	Error logging
`sstream`		String manipulation
`unistd.h`	C (Standard on UNIX systems)	File closing and sleep functions
`fcntl.h`		Serial port file descriptor I/O and flags
`termios.h`		Serial communications
`poll.h`		Receiving hardware data
`math.h`		Conversion between degrees and camera units
`playercore.h`	C (Requires *Player* headers)	Access to the *Player* API

2.3.3 Constants

Just as in *sonyevid30*, there are a number of constants within the modified implementation, these may be set at runtime or within the source code but are never changed once set. A table of constants and their purpose is shown below. *Nb*: `[DXX]` is used as a placeholder for a given model name such as *D30*.

Table 2.2: Constants defined within *sony-ptz*

Name	Type	Usage
`[DXX]_MODEL_HEX`	*int (hex)*	Hexadecimal representation of camera model number. Used for camera detection during calibration.

`[DXX]_MIN_PAN_DEGREES`
`[DXX]_MAX_PAN_DEGREES`
`[DXX]_MIN_TILT_DEGREES`
`[DXX]_MAX_TILT_DEGREES`
	float	Self-explanatory, dynamically set during calibration.

`[DXX]_MIN_ZOOM_FOV`
`[DXX]_MAX_ZOOM_FOV`
`[DXX]_MIN_ZOOM_SPEED`
	float	The values for minimum and maximum zoom and minimum zoom speed depend on the camera and lens used and are static.

`MAX_ZOOM_SPEED`	*float*	All supported cameras share the same maximum zoom speed.
`[DXX]_PAN_SPEEDS` `[DXX]_TILT_SPEEDS`	*float[]*	All supported pan and tilt speeds in *radians/sec*. Supported values are listed in the technical manuals.
`NUM_SUPPORTED_MODELS`	*int*	Contains the number of supported camera models, acts as an iterator during calibration.
`EVI_[DXX]`	*ptz_cam_t*	*struct* representing an *EVID* camera model, one for each supported model.
`SONY_PTZ_CAMERAS`	*ptz_cam_t[]*	Array containing all *ptz_cam_t* structs, the correct *struct* will be selected from these during calibration.

The key difference between this design and the way *sonyevid30* implements constants is that the majority of these values are auto-generated during calibration, rather than being static.

2.3.4 Class Structure

The structure of the system is presented in terms of a UML class diagram in **Figure 2.2** on the next page.

As we can see, *sony-ptz* inherits from the *ThreadedDriver* class described in the previous chapter.

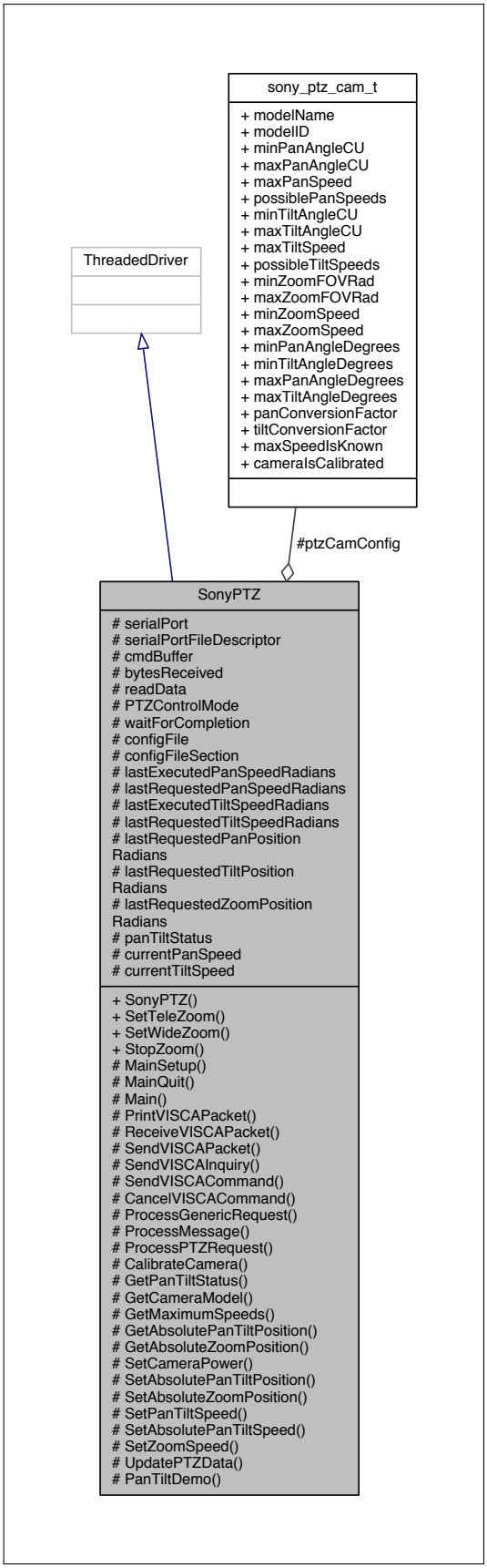


Figure 2.2: Class diagram describing the *sony-ptz* driver. [Doxygen]

The driver class *SonyPTZ* provides concrete implementations for the virtual *Main*, *MainSetup* and *MainQuit* methods found within its parent class.

There are also a number of other methods for hardware interaction and message processing, many of these use extremely similar code to the implementation in *sonyevid30* and full credit should be given to the original developers. This allowed the technical work to focus on fixing bugs, implementing new functionality and documenting and refactoring the existing codebase.

Several utility functions are not represented here as they are external to the class, these include functions used for conversion between radians and internal camera units (taken verbatim from *sonyevid30* with a minor bug fix contributed by the project supervisor) and the mandatory driver registration function common to all *Player* drivers.

Note that (just as in *sonyevid30*) a *struct* of type *sony_ptz_cam_t* is used within the main class in an object named *ptzCamConfig*, this defines and stores all the parameters for an *EVI* camera and is discussed in more detail later in this chapter.

Many of the member variables and methods within the class are self-explanatory and it is unnecessary to list them all. However, some are crucial to the design of the system and warrant further discussion.

Message handling is implemented within the `ProcessMessage` method, which calls `ProcessPTZRequest`, which itself calls the appropriate hardware method depending on the type of message (speed / position / misc).

Most methods in the driver will be accessed by clients via a C++ *PtzProxy* and therefore are not publicly accessible. The exceptions to this are the camera zoom methods, which are used to control zoom in unmodified versions of *Player* which do not support zoom speed control.

CalibrateCamera is one of the most important methods and sets values for a number of constants (listed in the previous section) and variables depending on the connected hardware. It generates and lists calibration data which may be added to the configuration file for future re-use, so that the calibration process only needs to be completed once for each model of camera.

VISCA communications are implemented using the `SendVISCAInquiry`, `SendVISCACommand` and `ReceiveVISCAPacket` methods. The `SendVISCAInquiry` and `SendVISCACommand` methods construct a packet and then class `SendVISCAPacket` which is a lower level method that adds a header and terminator to the packet and sends it over the serial connection established in `MainSetup`.

`ReceiveVISCAPacket` retrieves the response to a *VISCA* command (typically an acknowledgement) or inquiry (typically an information return, i.e. data), which is crucial as it allows the camera to communicate back to the controller for two-way communication.

`GetPanTiltStatus` is used during calibration and returns the status of a pan or tilt command, typically indicating if it has finished or hit a hardware limit - more information on this can be found in the next chapter when we explore how calibration is implemented. This method updates the value of the *panTiltStatus* instance variable.

There are a number of methods which return various position or speed information from the camera, `GetCameraModel` is used for calibration and assists with populating a *sony_ptz_cam_t struct* with the correct parameters for a given model of camera, depending on its hexadecimal model identifier. The remainder of the data within the *struct* is generated or read in during the calibration process.

2.4 Data Structure Design

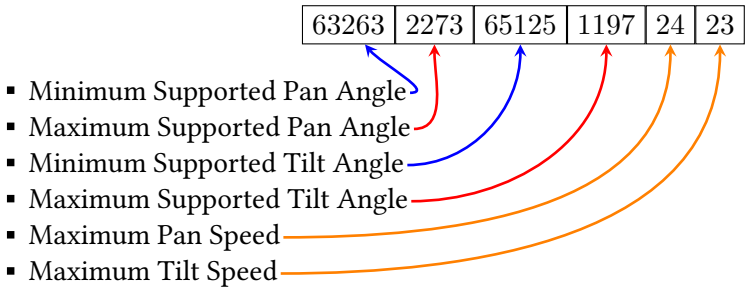
2.4.1 Representing an *EVI* Camera: The *ptz_cam_t* Structure

Similar to the unmodified driver, the central data structure used by the system is a *struct* of type *ptz_cam_t*, containing fields which represent all possible parameters for an *EVI* camera.

These fields store hardware limits, available speeds, model identifiers, calibration flags and other information used to uniquely identify a connected camera. An explanation of each field is included in **Table 2.3** on the next page.

This data structure was designed so that data for fields supporting calibration could be read directly from the driver configuration file. The other values are set during the `MainSetup` method.

We looked at an example configuration file in the previous chapter. One of the custom fields present in the file had a key of "EVI-D70P", the values after this are calibration data which takes the following form:



Calibration data is comprised of decimal values which represent hardware limits in camera units. These provide values for six of the fields in the *ptz_cam_t* structure, the remaining values are calculated or queried at runtime.

An attractive property of this design is that there is no need to store the corresponding minimum and maximum values in degrees. As each camera has clearly defined ranges for pan and tilt, it is possible to perform the conversion mathematically when required.

Table 2.3: Fields comprising the *sony_ptz_cam_t* data structure

Field	Type	Description
`modelName`	*char[]*	String representation of a camera model name, e.g. "EVI-D30".
`modelID`	*int (hex)*	Hexadecimal model identifier reported by the camera.
`minPanAngle[Unit]` `maxPanAngle[Unit]`	*int (hex) / double*	Hexadecimal integer for internal camera units, double for degrees. Set during calibration.
`maxPanSpeed`	*int (hex)*	Maximum pan speed in internal camera units, set during calibration.
`possiblePanSpeeds`	*const float**	Array containing all valid pan speeds in degrees per second.
`minTiltAngle[Unit]` `maxTiltAngle[Unit]`	*int (hex) / double*	Hexadecimal integer for internal camera units, double for degrees. Set during calibration.
`maxTiltSpeed`	*int (hex)*	Maximum tilt speed in internal camera units.
`possibleTiltSpeeds`	*const float**	Array containing all valid tilt speeds in degrees per second.
`minZoomFOVRad` `maxZoomFOVRad`	*double*	Field of view in radians, varies depending on the camera and lens used, set during calibration.
`minZoomSpeed` `maxZoomSpeed`	*int (hex)*	Measured in camera units, value varies between cameras, set during calibration.
`panConversionFactor`	*double*	Relates the pan angle in camera units to the pan angle in degrees. (*maxPanCU* / *maxPanDegrees*)
`tiltConversionFactor`	*double*	Relates the tilt angle in camera units to the tilt angle in degrees. (*maxTiltCU* / *maxTiltDegrees*)
`maxSpeedIsKnown`	*bool*	Indicates whether maximum speed is known for thresholding invalid speed values. Not used in the final version of the driver.
`cameraIsCalibrated`	*bool*	Flag which indicates if camera has been calibrated, used to decide whether to calibrate a camera or read values from a previous calibration.

2.4.2 External Variables and Functions

As previously mentioned, there are a number of external functions and variables within the system.

These were defined outside the *SonyPTZ* class either because they needed to be, such as the driver registration functions which must be defined outside the driver class, or because the functionality did not require or depend on anything within the class.

Most of these functions are conversion functions used to convert between radians and internal camera units and as previously mentioned, are taken verbatim from the original codebase with minor changes contributed by Fred Labrosse. The reason for this is that these calculations have been proven to work and using existing and proven implementations is preferable to "reinventing the wheel". These conversion functions are crucial for ensuring the output from the camera is understandable, e.g. "Tilt is at 45°" rather than "Tilt is at 0x067F" and for allowing input in radians or degrees which is then translated by the driver into a camera-specific format.

2.5 Data Flow

Data flow throughout the system is dependent on external inputs (e.g. subscribers) but a simplified version (which ignores the complexities of multi-threading) is presented in **Figure 2.3** on the next page.

This diagram ignores the calibration process (which will be described in the next chapter) and is highly simplified - there are more method calls within the system than those pictured.

Despite the simplification, the diagram provides an acceptable overview of how data moves through the system. A summary of the diagram is as follows:

1. The driver initialises, registers with the *Player* server and acquires a reference to the configuration file.
2. A call to the *SonyPTZ* constructor creates an object to control a given camera.
3. The serial port and other resources are prepared and the main messaging processing loop is entered.
4. The hardware is regularly polled to determine if there is new data or not
5. The `UpdatePTZData` method is called, which calls the appropriate method to send a packet, which in turn calls the method which receives the response.

At this point the driver will continue the message processing loop if there are still subscribers connected, otherwise it will call the function to perform cleanup tasks and then terminate.

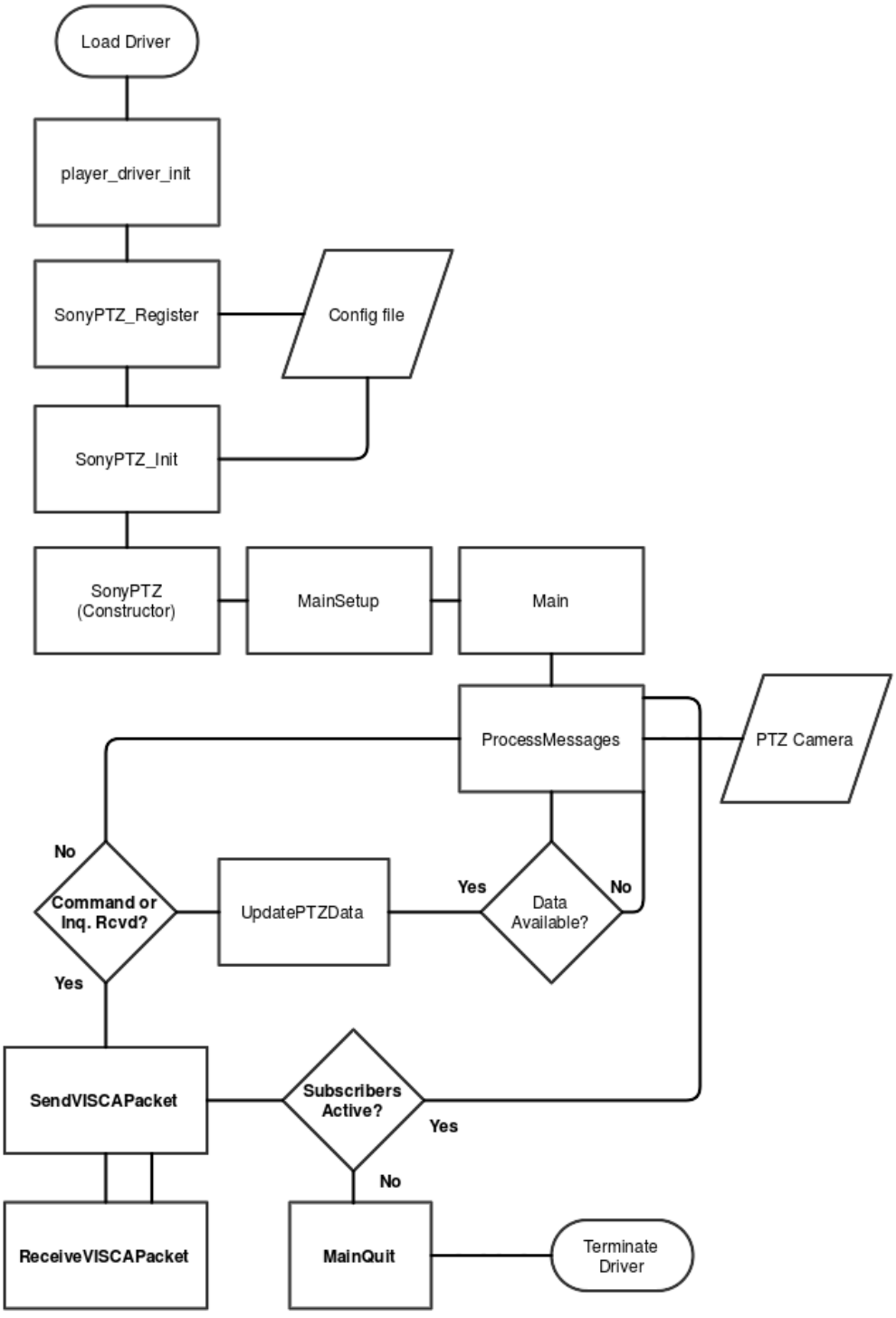


Figure 2.3: Diagram illustrating basic operation of the driver, including data flow.

Chapter 3

Refactoring & Implementation

> C makes it easy to shoot yourself in the foot;
> C++ makes it harder, but when you do it
> blows your whole leg off.
>
> Bjarne Stroustrup

Within the previous two chapters, we have examined the background and the design of the modified system created during this project. In this chapter, we will explore the implementation details related to this refactored system.

While there are different definitions of refactoring, the approach taken by this project was to start with a blank slate and add to the piece together the driver iteratively using existing driver code, modifying or adding to it as required and then testing. This is as opposed to starting with the existing driver and then refactoring the code directly. Great effort was taken to fully understand the system, at no point was code used verbatim without a thorough understanding of its function and how (and why) to modify it if required.

This approach resulted in a lean system which took only what was required from *sonyevid30* and allowed regression testing to be carried out with ease. This chapter chronicles the evolution of the software from a simple proof of concept using only the most basic code from *sonyevid30*, to a fully-fledged alternative to the existing driver with a cleaner design and several new features.

3.1 Iterative Development

When using an iterative development methodology, it is important that each iteration has a clear aim and is long enough to accomplish the primary objectives, yet short enough to react to changes in the software requirements.

There was a significant amount of preliminary work required, including setting up the development environment and testing basic hardware communications. Adding this functionality to the driver, refactoring as required and testing comprised the first iteration.

When identifying additional iterations, it was decided that most sensible approach was for these to coincide with project milestones.

The assessment of this project included two demonstrations, one mid-way through the project and one as a final demonstration. These provided an ideal target for the end point of two iterations.

Therefore, the development of *sony-ptz* consisted of three stages; the initial iteration, the mid-project iteration and the final iteration.

3.2 Initial Iteration

The first iteration began as soon as the background research and analysis was complete. *Player*, *CMake* and the other assorted software listed in the previous chapter was installed and configured.

3.2.1 Driver Development Fundamentals

The first steps included initialising the *git* repository for the project source code, creating an appropriate *CMake* configuration file, installing the latest version of *Player* and conducting some basic sanity checks such as compiling a sample driver. This was done to ensure that the development environment was correctly configured.

Next, some testing with the *sonyevid30* driver was carried out to ensure the camera hardware was functioning correctly. The tool used to do this was *playerv*, a utility included with *Player* for data visualisation which also supports interactive hardware control.

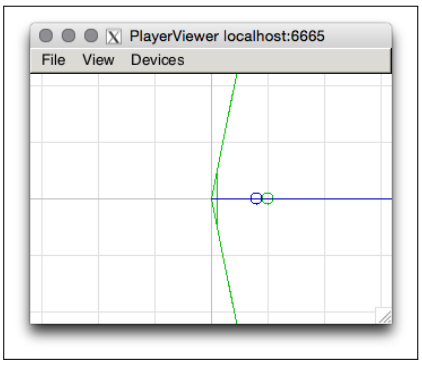


Figure 3.1: The *playerv* utility displaying live camera data.

During this testing, it became clear that the combination of the *EVI-D70*, the *sonyevid30* driver and the interactive functionality of *playerv* caused issues. After less than thirty seconds receiving commands, the driver locked up and required the *Player* server to be restarted.

The packet received before this happened was "$0x90$, $0x62$, $0x41$", which is defined in the *EVI-D70P* technical manual as "Command Not Executable". This was unexpected given that there is a separate packet representing a "Command Buffer Full' message.

The erroneous sending of this packet by the *D70* was identified as an issue with the unmodified system, but there was no mention of a different error packet being received. This use case would be revisited in the testing phase of the modified driver to ensure that it had been solved during the refactoring process.

At this point, it was time to begin development. Three objectives were identified for this iteration; the first was to produce a driver which successfully compiled and loaded, the second was to implement basic serial communications and the third required implementing functionality to send and receive *VISCA* packets. The completion of these tasks would provide a solid foundation for further development.

The first step was to create a 'dummy' driver which compiled but did not yet have any functionality. The *Player* project provides a guide [14] which explains how to produce a simple driver, this provided a useful source of information.

A new source file was created with an empty class definition inheriting from *ThreadedDriver* and method stubs with the minimum required functionality were added for the `MainSetup`, `Main` and `MainQuit` methods, the default constructor, initialisation and registration functions and the factory function for instantiating new driver instances.

Compilation using *CMake* was simple and the driver compiled successfully using *gcc*. This produced a shared object, the format for *Player* plugin drivers. In addition to the correct method and function stubs, *Player* requires a configuration file to load a driver. The layout of a configuration file was described in the previous chapter, recall that this file tells *Player* the name of the driver, the interface(s) supported (*ptz* in this case) by the driver and any other relevant directives.

After creating this file, *Player* refused to load the driver because it could not locate it. This issue was fixed by adding "`plugin ./build/sony-ptz.dylib`" to the configuration file. On Linux systems this would be changed to "`./build/sony-ptz.so`", due to the difference in shared object (library) formats between the two operating systems.

After this change the server correctly loaded the driver, however a print statement that had been added to the start of the *Main* method did not execute. This is because the *Player* server waits for client software to subscribe to a device before executing any driver code.

Because this was impractical during development, the optional configuration flag "`alwayson 1`" was added to the configuration file. This changes the behaviour so that the driver code will be executed as soon as it is loaded, greatly simplifying debugging and testing.

It is important to note that at this point, the driver did not communicate with the *EVI* camera in any way, none of the functionality for this had been implemented yet. However, the driver successfully compiled and executed, meeting the first objective of this iteration.

Before continuing, the functions currently implemented were fully documented and the current work was assessed to see if refactoring was required. As the amount of work done so far was fairly small, no significant refactoring was necessary and it was possible to move on to the next objective with only minor modifications.

3.2.2 Implementing Serial Communications

Before implementing packet-based communication, it was necessary to implement the serial interface for the physical connection between the camera and the controller. The code from *sonyevid30* worked perfectly for this and was used with minor structural changes, including renaming variables and functions to be clear and consistent.

The C standard library *termios* is used for configuring a serial port. Within UNIX-like systems, every piece of hardware is represented as a file, including serial ports. The *fcntl* library is used for setting file descriptor flags (e.g. *O_RDWR*) and the *unistd* library is used for reading from and writing to the port, and therefore the connected camera.

Serial port file descriptors are typically referenced as either `/dev/ttyS*` for a hardware serial port or `/dev/ttyUSB*` if using a USB adapter. The existing and modified drivers both read the desired port from their respective configuration files.

The steps to correctly open a serial port for reading and writing are summarised below, illustrated with some code taken from the existing system, refactored, added and tested during this iteration.

- Initialise any required variables.
- Open the serial port file descriptor (e.g. `/dev/ttyUSB`) using the standard `open` function with the required flags in **non-blocking** mode.

```
/* Try to read the serial port connected to the PTZ camera,
    * If this doesn't work then return -1 to indicate an error
    * O_RDWR means open in read / write mode, O_SYNC means writing will be done
↪   synchronously
    * O_NONBLOCK returns an error if no data is available rather than blocking,
    * We use this because a camera may not actually be connected
    * S_IRUSR and S_IWUSR are file permission bits allowing read and write access
↪   permissions
    */
   if ( ( serialPortFileDescriptor = open(this->serialPort.c_str(),
          O_RDWR | O_SYNC | O_NONBLOCK, S_IRUSR | S_IWUSR) )  < 0)
   {
       perror("Failed to open serial port");
       return -1;
   }
```

- Flush the serial port to prepare it for use.
- Initialise a *termios struct* for the serial port configuration.
- Set the members of the *struct* with the correct baud rate and mode using the relevant *cf** functions from *termios*.
- Set the attributes of the serial port using the contents of the *struct*.

```
/* If the serial port has been successfully flushed and attributes can be read,
   set up port for communicating with PTZ camera */

   /* Enable 'raw' mode as this is a serial port and not an actual tty / terminal */
   cfmakeraw(&serialConf);

   /* Set speed to 9600 baud for input and output, 9600 is the VISCA baud rate */
   cfsetspeed(&serialConf, B9600);

   /* Attempt to configure the serial port with the new settings, 'raw' and 9600 baud */
   if (tcsetattr(serialPortFileDescriptor, TCSAFLUSH, &serialConf) < 0)
   ...
```

- Attempt to read data from the serial port to determine if the link has been established correctly.
- Reopen the serial port in blocking mode if the link was successfully established.

```
   /* Finally, attempt to reopen the port in blocking mode,
    * now we are sure it is connected to the camera we want to communicate with
    * The third argument  to fcntl() defines what operation is performed on the flag
↪   bitmask
    * In this case XOR which removes the O_NONBLOCK flag.
    */
   if (fcntl(serialPortFileDescriptor, F_SETFL, serialFlags ^ O_NONBLOCK) < 0)
   {
       perror("Could not set serial port flags in SonyPTZ::MainSetup(), fcntl()
        ↪   failed");
       close(serialPortFileDescriptor);
       serialPortFileDescriptor = -1;
       return -1;
   }
```

This implementation enables data to be sent to and received from the serial port using the standard `read()` and `write()` C functions.

Notice that each library call is placed within a conditional statement, both versions of the driver include significant amounts of error detection and validation, which ensures the hardware is always in a clearly-defined state.

Although *sony-ptz* is effectively written in C++, some functionality such as serial I/O is implemented using C libraries, as it is in *sonyevid30*. As C++ provides access to a lot of C functionality, combining the two typically compiles and works as expected. However, care must be taken with C functions to avoid linker issues caused by "name-mangling" performed by the C++ compiler.

The reason for the use of both languages is that C is preferable to C++ when dealing with byte-level hardware access [23], as it can provide better performance and a higher degree of control. *Player* drivers typically combine C and C++ where it is appropriate to do so; although it is important to take a disciplined approach, being consistent with which language is used when, avoiding undefined behaviour and not combining the two within a single statement or logical unit.

3.2.3 Sending and Receiving *VISCA* Packets

It was now possible to read data from and send data to the camera via the serial port, however data sent or received is meaningless without the correct protocol.

We recall from the previous chapters that *VISCA* is a simple packet-based serial protocol, the driver needed to implement this in order to successfully send commands and inquiries to the camera and receive responses.

At this point in the iteration, creating a new data structure to represent a *VISCA* packet was considered, this would have included fields such as *packetTargetAddress*, *packetTerminator* and *packetMessage*. This was removed later in the process because it added an additional layer of abstraction to what is essentially a number of hexadecimal values in a specific order.

The driver provides abstraction for its users by containing these values within methods with sensible names like `GetPanTiltPosition` or `SetCameraPower`. For developers extending the driver, it is arguably easier to work directly with the underlying data rather than setting and reading fields from a structure each time a packet needs to be constructed or read.

The first step was to implement a method to send a *VISCA* packet, this is effectively as simple as writing specific values to the serial port. As discussed in the first chapter, *VISCA* expects a serial port configured with what are typically referred to as *8-N-1* parameters. Eight data bits, no parity bit and one stop bit.

The implementation described in the previous section configures the port accordingly, so the next step was to break down the task of sending a packet into its constituent parts and implement those, the existing code to do this was taken from *sonyevid30* and modified where issues were discovered.

Commands and Inquires are distinct in *VISCA*, so two new methods were created named `SendVISCACommand` and `SendVISCAInquiry`, these are analogous to the `Send` and `SendCommand` functions in *sonyevid30*.

The parameters for the method to send a command include the command values to send (defined as an *unsigned char **), the length of the command in bytes and an optional parameter to specify which camera (1-7) the command will be sent to. This is partially implemented in the unmodified driver and it was decided that the functionality would remain in *sony-ptz*.

After receiving a command, an acknowledgement (*ACK*) is sent back by the camera. If the *ACK* is of the wrong length (the specification defines an *ACK* as 3 bytes) or the wrong format, this needs to be reported to the driver.

Commands generate two responses from the camera, the *ACK* to confirm a command has been received and a completion packet which is sent when the command has been executed.

These messages differ by a single byte, the second byte of the response is $0x41$ for an *ACK* and $0x51$ for a completion.

For reference, a partial listing of the SendVISCACommand method (heavily based on the existing implementation) is provided on the next page.

```
/* Method to send a VISCA command */
int SonyPTZ::SendVISCACommand(unsigned char* cmdStr, int cmdLen, uint8_t camID)
{
    /* Variables to store the command reply and its length */
    unsigned char cmdReply[MAX_PTZ_PACKET_LENGTH];
    int cmdReplyLen;

    /* This uses a lower level function to send the actual serial packet */
    if ((cmdReplyLen = SendVISCAPacket(cmdStr, cmdLen, cmdReply, camID)) <= 0)
    {
        return cmdReplyLen;
    }

    /* Wait for the ACK which signals the command was received */
    while ( (cmdReply[0] != 0x90) || ( (cmdReply[1] >> 4) != 0x04 ) || (cmdReplyLen != 3) )
    /* ACK messages are 3 bytes long and for camera 1 should begin with 0x90 */
    {
        if ( (cmdReply[0] != 0x90) || ( (cmdReply[1] >> 4) != 0x05))
        {
            /* Print whatever was received instead of an ACK for debugging purposes */
            PrintVISCAPacket("SonyPTZ::SendVISCACommand(): Expected ACK, but received",
            cmdReply, cmdReplyLen);
        }

        /* Receive the reply, if the return value is 0 or less then return the length */
        if ((cmdReplyLen = ReceiveVISCAPacket(cmdReply)) <= 0)
        {
            return cmdReplyLen;
        }
    }

    ...

    return 0;
}
```

Listing 3: Implementation for sending a *VISCA* command

The implementation listed above includes calls to two new methods which have not yet been explained, `SendVISCAPacket` and `PrintVISCAPacket`. We will discuss these later in this chapter.

It is also worth noting that the above listing does not include functionality to control command queuing behaviour. In the full implementation there is a call to cancel existing commands unless a boolean value is set within the driver which indicates that this command is part of a sequence and to wait for each command to complete before executing the next one, rather than executing commands the moment they come in and discarding the command in progress.

This functionality was added to allow for movement sequences, is not present in *sonyevid30* and will be discussed later in this chapter.

Before we examine the `SendVISCAPacket` method, let us identify how `SendVISCAInquiry` differs from `SendVISCACommand`.

```
1  /* Function that uses SendVISCAPacket() and ReceiveVISCAPacket() to send an inquiry to a
   ↪  camera */
2  int SonyPTZ::SendVISCAInquiry(unsigned char* inqStr, int inqLen, unsigned char* inqReply,
   ↪  uint8_t camID)
3  {
4      int inqReplyLen;
5
6      if ((inqReplyLen = SendVISCAPacket(inqStr, inqLen, inqReply, camID)) <= 0)
7      {
8          return inqReplyLen;
9      }
10
11     while ((inqReply[0] != 0x90) || (inqReply[1] != 0x50))
12     {
13         /* Inquiry replies should always be 3 bytes */
14         if ((inqReply[0] != 0x90) || ((inqReply[1] >> 4) != 0x05) || (inqReplyLen != 3))
15         {
16             PrintVISCAPacket("SonyPTZ::SendVISCAInquiry(): expected information return but
               ↪  received",
17             inqReply, inqReplyLen);
18         }
19
20         /* Call receive function for reply */
21         if ((inqReplyLen = ReceiveVISCAPacket(inqReply)) <= 0)
22         {
23             return inqReplyLen;
24         }
25     }
26
27     return inqReplyLen;
28 }
```

Listing 4: Implementation for sending a *VISCA* inquiry

Note that this method has an additional parameter for the variable used to store the inquiry reply. Inquiries return information rather than just an acknowledgement. This method uses the same `SendVISCAPacket` method that is used when sending commands. The unmodified driver provides the `Send` method which is very similar.

Instead of waiting for an *ACK*, this method waits for an inquiry packet to be returned. These always have the first two bytes set to $0x90$, $0x50$. This packet should always be 3 bytes long.

At this point the method calls the `ReceiveVISCAPacket` method which is used to read and parse the information return packet.

`SendVISCACommand` and `SendVISCAInquiry` are higher-level methods which do not perform the sending or receiving of packets directly, instead they call other methods which conduct the actual communication.

The first of these is `SendVISCAPacket`. This method constructs a packet using the command or inquiry that is passed in and sends it to the camera via the serial port.

```
1  /* Low level function to send a packet via a serial connection
2   * Constructs the packet using the header, command or inquiry passed in and the terminator
3   * Then attempts to send it to the camera
4   */
5  int SonyPTZ::SendVISCAPacket(unsigned char* msgStr, int msgLen, unsigned char* msgReply,
    ↪  uint8_t camID)
6  {
7      unsigned char packetToSend[MAX_PTZ_PACKET_LENGTH];
8      int currentByte; // VISCA packets may be from 3 to 16 bytes in size
9
10     if (msgLen > MAX_PTZ_MESSAGE_LENGTH)
11     {
12         std::cerr << "SonyPTZ::SendVISCAPacket(): Message is too large at "
13                   << msgLen << "bytes in size" << std::endl;
14         return -1;
15     }
16
17     /* Address of the controller will always be 0, first camera will be 1, etc. */
18     packetToSend[0] = 0x80 | camID;
19
20     for (currentByte = 0; currentByte < msgLen; currentByte++)
21     {
22         packetToSend[currentByte + 1] = msgStr[currentByte];
23     }
24
25     packetToSend[currentByte + 1] = 0xFF; // VISCA packet terminator
26
27     /* Packet has been constructed, now we attempt to send it via the serial port */
28     if (write(serialPortFileDescriptor, packetToSend, currentByte + 2) < 0)
29     {
30         perror("Error sending VISCA packet in SonyPTZ::SendVISCAPacket()");
31         return -1;
32     }
33
34     return (ReceiveVISCAPacket(msgReply));
35 }
```

Listing 5: Implementation for constructing and sending a *VISCA* packet

This method accepts a *msgStr* parameter, which contains the hexadecimal values specified in the command or inquiry. It then prepends this with the target address and appends the terminator ($0xFF$) to the end to form a complete packet.

Initially, values relating to the length of various packets were hard-coded, but during the refactoring process for this method it became clear that these should be changed to constants as in the original driver, as they are used multiple times.

Due to the agile nature of this project, this was a simple change to make and demonstrated an evolution beyond the design planned earlier in the iteration. These constants were added as C++ preprocessor definitions and are listed in the **Table 3.1** below.

Table 3.1: *VISCA* Packet Constants

Name	Value
`MAX_PTZ_PACKET_LENGTH`	16
`MAX_PTZ_MESSAGE_LENGTH`	14
`MAX_PTZ_REPLY_LENGTH`	11
`MIN_PTZ_REPLY_LENGTH`	4
`MIN_PTZ_INQUIRY_LENGTH`	3

We have examined implementations for sending commands and inquiries and the implementation for constructing the final packet and sending it to the camera for execution. The only remaining element of *VISCA* packet communication is receiving responses.

This is where the `ReceiveVISCAPacket` method is used. The implementation in both drivers is relatively complex; though effort was made to simplify it in the modified driver, both use the same basic algorithm. The algorithm for receiving a packet is described as follows in simplified pseudocode:

Algorithm 1 Receive the response to a *VISCA* Packet

```
1:  procedure ReceiveVISCAPacket(cmdReply)
2:      Poll hardware for data d
3:      if d is available AND d ≠ storedReply then
4:          Read d from serial port
5:          for bit b in d do
6:              Ignore b prior to header (0x90)
7:              Append b to cmdReply
8:              Ignore b after terminator (0xFF)
9:          end for
10:         Save cmdReply as storedReply
11:         Return cmdReply
12:     else
13:         Return 0
14:     end if
15: end procedure
```

The actual implementation uses the *poll* library for synchronous I/O multiplexing and includes significant amounts of validation and optimisation, however the basic operations match what is outlined above.

Once the functionality to send and receive packets had been implemented, it required testing.

It was decided that the best way to do this was by implementing a command and an inquiry and comparing the results to *sonyevid30*. The final driver has many commands and inquiries, but it was decided the first should be something relatively simple and with a visible effect.

A new method named `SetCameraPower` was created, based on the `PowerOn` command in the unmodified driver. The purpose of this method is to query the current power state of the camera and switch it on or off depending on the value of an input parameter. The outcome of calling this method is easy to confirm visually.

A partial listing of *SetCameraPower*, demonstrating the use of the `SendVISCACommand` and `SendVISCAInquiry` methods, is presented on the next page.

```
    /* Actual command that is sent would be '0x8X 0x09 0x04 0x00 0xFF',
    where X is the address of the camera (0x81 for the first camera connected) */
    unsigned char powerInquiry[] = {0x09, 0x04, 0x00};

    /* Example reply: Y0 50 03 FF - the Y0 and FF are just the start and terminating packets,
    the 50 indicates an inquiry reply and 03 indicates the camera is currently powered down */
    unsigned char powerInquiryReply[MAX_PTZ_PACKET_LENGTH];

    /* Send a VISCA inquiry: Arguments are the command to send,
     * the length of the command in bytes (ignoring the header and terminator),
     * and the variable to store the reply in */
    int powerInquiryReplyLength = SendVISCAInquiry(powerInquiry, MIN_PTZ_INQUIRY_LENGTH,
     ↪  powerInquiryReply);

    if (powerInquiryReplyLength < MIN_PTZ_REPLY_LENGTH)
     /* Reply must be at least 4 bytes including header and terminator */
    {
        std::cerr << "Did not receive enough bytes from inquiry, problem with connection" <<
         ↪  std::endl;
        return -1;
    }

    else if (powerInquiryReply[2] == 0x02 && powerUp == true)
    {
        /* If the third byte of the reply is set to 0x02 the power is already on, do nothing
         ↪  */
        std::cerr << "Power is already on, command ignored" << std::endl;
        return 0;
    }

    else if (powerInquiryReply[2] == 0x02 && powerUp == false)
    /* If the power is on and we have specified the camera should power down */
    {
        /* The power on and power off commands are defined under CAM_POWER in the VISCA
         ↪  standard */
        unsigned char powerOffCommand[] = {0x01, 0x04, 0x00, 0x03};
        /* Notice that the address and terminator are left out, this is handled by
         ↪  SendVISCACommand() */

        /* Arguments are the command to send and the length of the command in bytes */
        return SendVISCACommand(powerOffCommand, 4);
    }

    /* If the third byte of the reply is set to 0x03 the camera is on standby, power up */
    else if (powerInquiryReply[2] == 0x03 && powerUp == true)
    {
        unsigned char powerOnCommand[] = {0x01, 0x04, 0x00, 0x02};
        std::cerr << "Attempting to power up the camera..." << std::endl;

        return SendVISCACommand(powerOnCommand, 4);
    }
```

Listing 6: Implementation for reading and setting camera power status

A summary of this method is presented on the next page.

1. Initialise the variables to store the hexadecimal inquiry packet and the reply.
2. Send a power inquiry using `SendVISCAInquiry`.
3. Perform validation on the reply and use it to detect the current state.
4. If required, construct the appropriate command and send it using `SendVISCACommand`.

After this method was completed, the first hardware test using this driver was conducted. Serial communication worked perfectly, as did sending and receiving *VISCA* packets. The driver could send a power inquiry, read the response and send the appropriate power command, which was executed successfully by the camera.

However, this testing discovered an issue. Even if the camera was on standby, it would report that it was already powered up. The *gdb* debugger was used to examine the system, particularly the value of the *powerInquiryReply* variable, which revealed that the camera was sending an incorrect information packet.

This issue was not present with the *D30* camera, and appeared intermittently in both drivers, so determining whether this was an issue with the *EVI-D70* in general or the specific hardware used during testing would require further investigation.

In order to make debugging easier, a simple function to print the hexadecimal representation of a *VISCA* packet to *stdout* was added.

This method operates directly on the *unsigned char* representation of the data, printing a hexadecimal representation of the input. If the system was object-oriented, it would have been possible to override the << operator of the *VISCA* packet class. However, any difference in ease of use or functionality would have been minimal.

```
1  /* Utility function to print a VISCA packet as a hexadecimal string for debugging purposes */
2  void SonyPTZ::PrintVISCAPacket(const char* infoStr, unsigned char* cmdStr, int cmdLen)
3  {
4      std::cout << infoStr << ": "; // Print the information string passed into the function
5
6      for (int i = 0; i < cmdLen; i++) // Iterate through the packet and print the hexadecimal
       ↪  pairs
7      {
8          printf(" %.2X", cmdStr[i]);
9      }
10
11     std::cout << std::endl;
12 }
```

Listing 7: Method to print the hexadecimal representation of a *VISCA* packet

A similar implementation exists in several *Player* PTZ drivers, including *sonyevid30* and *canonvcc4*, a driver for Canon PTZ cameras which use a different packet-based protocol.

As with the previous iteration, the end of the testing stage led to the current work being thoroughly documented and refactored where required. Variables were renamed for clarity and any extraneous logic was removed.

The core of the driver was intended to be compact and it had kept to this so far. The next iteration would see the driver evolve from a proof of concept into something of practical use.

3.3 Mid-Project Iteration

At this point, the modified driver had fully-functional *VISCA* communications, sending commands and inquiries and receiving replies to them. The foundations were in place to continue development.

The driver could not yet send pan, tilt or zoom commands and had no way to store camera parameters. It was important to define a data structure to represent an *EVI* camera as this would form a crucial component of the calibration process which would be implemented later in this iteration.

The driver needed to differentiate between camera models and store the parameters associated with the currently connected model in a more comprehensive way than the existing system.

In order to do this, the *sony_ptz_cam_t* data structure introduced in the previous chapter was implemented. This contains all of the required fields to hold state information in memory; including calibration status, hardware limits and a unique model identifier for each model of *EVI* camera.

This structure was revised throughout the iteration, particularly when calibration was implemented and the requirement for new fields became apparent.

3.3.1 Implementing Pan and Tilt

Before moving on to calibration, pan and tilt commands were implemented. Pan was implemented first, although there are no implementation differences between pan and tilt other than the contents of the *VISCA* packets and the respective hardware limits.

VISCA provides a level of abstraction where every hardware function can be implemented using some combination of commands, inquiries and / or replies. This dramatically simplifies the addition of new features.

There are two ways to pan, tilt or zoom a PTZ camera. One is to use an absolute movement command (e.g. "tilt to +45°") and the other is to set the speed and direction of movement. The former would be used for precise positioning and the latter would be more suitable for tracking an object. *sonyevid30* provides both of these functions and the implementations provided a base for adding similar functionality to the modified driver.

Specifying a position for the camera to pan, tilt or zoom to is referred to as absolute (rather than relative) movement. The client sends a position request to the camera and it moves as close as possible to that position. The *SetAbsolutePanTilt* method was implemented to perform this function.

This method performs basic error detection to ensure input is within the range of acceptable inputs for the hardware, thresholding values which fall outside of the specified hardware limits. The constants described in the previous chapter (such as *D30_MIN_TILT_DEGREES* were added to store these limits.

It is important to note that at this stage, the camera did not have any concept of degrees (or radians, or arcseconds), only internal camera units represented as hexadecimal values. At this point the pan and tilt values passed into the `SetAbsolutePanTilt` method were effectively

meaningless, the calibration process is what allows the camera to convert between internal and external representations.

Relative movement was implemented in a separate function named `SetPanTiltSpeed`. As the method name indicates, this was limited to pan and tilt speed control as the required changes to *Player* for zoom speed control had not yet been implemented.

3.3.2 Allowing Client Access to Driver Functionality

At this point, it was necessary to provide a means for clients to access the functionality provided by the driver. Client software uses a *PtzProxy* object to interface with the driver via *Player* and does not usually call any driver methods directly.

The communication between the client and the driver is done inside the driver's `Main` method during the message processing loop. This loop updates the variable which stores incoming camera data, processes messages from the hardware by calling the appropriate internal methods and checks if the driver needs to terminate. The thread is then suspended until more data is available. This method makes use of the *pthreads* library and runs in a separate thread to the rest of the driver to allow for continuous message processing.

A variable of type *player_ptz_cmd_t* (provided by *Player*) is used to store requested pan, tilt and zoom values and is updated each time this loop is executed. This is well implemented in the unmodified driver and was added piece by piece, refactored and extensively tested to ensure efficiency when added to *sony-ptz*

3.3.3 Accessing the Current Camera Position

We have discussed the methods for changing the position of the camera, but we have not yet looked at how to read position information from the camera. This is important, if the camera is 'blind' then the number of possible use cases would be dramatically reduced.

Therefore, the next objective of this iteration was to find a way to query the current position of the camera, and then find a way to relate this to a value in degrees.

The first step to receiving position data from the camera was to find the correct inquiry within the technical manuals. The inquiry to return the current camera position is named 'Pan-tiltPosInq' (see Appendix A) and returns a packet in the following format:

- `WWWW` := Current Pan Position
- `ZZZZ` := Current Tilt Position

The actual position data is encoded in 4 bytes (32 bits). The method to retrieve this uses left bit-shifting to extract the data from the rest of the information return, the same technique used in *sonyevid30*. In order to update both pan and tilt position within one function, the appropriate variables are passed in as pointers which are modified within the method.

The listing below presents the final version of the method, which includes conversion between camera units and degrees using a conversion value which can only be calculated using calibration. *ptzCamConfig* is the *sony_ptz_cam_t* object which stores the camera parameters.

The original version of the method simply returned the *absPanPosition* and *absTiltPosition* values in hexadecimal format, without conversion.

```
/* Updates the current pan and tilt position variables using pass-by-reference */
int SonyPTZ::GetAbsolutePanTiltPosition(short* panPositionDegrees, short* tiltPositionDegrees)
{
    unsigned char panTiltPositionInquiry[] = {0x09, 0x06, 0x12}; // 'Pan-tiltPosInq'
    unsigned char panTiltPositionReply[MAX_PTZ_PACKET_LENGTH];
    int panTiltPositionReplyLength;

    short absPanPosition, absTiltPosition;

    if ((panTiltPositionReplyLength = SendVISCAInquiry(panTiltPositionInquiry, 3,
     ↪  panTiltPositionReply)
        ) <= 0)
    {
        return panTiltPositionReplyLength;
    }

    /* Get the 4 bytes that represent the current pan position */
    absPanPosition = panTiltPositionReply[5];
    absPanPosition |= (panTiltPositionReply[4] <<  4);
    absPanPosition |= (panTiltPositionReply[3] <<  8);
    absPanPosition |= (panTiltPositionReply[2] << 12);

    /* Set the current pan position with respect to the conversion factor */
    *panPositionDegrees = (short) round(absPanPosition / ptzCamConfig.panConversionFactor);

    /* Next 4 bytes are the current tilt position */
    absTiltPosition = panTiltPositionReply[9];
    absTiltPosition |= (panTiltPositionReply[8] <<  4);
    absTiltPosition |= (panTiltPositionReply[7] <<  8);
    absTiltPosition |= (panTiltPositionReply[6] << 12);

    /* Set the current tilt position */
    *tiltPositionDegrees = (short) round(absTiltPosition / ptzCamConfig.tiltConversionFactor);

    return 0;
}
```

Listing 8: Method for retrieving position data from a connected camera

3.4 Implementing Hardware Calibration

A major goal of this project was to devise a way to calibrate the driver for use with a specific camera, allowing it to fully exploit the capabilities of that particular model. This involved reading properties from the camera instead of setting these as hard-coded values within the source code the way *sonyevid30* does.

Calibration is used within the driver for accessing the maximum speed and the upper and lower limits of pan, tilt and zoom. It is also crucially important for providing accurate conversions between internal camera units and degrees.

Unfortunately, there is no function within the *VISCA* specification to read the minimum or maximum pan and tilt values from a camera. Therefore, an implementation was devised from scratch. The design and implementation of this novel calibration function is the key technical contribution of this project.

Calibration is one of the key features which differentiates *sony-ptz* from other PTZ camera drivers and much of the second iteration was dedicated to it. Research at the time of publication indicated that no other available PTZ driver had support for calibration, making *sony-ptz* the first driver to implement this.

3.4.1 Identifying a Camera Model

The first step to performing calibration is to devise a means of identifying which model of camera is currently connected. All *EVI* cameras have a model identifier within the firmware and there is a *VISCA* inquiry to retrieve this information.

There is no unique hardware identifier but it is a fair assumption to state that all (undamaged) cameras of the same model should have the same pan, tilt and zoom limits.

Three new constants were defined as preprocessor directives to store the model identifiers of the *D30*, *D70* and *D100* cameras. All variants of a model share the same model identifier.

This allows the calibration method to compare the model identifier reported by the connected camera with the data for each model and correctly configure the driver for that model. Within this system and the original driver, an unknown model is represented as a *D30*. This compatibility mode allows the driver to control any *VISCA* compatible camera, although control will not be optimal unless explicit support for that model is added.

3.4.2 Retrieving the Maximum Speeds

The next stage is to gather the maximum supported camera speed, represented in internal camera units. Setting or reading absolute speeds in rad/s or another universal unit is not currently supported.

It is possible to obtain the maximum pan and tilt speeds using the 'Pan-tiltMaxSpeedInq' *VISCA* inquiry, which is similar to all other *VISCA* inquiries we have seen and therefore not particularly noteworthy.

The method `GetMaximumSpeeds` implements this inquiry and saves the reported values to the *maxPanSpeed* and *maxTiltSpeed* variables within the *sony_ptz_cam_t* data structure for later access. The minimum supported speed is zero (i.e. stationary), the maximum speeds vary by camera model.

3.4.3 Getting the Pan/Tilt Status

At this point, the driver could recognise which model of camera was connected, and set the maximum speed accordingly. The most important part of the calibration process was to accurately determine the minimum and maximum pan and tilt limits for the hardware.

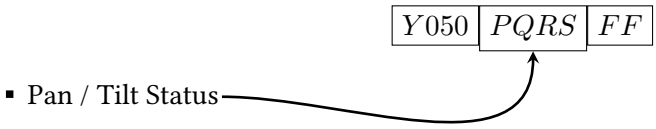
The *sonyevid30* driver uses stored constants to represent hardware limits - within this driver the maximum pan is set universally to +100°, far below the maximum pan of an *EVI-D70P* camera. The maximum tilt value is also mis-configured.

Calibration should convert limits in camera units to limits in degrees. The supported hardware limits in camera units are provided within the technical manuals, however testing proved that these values were not completely accurate. Given this, the pan and tilt limits needed to be set dynamically.

Because there is no way to request the limits from the camera, a different solution was devised. There is a poorly documented inquiry named 'panTiltModeInq', despite the name which appears to be related to determining if the camera is set to position or velocity mode, this inquiry returns the status of the last-executed pan or tilt command.

There was little documentation for this inquiry available, so several days of testing and experimentation were required to create a suitable implementation.

The packet returned by this inquiry takes the following form:



Two bytes of data are returned. There is a small amount of information on this inquiry available within the technical manuals[1] listing a series of status codes. The main work involved was in in reconciling these status codes with actual camera movements.

The table of status codes provided by Sony is partially recreated below. There are a total of eighteen different status codes, but for calibration purposes we are only interested in seven.

Table 3.2: Pan-Tilt Status Codes

P	Q	R	S	
0---	----	0---	---1	Pan has reached the left endpoint.
0---	----	0---	--1-	Pan has reached the right endpoint.
0---	----	0---	-1--	Tilt has reached the top endpoint.
0---	----	0---	1---	Tilt has reached the bottom endpoint.
0---	01--	0---	----	Pan-Tilt is moving.
0---	10--	0---	----	Pan-Tilt operation completed.
0---	11--	0---	----	Pan-Tilt operation failed.

[1] Sony EVI-D70P Technical Manual, pg. 51
Sony EVI-D100 Technical Manual, pg. 39

Depending on the position of the bits in the return data, it is possible to determine if a movement command is in progress, succeeded or failed. It is also possible to query the status and determine when a limit has been reached. This was a breakthrough for implementing calibration, because it provided a way to accurately determine the limits of connected hardware.

Recall that *VISCA* places the most-significant bit first, so an "S-value" of $0x02$ would be equivalent to the binary representation for pan reaching the right endpoint (`--1-`).

3.4.4 Implementing a Calibration Method

During the previous pages, we have identified and discussed the components required to implement a fully-featured calibration function. All that remained was to combine these in a single method which would be called from the `MainSetup` method when the driver was initialised.

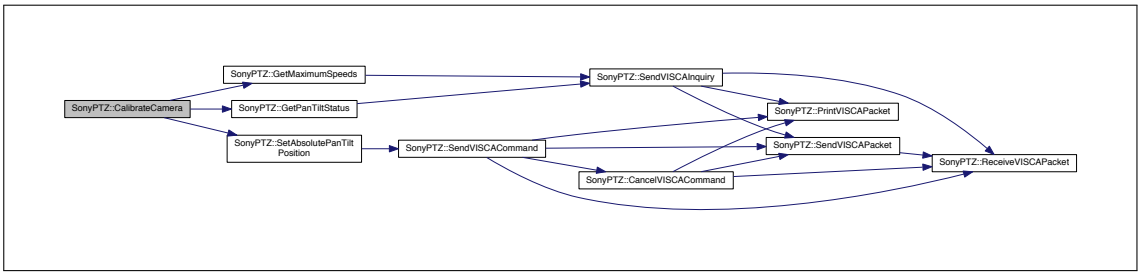


Figure 3.2: Call graph for the `CalibrateCamera` method. [Doxygen]

The `CalibrateCamera` method reads the camera model identifier set in `GetCameraModel`, sets the maximum speeds and then performs a series of pan and tilt movements simultaneously to determine when a limit is reached. The is accomplished using two test positions, one for pan and one for tilt, which are incremented or decremented depending on whether the method is searching for a lower limit or an upper limit.

After each movement the pan-tilt status is retrieved as described in the previous section. The fourth byte in the packet (corresponding to S) which changes when a limit is reached is read and if any of the values indicating a limit are present then the current test position is set as the minimum or maximum limit for pan or tilt accordingly. This makes calibration extremely accurate as it is reading values directly from the hardware.

Once hardware limits can be determined, it is possible to relate internal camera units to degrees using simple division or multiplication:

1. `panConversionFactor = maxPanAngleCameraUnits / maxPanAngleDegrees`
2. Convert Position to Degrees: `absPanPosition / panConversionFactor`
3. Set Position in Degrees: `panPositionRequestDegrees * panConversionFactor`

The same calculations apply for tilt. See Appendix B for the source code of the calibration method.

3.4.5 Making Calibration Data Persistent

The concept of a driver configuration file has been discussed at length within this dissertation. Recall that these are text files containing space-delimited key-value pairs which store *Player* and driver-specific configuration data. The initial version of the calibration function completed in approximately two minutes. After optimisation to simultaneously test pan and tilt limits, this improved to thirty seconds.

Despite this improvement, requiring a calibration process each time the camera is connected is not ideal, regardless of how long calibration takes. At a preliminary software demonstration, the project supervisor suggested that saving this data to a file for future reuse would be preferable.

The *ConfigFile* class provided by *Player* includes an API for reading from configuration files but not writing to them. In order for the driver to be as self-contained as possible, it was decided that rather than using a separate file with its own file format, the existing configuration file could be used to store this data.

Logic was added to the end of the calibration method to open the configuration file (the path of which was accessed using *filename* property of the *ConfigFile* class), convert a number of fields in the *sony_ptz_cam_t* structure to their string representation using a C++ *std::ostringstream* object and save this to the end of the file.

Later in the development process this behaviour was modified to run a calibration by default and print the suggested calibration settings to the screen. This is preferable to writing to the file directly whilst it may be in use by *Player*, it also simplifies the logic required as there is no need to parse and write config files which could potentially cause file corruption.

Changes were made in `MainSetup` to read existing calibration data from the configuration file if any is available or to calibrate the camera if not.

3.4.6 Preparing for a Demonstration

At this point, the driver could control camera pan and tilt position and speed, retrieve the current camera position in a sensible format and included a working calibration function. The second iteration was almost at an end.

Many of the original objectives had been met, but the question at hand was how to demonstrate this. With the mid-project demonstration nearing, a method was created which aimed to show all of the functionality currently implemented by the modified driver and how it improved on the unmodified system.

This method was designed to demonstrate the following driver features:

- Serial Communication.
- Sending and receiving *VISCA* packets.
- Controlling the pan and tilt using a target position in degrees.
- Returning the current position converted from internal units to degrees.
- Calibration with a camera to determine the range of movement, maximum speeds, etc.

The following diagram demonstrates the structure of the demo method:

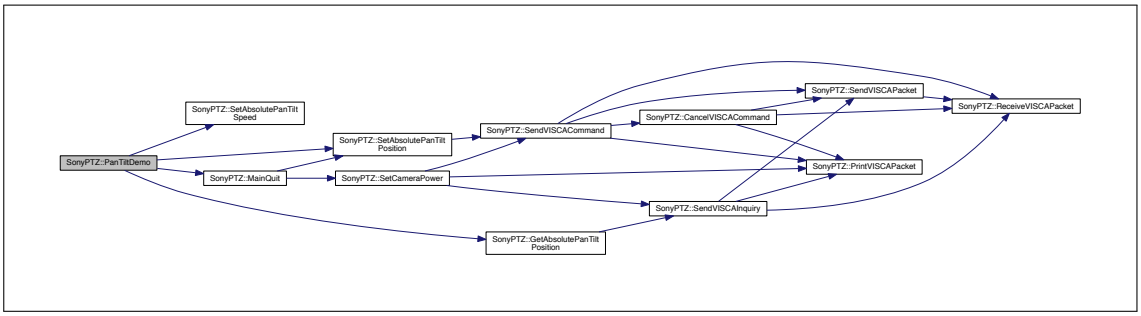


Figure 3.3: Call graph for the *PanTiltDemo* method. [Doxygen]

With this completed, the mid-project iteration was at an end and the driver had successfully met many of its objectives.

3.5 Final Iteration

There were a number of tasks remaining for the final iteration, although it was the shortest of the three. The driver did not yet support zooming (or setting zoom speed) and the changes outlined in the first chapter still needed to be applied to the *Player* source tree.

The mid-project demonstration highlighted a need to read the current position reported by the camera more accurately, whilst post-iteration code reviews highlighted the need to refactor various methods within the system. There were additional requirements for the final demo such as incorporating a live video stream to demonstrate zoom functionality.

3.5.1 Modifying *Player* and Generating a Patch

The first step in this iteration involved implementing the changes to the *Player* structure, including changes to the *libplayercore* and *libplayerc++* libraries as outlined in the first chapter. This predominantly involved adding an extra zoom speed parameter to existing functions such as `playerc_ptz_set_ws`.

Another task involved updating the C++ methods which call these functions, such as `PtzProxy::SetSpeed`, accordingly.

It was also necessary to update *libplayerinterface* which defines data structures for interfaces - in this case the *player_ptz_cmd_t* structure used by PTZ cameras needed an additional field for zoom speed. It is worth noting that this addition could potentially break compatibility with older PTZ drivers who expect this structure to be a certain size. *sony-ptz* checks the size of the field for both the original size and the size when using a *Player* server compiled with these changes.

Proposed additions to *Player* are sent to the maintainers in the form of patch files. A patch was generated by downloading the latest *Player* trunk from the *svn* repository, making the required modifications and then running `svn diff > zoom_speed.patch` to generate a patch file in the standard UNIX *diff* format.

3.5.2 Implementing Zoom

Figure 3.4: USB converter used for viewing video output from a camera. [creativecomputing.net]

Next, zoom support was added to the driver by implementing four additional methods. These are `SetAbsoluteZoomPosition`, `StopZoom` and the only two public methods within the driver, `SetWideZoom` and `SetTeleZoom`. The former two are very similar to the respective methods in *sonyevid30*.

It is possible to visually confirm when the camera is panning or tilting, this is not the case when zooming so an alternative approach is required. Fortunately, each *EVI* camera provides *RCA* and *S-VIDEO* outputs. In order to access this video stream, a display which supports one of these connections is usually required.

These are difficult to obtain so research was conducted to find an alternative solution. An *S-VIDEO* $\rightarrow$ USB converter (**Figure 3.4**) was obtained, which works as a *Video 4 Linux* compatible input. The device allows a live video feed from the camera to be displayed on the screen of a Linux computer using a compatible media player. This was useful for testing the zoom functionality and to provide an additional component to the final demonstration.

New fields were added to the *sony_ptz_cam_t* structure to store the minimum and maximum fields of view and speeds for zooming. The calibration process does not currently calibrate for zoom because the values depend on the camera and lens used, the values for the default lenses were added as constants for each supported camera. The minimum zoom speed is set to $0x02$ on the *D30* and $0x00$ on the other cameras, the maximum supported zoom speed is $0x07$ on all cameras. Currently, the zoom support in the driver is not as advanced as the support for pan and tilt.

However, this implementation supports zooming to any arbitrary zoom level and allows zooming to either a wide or telescopic angle at a specified speed, an improvement on *sonyevid30* which does not support zoom speed at all. Further development work would allow arbitrary levels of zoom at arbitrary speeds.

The demonstration function implemented in the previous iteration was modified to demonstrate the zoom functionality and changes were made to the main event processing loop in the driver to leverage the changes made to *Player*.

3.5.3 Completing the Driver

By this point, the driver was almost ready for its first release. It is important to note that the current implementation does not implement every conceivable feature and a number of areas would be candidates for further development work. Despite this, there is a solid foundation in place which any future development can build upon in a way that would have been very difficult without the refactoring and development efforts outlined in this chapter.

Before release, the driver underwent significant amounts of testing which will be described in the next chapter. As well as this, a thorough code review led to a number of refactoring

steps including removing unused variables and methods, renaming and a general cleanup of the codebase. Finally, some of the validation logic was simplified to reflect changes which had been made since it was first implemented.

The first release of the driver meets all of its stated objectives and provides an excellent alternative to *sonyevid30* with new features and a cleaner codebase which could easily be extended to support additional hardware.

After this report is submitted, all of the source code created during the development process will be made available to the open source community, in accordance with the aims of this project and the use (either as a reference or directly) of significant amounts of GPL-licensed code from the original driver in *sony-ptz*.

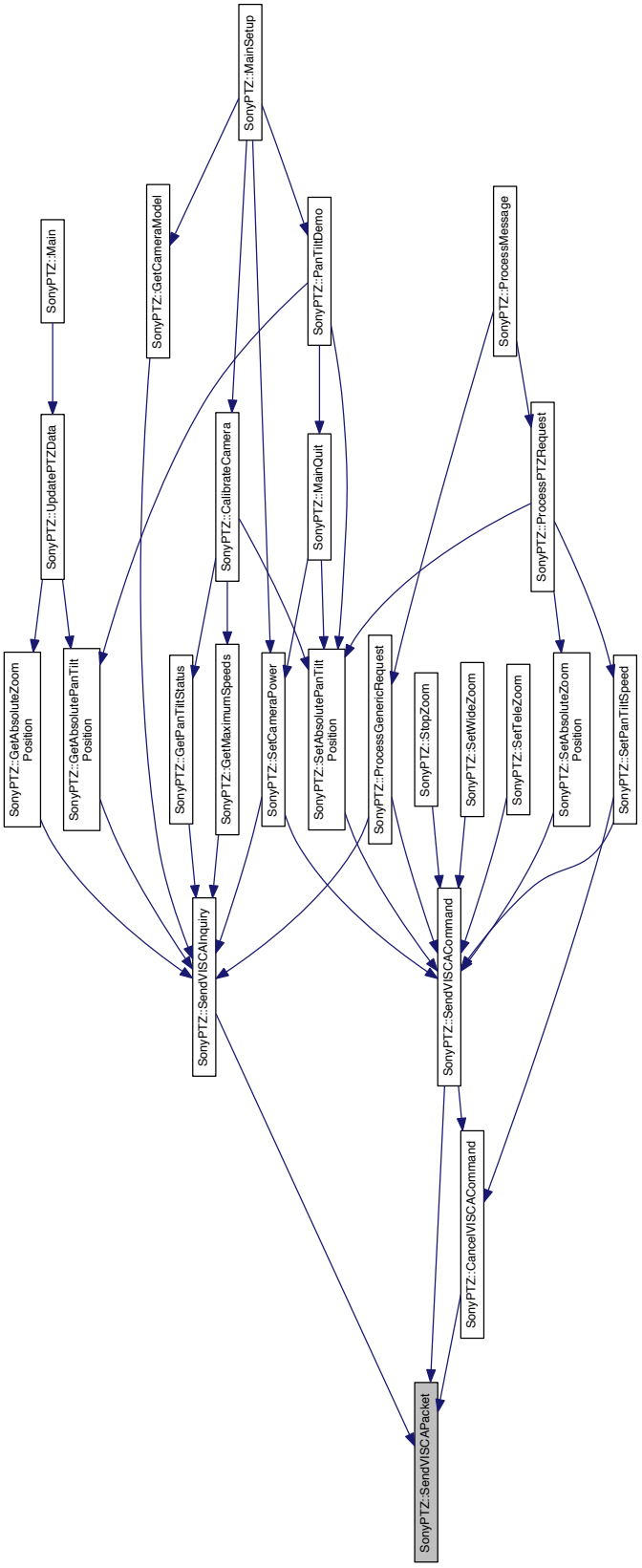


Figure 3.5: Call graph illustrating the overall structure of the final driver. [Doxygen]

Chapter 4

Testing

> Relay #70, Panel F.
> Moth in relay.
> First actual case of bug being found.
>
> Grace Hopper, 1947

This chapter discusses several topics which are broadly related to testing. Software testing involves much more than just creating a series of unit tests or filling in a table, testing is proof that the software does what it claims to do. As the software in question is a hardware driver, testing is even more crucial to ensure the hardware involved is not damaged in any way.

In UNIX-like operating systems, hardware drivers are often implemented as kernel modules and an improperly coded driver can potentially destabilise the entire operating system. This is not the case with *sony-ptz* due to the higher level of abstraction, but the driver was tested to the same standards of stability, performance and functionality that any other hardware driver would be expected to meet.

Automated testing when hardware is required raises a number of challenges, for this reason traditional unit-testing was not one of the testing approaches used within this project. Despite this, a significant amount of testing was carried out during the development process. Testing included static analysis, compatibility testing and data verification, among other methods.

Throughout this chapter, we will examine and justify the various forms of testing that were carried out during the project.

4.1 Defensive Development: Software for an Uncertain World

Defensive development is a technique which aims to create secure, error-resistant systems by anticipating and minimising inputs which can lead to undefined states. Defensive practices were used heavily in the first major firewall deployment at AT&T [3] in the 1990s and are well-suited to software requiring a high degree of stability.

The modified driver created in this project aimed to avoid issues caused by the hardware being in an undefined state. The primary way this was achieved was through making as few assumptions about the hardware state or data within the system as possible. There is a large amount of data validation and error detection throughout the driver, every statement which actively affects the hardware is placed inside a conditional statement which detects errors. This means that even if the data sent or received is unexpected, the hardware will always remain in a clearly defined state.

In the case of errors which should theoretically never occur, the typical response is to terminate execution of the driver, as this indicates a severe hardware of software misconfiguration.

With less severe issues, a thresholding system is in place which will adjust values to acceptable parameters. For example; if a tilt value of greater than 90° is sent to the camera, the driver will intercept this value and reduce it to the maximum supported tilt value, whether that is 90° (for the *EVI-D70*) or a lower value.

The outcome of this design is that the driver protects the hardware from potential damage as it does not rely on the existence of hardware safety features, although these are present.

4.2 Position and Angle Verification

One of the key features of the driver is the ability to return the current pan, tilt and zoom positions of a connected camera. An important question related to this is, "how can we prove the values reported are accurate?". Without developing a highly complex hardware test harness, it is impossible to conclusively prove that these values are accurate.

Figure 4.1: The *Theodolite* application. [Hunter Research & Technology Ltd.]

However, the accuracy can be proven beyond reasonable doubt. One method used was the concept of testing using an *inverse function*, an example of this would involve commanding the camera to pan or tilt to a specific angle in degrees and then requesting this position in internal camera units. After this, the camera would be returned to the default position and the previous position would be requested again, in camera units rather than degrees.

If the original position in degrees is returned, this verifies that the conversion between camera units and degrees is consistent and likely to be correct.

An additional way of verifying camera positions is to use a measuring tool. The particular tool used for this verification was an *iPhone* application named *Theodolite*, which provides a graphical display of horizontal and vertical angles and various other measurements. This can be used for verification that an angle which appears to be 90° is in fact exactly 90°. The way this tool was used was for the driver to command the camera to a known position (typically a minimum or maximum limit) and then verify that the actual angle matched the expected angle.

This was used to confirm an issue in the conversion between degrees and camera units, after the maximum tilt on the *EVI-D70* was incorrectly reported as 83° during testing.

4.3 Hardware Test Function

The method designed for the two project demonstrations is itself a form of testing. All of the externally exposed driver functionality is executed in order, with copious amounts of error detection. If there is an issue or a misconfiguration then this function will identify it, albeit not in a fully automated way.

An example of this would be how the method reports the position of known angles in degrees, if the values reported are incorrect then this indicates an issue with the driver, which can then be investigated.

4.4 Static Analysis

Static analysis is a widely used method for identifying potential issues with a system when it is impractical or impossible to detect these errors at runtime. Due to the need for hardware interaction and the importance of minimising potential risk, static analysis is extremely well-suited for use on hardware drivers or other systems requiring a high level of stability [11]. Two tools were used for static analysis, *cppcheck* and *scan-build*, the static analysis tool included with *clang*.

One of the benefits of *cppcheck* is that the executable is small in size, self-contained and does not require any resources other than the source file to analyse. The analysis performed identifies issues such as potential memory leaks, variables which have greater scope than required (e.g. global variables used within only one method) and unused functions. This was extremely useful for identifying potential issues with the code, although it requires careful attention to false-positives.

The other tool used; *scan-build* [19], is arguably more comprehensive than *cppcheck*, although it is not aimed specifically at C++. Another consideration is that it uses the standard build system by adjusting the *CC* and *CXX* environmental variables to point to a 'fake' compiler, which performs static analysis as an additional step before calling the regular compiler.

This tool was used to identify an array index out-of-bounds error, which was causing a segmentation fault during the early part of the second iteration. In addition to judicious use of the *gdb* debugger (as we will explore in the next section), *scan-build* helped identify an error in the size of a variable to store the reply from a *VISCA* inquiry. The error in question was caused by the fact that the variable had its size set to `MAX_PTZ_MESSAGE_LENGTH` (14) rather than `MAX_PTZ_PACKET_LENGTH` (16). This seemingly innocuous oversight was causing the driver to crash.

Whilst static analysis is not a replacement for testing with 'live' code, the inclusion of these tools into the project greatly assisted testing and debugging efforts.

4.5 Data Verification using a Debugger

Although it is not strictly a form of testing in the traditional sense, using a debugger to verify values was extremely important in testing and isolating issues with the system. Before the method to print the contents of *VISCA* packets was implemented, the only way to see and verify the values of packets was using a debugger.

The debugger used in this project was *gdb*, which should require no introduction. There were limited resources available describing how to use a debugger with *Player*, but experimentation yielded a suitable workflow, which is described on the next page.

1. Run *gdb* with *player* as an argument: `gdb player`
2. On the prompt that appears, use the `run` command with the driver configuration file as an argument, `run sony-ptz.cfg`
3. After the driver has started to load up, hit Ctrl - C to trigger a *SIGINT*, which will halt execution.
4. Set breakpoints for the lines or methods you wish to examine.
5. Use the `run` command again to restart execution, which will break when required.

As *VISCA* packets are best represented using hexadecimal, the `p/x` (print in hexadecimal) debugger command was very useful. Using a debugger provided a huge amount of control over the system and helped isolate many issues, the debugger was also used extensively when implementing calibration because it provided an easy way to view large numbers of *VISCA* reply packets. This was instrumental in identifying the correct values corresponding to the pan and tilt status codes described in the previous chapter.

4.6 Compatibility Testing

An important aspect of testing the driver involved ensuring that all three supported hardware models functioned correctly and consistently. This includes the calibration process, basic driver operation and zoom control. The driver aims to provide a consistent interface which works the same way regardless of which camera is connected.

The first camera used for testing was the *EVI-D70P*, as this camera displayed the greatest number of issues with the *sonyevid30* driver. Once testing had indicated that the problems with the *D70* were not present in *sony-ptz*, attention turned to thoroughly testing the *D30* and *D100* cameras. Optimisations made for quicker calibration on the *D70* caused issues when used with the other models so they were disabled when not using the *D70*.

In the first chapter of this dissertation, we saw how the *playerv* utility would crash when used with the *D70*, this behaviour is fixed in the new driver. Similarly, the *D100* was mistakenly recognised as a *D30*, which has the same pan and tilt limits but is slower, this has never been a problem for *sony-ptz* as compatibility was a key focus of development from the start and this was one of the first bugs fixed.

Further to this, an issue was identified whereby the zoom control on the *D70* was reversed. It was not possible to reproduce this issue with either the unmodified or modified drivers, indicating that the issue could have been transient and caused by a hardware or software misconfiguration.

4.7 Manual Testing

As well as the testing described previously, the driver underwent a series of manual tests designed to test all functionality exposed to client software. The results of these tests are presented over the following pages.

Table 4.1: Results of testing the *sony-ptz* driver for various use cases

Test	Expected	Actual	Pass
Pan to Minimum Angle	The camera correctly pans to the minimum supported angle in one continuous motion.		✓
Pan to Maximum Angle	The camera correctly pans to the maximum supported angle in one continuous motion.		✓
Tilt to Minimum Angle	The camera correctly tilts to the minimum supported angle in onecontinuous motion.		✓
Tilt to Maximum Angle	The camera correctly tilts to the maximum supported angle in onecontinuous motion.		✓
Pan to 70°	The camera pans to exactly 70°.		✓
Tilt to -10°	The camera tilts to exactly -10°. Demonstrates correct handling of negative position values.		✓
Pan to 30°, Tilt to 20° and report current position	Pan is exactly 30°, tilt is exactly 20° and both the position reported and angle verification reflect this.		✓
Zoom to Maximum and report current zoom level	The camera zooms to maximum supported zoom level and reports the zoom as 100%.		✓
Zoom to Minimum at lowest speed	The camera zooms to the minimum supported zoom level at the lowest speed as indicated visually and by verifying the speed parameter passed to the zoom method.		✓
Calibrate supported hardware and verify angles	All three models of hardware should correctly run the calibration process and produce calibration data with the correct minimum and maximum values for pan, tilt and speed.		✓
Read calibration data from configuration file	The driver correctly sets internal variables according to the calibration data with the correct minimum and maximum pan and tilt values and speeds.		✓
Zoom to half of the maximum zoom level at the lowest speed	The camera zooms to exactly half the maximum zoom level at the lowest supported speed.	Not yet implemented.	✗

Report minimum and maximum zoom speed and cross-reference this with data provided in technical manuals	The camera correctly reports a maximum zoom speed of 0x07 for all cameras and minimum values of `0x02` for the *EVI-D30* and `0x00` for other models as outlined in the manuals.	✓

4.8 Issues Identified in Testing

Testing was mostly successful, with only a few minor issues discovered. The most significant issues identified were inaccuracies in the function to return the camera's current position in degrees.

The first problem discovered was an error where the pan conversion value was calculated incorrectly for one particular camera, this resulted in highly inaccurate values being returned. The other issue was more subtle and was related to a rounding error caused by using integer division, this was fixed by explicitly rounding the result of the affected calculation.

Testing also identified differences in the way each camera handled hardware calibration. As this had first been implemented for the *EVI-D70*, certain changes were required to ensure calibration functioned correctly with the other types of camera. It is possible to implement hardware-specific calibration optimisations to improve calibration speed or accuracy and some of these were implemented for use with the *D70*.

Other than some minor issues, the testing processes put in place revealed that the system was generally functioning as intended. There are always challenges when testing software which interacts with hardware, but the wide variety of techniques used in this project provided multiple ways to verify the driver was functioning correctly.

Chapter 5

Evaluation

> Being idealistic really helps you overcome some of the many obstacles put in your path.
>
> Andy Hertzfeld

We have arrived at the final chapter of this dissertation, after exploring an introduction to PTZ cameras and *Player*, looking at the design of the new driver, conducing the refactoring and development process and discussing some thoughts on testing. It is now time to evaluate the success of the project. 'Success' is a nebulous term, but a fair approximation can be achieved by looking at the objectives described in the first chapter and considering the extent to which they have been achieved.

The evaluation will be divided into two parts, the first will consider aspects of the overall project including methodology, planning and other aspects relating to project management. The second will be a technical evaluation of the system produced and the tools used during the process.

Before continuing, let us remind ourselves of the objectives the project was supposed to meet:

1. To create an improved hardware driver, built on the existing system with all of its core functionality intact.
2. To provide an implementation with a modern, clean and consistent code-base that encourages modification and extension.
3. To implement new features such as hardware calibration which make the modified driver outwardly superior to the existing one.
4. To produce comprehensive documentation for using the driver so it can be easily extended and integrated into projects.
5. To produce a patch file containing changes to *Player* to add zoom speed control and to submit this to the maintainers.
6. To produce a hardware test function which demonstrates the functionality of the modified driver.

These six items are mostly related to the technical aims of the project, however there are other considerations such as assessing the feasibility of the final design and evaluating what would be changed if the project were to be repeated.

5.1 Project Evaluation

Before we discuss the technical aspects, let us first consider the project in its entirety. One of the first steps in the project was identifying the requirements.

The primary requirement was to create an improved *Player* driver which maintained and improved compatibility with the target PTZ hardware and easier to extend than the original system. Additional requirements were quickly identified, such as implementing a calibration function to make the new driver more accurate than existing systems.

The requirements identified at the beginning of the project remained fairly static throughout, it is acceptable to say that the requirements analysis was accurate. The requirements did not have a huge amount of scope for changes, although the use of agile methodologies allowed the development process to seamlessly integrate new requirements as they were identified. Features such as saving calibration data to a file were not originally intended but were easily woven into the current iteration and then changed later as the project evolved.

The methodologies used in this project were adapted from 'standard' agile techniques such as *eXtreme Programming*, with modifications made wherever necessary to allow for a project conducted by one person. The choice of methodologies lent discipline to the development process (taking it far away from "code-and-hack") and provided structure which meant that there was never a time during development where it wasn't clear what the next steps were. *Kanban* also deserves an honourable mention, not just as an organisational tool but also as a source of motivation. It is difficult to procrastinate when you can glance at a chart and see research to do, code to write or testing to carry out, separated into tasks small enough to complete several a day.

Code reviews and extensive testing practices were a huge help in implementing the new driver, functionality was carefully added and refined in stages, rather than just being pieced together. Whilst it is arguable that any structured approach to development, including the traditional waterfall approach also offers this benefit; the development of this project was made more efficient by finding the right balance between rigid plan-based methodologies and traditional open source development approaches which tend to be relatively unstructured.

The chosen methodologies were generally suitable, refactoring was difficult at times when the current solution was technically acceptable, but it led to a much cleaner and more compact codebase. Refactoring included both the modification and re-use of existing code and refactoring of that modified code once it had been added to the new system.

One of the aims of this project was to release the final products to the open source community, all of the code and other resources that came out of this project will be made publicly available shortly after submission. Being able to view and modify source code to fit the needs of the project greatly assisted development and it is fitting (and indeed required, due to the GPL license of the original code) that the modified driver and other resources created during this project are made available to assist others in the future.

In general the project was successful, one mistake was prioritising the *EVI-D70* support above the other supported models. The other models were not tested until relatively late in the development process, and while the nature of *VISCA* guaranteed a degree of consistency between models, it would have been more sensible to regularly test the driver with all supported hardware in order to avoid potential issues.

5.2 Technical Evaluation

Now that the overall project has been evaluated, let us go into more detail about the technical aspects. The first question is whether or not the modified driver met its stated technical objectives, the answer is that it did in all areas and in some cases even exceeded them.

The driver provides full pan, tilt and zoom functionality for all three supported models, introduces a functional and relatively accurate calibration system (which cannot be found in competing software) and is designed to be easy to use, maintain and extend.

Despite the success of the driver, there are certain areas which could be developed further. These will be discussed later in this chapter.

5.2.1 Assessing the Suitability of the Development Tools

The development environment and tools used were generally suitable. Some decisions such as the choice of programming language were already decided due to the requirements. However, even if they had not been, C++ is an excellent language for writing a driver, possessing the power and versatility of C with many of the conveniences offered by higher level languages. The combination of object-oriented structure and direct hardware level access was extremely suitable for this project.

Occasionally variants of C can be unwieldy due to the need to constantly recompile, avoid linker issues and ensure the correct headers and includes are present. The addition of *CMake* to the project minimised these issues. Other software used included the static analysis tools *cppcheck* and *scan-build*, both of which were easy to use and provided additional confidence that the driver was functioning as intended. Although it was a subjective choice, the use of the *MacVim* editor combined with a C++ semantic-completion plugin made development highly efficient, as well as allowing the same editor to be used for both the source code and the documentation.

Doxygen was extremely useful for creating some of the diagrams used in this dissertation, as well as providing the ability to provide a rich source of documentation directly within the code. *Player* was generally easy to work with, despite lacking documentation in some areas. This was mitigated to an extent by the clarity and availability of the source code.

In terms of operating systems; although *Player* development focuses on Linux, it was easy to compile *Player* and its drivers on OS X. Support for peripherals such as the video capture card described in the previous chapter was actually superior in OS X compared to Linux as an external driver was not required. This balanced out with the small amount of additional difficulty installing and configuring *Player* for development on OS X rather than Linux, mostly limited to the need to install *gcc*.

The Sony camera hardware was relatively well documented, with a few surprising omissions (such as detailed information on what the *Pan-tiltMode* inquiry actually does) and minor differences between the hardware limits listed in the manual and those that were recorded during testing.

5.2.2 Design Evaluation

The design of the modified driver is compact and relatively simple. In retrospect, the decision to take the most direct approach and minimise unnecessary abstraction was a good one, all functionality is neatly arranged in a single source file with easy access to variables. Maintainability was a key concern and the aim of the design was to make it easy to construct a mental model of the overall driver architecture, even without driver or *Player* development experience.

The decision to keep everything contained in a single file was a controversial one and was reconsidered several times during development. A fully object-oriented system using multiple classes may have resulted in a more extensible and reusable system. Similarly, separating constants and data structure definitions into a header file would have allowed for cleaner separation between structure and implementation. Despite this, the fragmentation caused by taking either of these approaches would have been a significant issue.

Another design decision was to ignore the buffering functionality within the hardware. This was perhaps the most controversial design decision and represented a major point of convergence between the *sonyevid30* driver and *sony-ptz*, in which the two drivers were no longer interchangeable in all cases. Conversely, the decision to remove as many stored constants as possible and replace this with a dynamic system yielded many advantages and no obvious issues. This was an easy design decision and evidence from development and testing reinforced the benefits of this approach.

Two designs were considered for the calibration method, although the Design chapter only describes the final design. The initial design provided an elegant solution involving continuously incrementing or decrementing the pan and tilt test positions until the current position stopped changing, indicating that a hardware limit had been reached.

In practice this could not be implemented reliably, so the current design was chosen which directly reads the status of the camera and is more reliable. In the future it may be worth revisiting the calibration implementation to make it as accurate and quick as possible.

5.2.3 Future Additions

Whilst the driver does provide a lot of functionality in its current state, there are a number of areas future work could focus on. The calibration function works, but it could be refactored to improve consistency between models. There is currently some hardware-specific code which allows calibration to work seamlessly between different models of camera. In a future version it would be sensible to simplify calibration so that it works exactly the same way on all supported hardware, this would be a more robust solution.

Another area where work could be conducted is in supporting multiple cameras concurrently. The *VISCA* protocol supports this and the hardware can be daisy-chained with up to seven cameras running concurrently from a single controller. This is partially supported in the current driver (just as it was in *sonyevid30*), with an optional parameter which accepts the ID of the camera to send a command or inquiry to. With further testing and refactoring, this functionality could be completed and the driver would be able to control an entire array of cameras.

In a similar vein; the driver technically supports a greater range of *VISCA* cameras than the three that were tested, although with less than optimal control because unknown cameras are treated

as an *EVI-D30*. It would be trivial to extend the driver to support additional *VISCA* hardware as all commands and inquiries currently implemented would continue to function correctly.

As discussed in previous chapters, the zoom control in the driver is not currently as advanced as the support for pan and tilt. The driver correctly supports variable zoom speeds, fulfilling a key design objective. Despite this, there is room for further development in this area.

The zoom functionality does not yet allow for zooming to an arbitrary position at an arbitrary speed, this is theoretically possible but would further development work to implement. Another area related to zoom is the ability for the camera to return the current zoom position in a useful format.

In the current version of the driver zoom positions are returned in camera units, although an experimental feature was added at the end of the final iteration which returns an approximate camera zoom position as a percentage of the total zoom. It would be useful to implement this functionality for zoom speeds too, rather than using camera units. For many use cases it would be acceptable to continually increment or decrement the zoom level in response to external input, however it would be preferable if the driver allowed for more granular control of the zoom functionality.

A further suggestion made during a demonstration was implementing logic to handle motor 'drift'. It is possible that the hardware may degrade over time and the calibration data gained initially may no longer be accurate. The driver does not currently include provisions for this, but it would be simple to add an additional field containing a timestamp to the calibration data. This could then trigger re-calibration after a set period to ensure the calibration data used remains as accurate as possible.

Another suggestion was to create an interactive client application which uses the driver to interact with a camera and present the resulting data visually. This was an excellent idea but unfortunately was not possible due to time constraints. In order to implement this in the future, an application could be created which provides a GUI with a `C++` *Player* client backend which interfaces with the driver.

5.3 Conclusion

Throughout this dissertation, we have discussed the evolution of a system which improves support for Sony EVI cameras in *Player*, from background research and prototyping to the creation of a fully-functional, compact driver which can be used for many real-world uses. The driver met all of its technical objectives, although as with most software, some areas leave room for improvement. The project itself was an intense period of design, development and reflection which ultimately resulted in a highly polished, useful piece of software.

The modified driver produced is not perfect, but it is definitely suitable for everyday use. It does not attempt to meet every possible use case, but it does provide full control of pan, tilt and zoom in a well-documented and intuitive way. In the future, the driver could be extended to support new hardware, features and use cases.

As it stands, the foundations have been laid for a robust and useful hardware driver that represents an improvement to the current state of the art and offers significant future potential.

Appendix A

Technical References

This appendix contains several pages from the technical manual for the Sony *EVI-D70* camera, including a sample of *VISCA* commands and the pan-tilt status table featured in the Implementation & Refactoring chapter.

These pages are included for reference purposes only and do not imply any copyright claim on behalf of the author or implied endorsement of this project by Sony.

Command List

VISCA[1)] RS-232C/RS-422 Commands

Use of RS-232C/RS-422 control software which has been developed based upon this command list may cause malfunction or damage to hardware and software. Sony Corporation is not liable for any such damage.

Overview of VISCA

In VISCA, the side outputting commands, for example, a computer, is called the controller, while the side receiving the commands, such as an EVI-D70/P, is called the peripheral device. The EVI-D70/P serves as a peripheral device in VISCA. In VISCA, up to seven peripheral devices like the EVI-D70/P can be connected to one controller using communication conforming to the RS-232C/RS-422 standard. The parameters of RS-232C/422 are as follows.

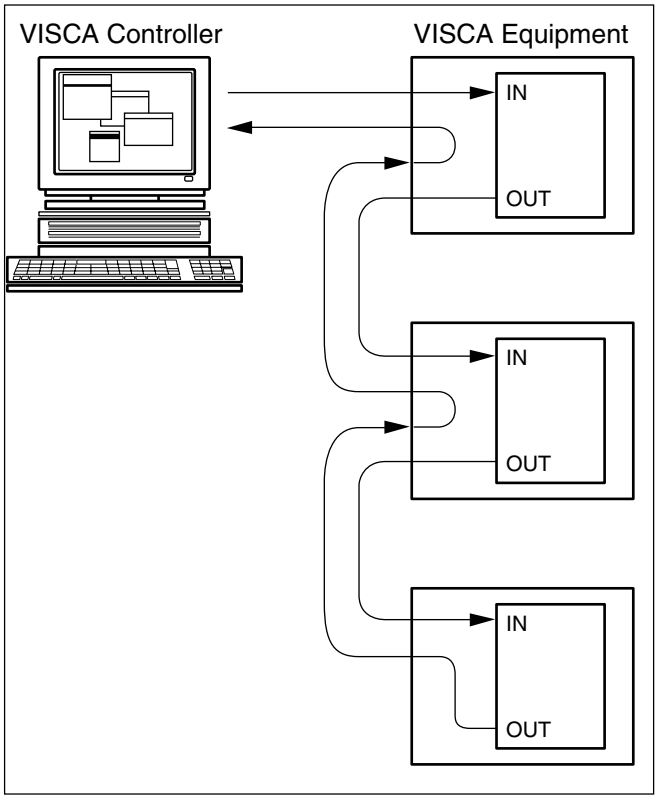
- Communication speed: 9600 bps/38400 bps
- Data bits : 8
- Start bit : 1
- Stop bit : 1
- Non parity

Flow control using XON/XOFF and RTS/CTS, etc., is not supported.

Peripheral devices are connected in a daisy chain. As shown in Fig. 1, the actual internal connection is a one-direction ring, so that messages return to the controller via the peripheral devices. The devices on the network are assigned addresses.

The address of the controller is fixed at 0. The addresses of the peripheral devices are 1, 2, 3 ... in order, starting from the one nearest the controller. The address of the peripheral device is set by sending address commands during the initialization of the network.
The VISCA devices each have a VISCA IN and VISCA OUT connector.
Set the DTR input (the S output of the controller) of VISCA IN to H when controlling VISCA equipment from the controller.

Fig. 1 VISCA network configuration



1) VISCA is a protocol which controls consumer camcorders developed by Sony. “VISCA” is a trademark of Sony Corporation.

VISCA Communication Specifications

VISCA packet structure

The basic unit of VISCA communication is called a packet (Fig. 2). The first byte of the packet is called the header and comprises the sender's and receiver's addresses. For example, the header of the packet sent to the EVI-D70/P assigned address 1 from the controller (address 0) is hexadecimal 81H. The packet sent to the EVI-D70/P assigned address 2 is 82H. In the command list, as the header is 8X, input the address of the EVI-D70/P at X. The header of the reply packet from the EVI-D70/P assigned address 1 is 90H. The packet from the EVI-D70/P assigned address 2 is A0H.
Some of the commands for setting EVI-D70/P units can be sent to all devices at one time (broadcast). In the case of broadcast, the header should be hexadecimal 88H.
When the terminator is FFH, it signifies the end of the packet.

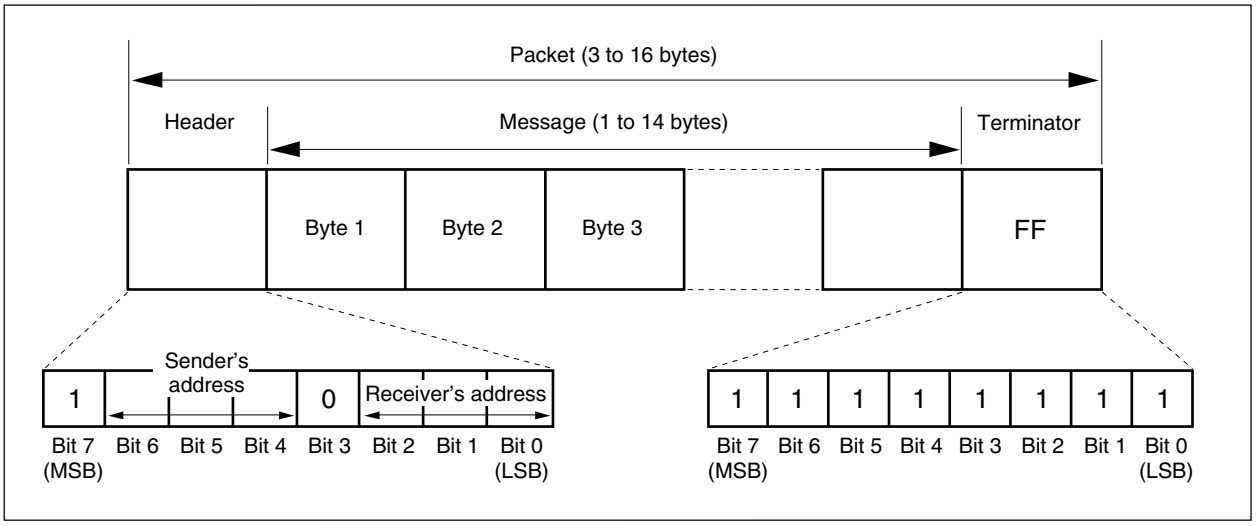


Fig. 2 Packet structure

Note

Fig. 2 shows the packet structure, while Fig. 3 shows the actual waveform. Data flow will take place with the LSB first.

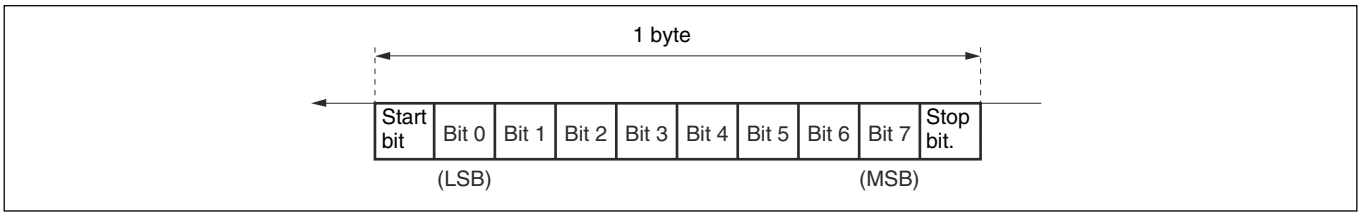
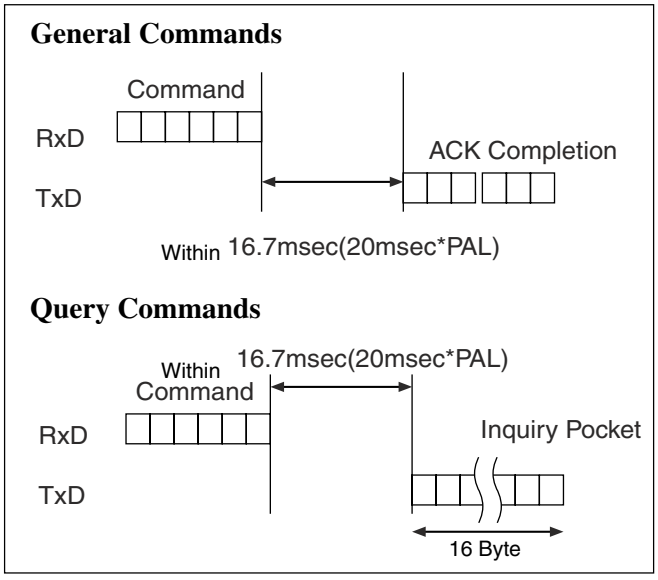


Fig. 3 Actual waveform for 1 byte.

Timing Chart

As VISCA Command processing can only be carried out one time in a Vertical cycle, it takes the maximum 1V cycle time for an ACK/Completion to be returned. If the Command ACK/Completion communication time can be cut shorter than the1V cycle time, then every 1V cycle can receive a Command.
From this point, if 2 or more commands in a row are to be sent, wait for the first command (for normal commands, an ACK or an error message, for query commands, an Inquiry Packet) to be carried out before sending the next one.



Command and inquiry

● **Command**
Sends operational commands to the EVI-D70/P.

● **Inquiry**
Used for inquiring about the current state of the EVI-D70/P.

	Command Packet	Note
Inquiry	8X QQ RR ... FF	QQ[1] = Command/Inquiry, RR[2] = category code

[1] QQ = 01 (Command), 09 (Inquiry)
[2] RR = 00 (Interface), 04 (camera 1), 06 (Pan/Tilter)

X = 1 to 7: EVI-D70/P address

Responses for commands and inquiries

● **ACK message**
Returned by the EVI-D70/P when it receives a command. No ACK message is returned for inquiries.

● **Completion message**
Returned by the EVI-D70/P when execution of commands or inquiries is completed. In the case of inquiry commands, it will contain reply data for the inquiry after the 3rd byte of the packet. If the ACK message is omitted, the socket number will contain a 0.

	Reply Packet	Note
Ack	X0 4Y FF	Y = socket number
Completion (commands)	X0 5Y FF	Y = socket number
Completion (Inquiries)	X0 5Y ... FF	Y = socket number

X = 9 to F: EVI-D70/P address + 8

● **Error message**
When a command or inquiry command could not be executed or failed, an error message is returned instead of the completion message.

Error Packet	Description
X0 6Y 01 FF	Message length error (>14 bytes)
X0 6Y 02 FF	Syntax Error
X0 6Y 03 FF	Command buffer full
X0 6Y 04 FF	Command cancelled
X0 6Y 05 FF	No socket (to be cancelled)
X0 6Y 41 FF	Command not executable

X = 9 to F: EVI-D70/P address + 8, Y = socket number

Socket number

When command messages are sent to the EVI-D70/P, it is normal to send the next command message after waiting for the completion message or error message to return. However to deal with advanced uses, the EVI-D70/P has two buffers (memories) for commands, so that up to two commands including the commands currently being executed can be received. When the EVI-D70/P receives commands, it notifies the sender which command buffer was used using the socket number of the ACK message.
As the completion message or error message also has a socket number, it indicates which command has ended.
Even when two command buffers are being used at any one time, an EVI-D70/P management command and some inquiry messages can be executed.
The ACK message is not returned for these commands and inquiries, and only the completion message of socket number 0 is returned.

Command execution cancel

To cancel a command which has already been sent, send the Cancel command as the next command. To cancel one of any two commands which have been sent, use the cancel message.

	Cancel Packet	Note
Cancel	8X 2Y FF	Y = socket number

X = 1 to 7: EVI-D70/P address, Y = socket number

The Command canceled error message will be returned for this command, but this is not a fault. It indicates that the command has been canceled.

VISCA Command/ACK Protocol

Command	Command Message	Reply Message	Comments
General Command	81 01 04 38 02 FF (Example)	90 41 FF (ACK)+90 51 FF (Completion) 90 42 FF 90 52 FF	Returns ACK when a command has been accepted, and Completion when a command has been executed.
	81 01 04 38 FF (Example)	90 60 02 FF (Syntax Error)	Accepted a command which is not supported or a command lacking parameters.
	81 01 04 38 02 FF (Example)	90 60 03 FF (Command Buffer Full)	There are two commands currently being executed, and the command could not be accepted.
	81 01 04 08 02 FF (Example)	90 61 41 FF (Command Not Executable) 90 62 41FF	Could not execute the command in the current mode.
Inquiry Command	81 09 04 38 FF (Example)	90 50 02 FF (Completion)	ACK is not returned for the inquiry command.
	81 09 05 38 FF (Example)	90 60 02 FF (Syntax Error)	Accepted an incompatible command.
Address Set	88 30 01 FF	88 30 02 FF	Returned the device address to +1.
IF_Clear(Broadcast)	88 01 00 01 FF	88 01 00 01 FF	Returned the same command.
IF_Clear (For x)	8x 01 00 01 FF	z0 50 FF (Completion)	ACK is not returned for this command.
Command Cancel	8x 2y FF	z0 6y 04 FF (Command Canceled)	Returned when the command of the socket specified is canceled. Completion for the command canceled is not returned.
		z0 6y 05 FF (No Socket)	Returned when the command of the specified socket has already been completed or when the socket number specified is wrong.

VISCA Camera-Issued Messages

ACK/Completion Messages

	Command Messages	Comments
ACK	z0 4y FF (y:Socket No.)	Returned when the command is accepted.
Completion	z0 5y FF (y:Socket No.)	Returned when the command has been executed.

z = Device address + 8

Error Messages

	Command Messages	Comments
Syntax Error	z0 60 02 FF	Returned when the command format is different or when a command with illegal command parameters is accepted.
Command Buffer Full	z0 60 03 FF	Indicates that two sockets are already being used (executing two commands) and the command could not be accepted when received.
Command Canceled	z0 6y 04 FF (y:Socket No.)	Returned when a command which is being executed in a socket specified by the cancel command is canceled. The completion message for the command is not returned.
No Socket	z0 6y 05 FF (y:Socket No.)	Returned when no command is executed in a socket specified by the cancel command, or when an invalid socket number is specified.
Command Not Executable	z0 6y 41 FF (y:Socket No.)	Returned when a command cannot be executed due to current conditions. For example, when commands controlling the focus manually are received during auto focus.

Network Change Message

	Command Message	Comments
Network Change	z0 38 FF	Issued when power is being routed to the camera, or when the VISCA device is connected to or disconnected from the VISCA RS-232C/RS-422 OUT connector used for communication.

EVI-D70/P Commands

EVI-D70/P Command List (1/4)

Command Set	Command	Command Packet	Comments
AddressSet	Broadcast	88 30 01 FF	Address setting
IF_Clear	Broadcast	88 01 00 01 FF	I/F Clear
CommandCancel		8x 2p FF	p: Socket No.(=1or2)
CAM_Power	On	8x 01 04 00 02 FF	Power ON/OFF
	Off	8x 01 04 00 03 FF	
CAM_AutoPowerOff	Direct	8x 01 04 40 0p 0q 0r 0s FF	Auto Power Off pqrs: Power Off Timer 0000 (Timer Off) to FFFF (65535min) Initial value: 0000 The power automatically turns off if the camera does not receive any VISCA commands or any signals from the Remote Commander for the duration you set in the timer.
CAM_NightPoweroff	Direct	8x 01 04 41 0p 0q 0r 0s FF	Pqrs: Power Off Timer 0000~FFFF (Units: Min.) A setting of 0 (zero min.) is equivalent to OFF, and the smallest value that can be set is 1 min. When the Day/Night function is in effect and the Night setting has been made, if an operation is not attempted via either a VISCA command or the remote controller, the unit will continue to operate for the time set in the timer, and will then shut off automatically.
CAM_Zoom	Stop	8x 01 04 07 00 FF	
	Tele(Standard)	8x 01 04 07 02 FF	
	Wide(Standard)	8x 01 04 07 03 FF	
	Tele(Variable)	8x 01 04 07 2p FF	p=0 (Low) to 7 (High)
	Wide(Variable)	8x 01 04 07 3p FF	
	Direct	8x 01 04 47 0p 0q 0r 0s FF	pqrs: Zoom Position
CAM_DZoom	On	8x 01 04 06 02 FF	Digital zoom ON/OFF
	Off	8x 01 04 06 03 FF	
	Combine Mode	8x 01 04 36 00 FF	Optical/Digital Zoom Combined
	Separate Mode	8x 01 04 36 01 FF	Optical/Digital Zoom Separate
	Stop	8x 01 04 06 00 FF	
	Tele(Variable)	8x 01 04 06 2p FF	p=0 (Low) to 7 (High)
	Wide(Variable)	8x 01 04 06 3p FF	
	x1/Max	8x 01 04 06 10 FF	x1/MAX Magnification Switchover
	Direct	8x 01 04 46 00 00 0p 0q FF	pq: D-Zoom Position
CAM_Focus	Stop	8x 01 04 08 00 FF	
	Far(Standard)	8x 01 04 08 02 FF	
	Near(Standard)	8x 01 04 08 03 FF	
	Far(Variable)	8x 01 04 08 2p FF	p=0 (Low) to 7 (High)
	Near(Variable)	8x 01 04 08 3p FF	
	Direct	8x 01 04 48 0p 0q 0r 0s FF	pqrs: Focus Position
	Auto Focus	8x 01 04 38 02 FF	AF ON/OFF
	Manual Focus	8x 01 04 38 03 FF	
	Auto/Manual	8x 01 04 38 10 FF	
	One Push Trigger	8x 01 04 18 01 FF	One Push AF Trigger
	Infinity	8x 01 04 18 02 FF	Forced infinity
	Near Limit	8x 01 04 28 0p 0q 0r 0s FF	pqrs: Focus Near Limit Position

EVI-D70/P Inquiry Command List (1/2)

Inquiry Command	Command Packet	Inquiry Packet	Comments
CAM_PowerInq	8x 09 04 00 FF	y0 50 02 FF	On
		y0 50 03 FF	Off
CAM_AutoPowerOffInq	8x 09 04 40 FF	y0 50 0p 0q 0r 0s FF	pqrs: PowerOff Timer
CAM_NightPowerOff Inq	8x 09 04 41 FF	y0 50 0p 0q 0r 0s FF	pqrs: NightPowerOff Timer
CAM_ZoomPosInq	8x 09 04 47 FF	y0 50 0p 0q 0r 0s FF	pqrs: Zoom Position
CAM_DZoomModeInq	8x 09 04 06 FF	y0 50 02 FF	D-Zoom On
		y0 50 03 FF	D-Zoom Off
CAM_DZoomC/SModeInq	8x 09 04 36 FF	y0 50 00 FF	Combine Mode
		y0 50 01 FF	Separate Mode
CAM_DZoomPosInq	8x 09 04 46 FF	y0 50 00 00 0p 0q FF	pq: D-Zoom Position
CAM_FocusModeInq	8x 09 04 38 FF	y0 50 02 FF	Auto Focus
		y0 50 03 FF	Manual Focus
CAM_FocusPosInq	8x 09 04 48 FF	y0 50 0p 0q 0r 0s FF	pqrs: Focus Position
CAM_FocusNearLimitInq	8x 09 04 28 FF	y0 50 0p 0q 0r 0s FF	pqrs: Focus Near Limit Position
CAM_AFSensitivityInq	8x 09 04 58 FF	y0 50 02 FF	AF Sensitivity Normal
		y0 50 03 FF	AF Sensitivity Low
CAM_AFModeInq	8x 09 04 57 FF	y0 50 00 FF	Normal AF
		y0 50 01 FF	Interval AF
		y0 50 02 FF	Zoom Trigger AF
CAM_AFTimeSettingInq	8x 09 04 27 FF	y0 50 0p 0q 0r 0s FF	pq: Movement Time, rs: Interval
CAM_WBModeInq	8x 09 04 35 FF	y0 50 00 FF	Auto
		y0 50 01 FF	In Door
		y0 50 02 FF	Out Door
		y0 50 03 FF	One Push WB
		y0 50 04 FF	ATW
		y0 50 05 FF	Manual
CAM_RGainInq	8x 09 04 43 FF	y0 50 00 00 0p 0q FF	pq: R Gain
CAM_BGainInq	8x 09 04 44 FF	y0 50 00 00 0p 0q FF	pq: B Gain
CAM_AEModeInq	8x 09 04 39 FF	y0 50 00 FF	Full Auto
		y0 50 03 FF	Manual
		y0 50 0A FF	Shutter Priority
		y0 50 0B FF	Iris Priority
		y0 50 0D FF	Bright
CAM_SlowShutterModeInq	8x 09 04 5A FF	y0 50 02 FF	Auto
		y0 50 03 FF	Manual
CAM_ShutterPosInq	8x 09 04 4A FF	y0 50 00 00 0p 0q FF	pq: Shutter Position
CAM_IrisPosInq	8x 09 04 4B FF	y0 50 00 00 0p 0q FF	pq: Iris Position
CAM_GainPosInq	8x 09 04 4C FF	y0 50 00 00 0p 0q FF	pq: Gain Position
CAM_BrightPosInq	8x 09 04 4D FF	y0 50 00 00 0p 0q FF	pq: Bright Position
CAM_ExpCompModeInq	8x 09 04 3E FF	y0 50 02 FF	On
		y0 50 03 FF	Off
CAM_ExpCompPosInq	8x 09 04 4E FF	y0 50 00 00 0p 0q FF	pq: ExpComp Position
CAM_BacklightModeInq	8x 09 04 33 FF	y0 50 02 FF	On
		y0 50 03 FF	Off
CAM_SpotAEModeInq	8x 09 04 59 FF	y0 50 02 FF	On
		y0 50 03 FF	Off
CAM_SpotAEPosInq	8x 09 04 29 FF	y0 50 0p 0q 0r 0s FF	pq: X position, rs: Y position
CAM_ApertureInq	8x 09 04 42 FF	y0 50 00 00 0p 0q FF	pq: Aperture Gain

When using Digital Zoom Separate Mode (CAM_DZoomPos Inq)

Magnification	Position Data
×1	00
×2	80
×3	AA
×4	C0
×5	CC
×6	D5
×7	DB
×8	E0
×9	E3
×10	E6
×11	E8
×12	EB

Lens Control

Zoom Position	0000 to 4000 to 7AC0 Wide end Optical Tele end Digital Tele end	
Focus Position	1000 to C000 Far end Near end	
Focus Near Limit	1000: Over Inf 2000: 8.0 m 3000: 3.5 m 4000: 2.0 m 5000: 1.4 m 6000: 1 m 7000: 80 cm 8000: 29 cm 9000: 10 cm A000: 4.7 cm B000: 2.3 cm C000: 1.0 cm	As the distance on the left will differ due to temperature characteristics, etc., use as approximate values. * The lower 1 byte is fixed at 00.

Pan/Tilt Position (Reference values)

		Pan Position Data	Tilt Position Data (Image Flip: OFF)	Tilt Position Data (Image Flip: ON)
Angle	+170°	08DB	–	–
	+150°	07DB	–	–
	+90°	04B0	04B0	–
	+30°	0190	0190	0190
	+10°	0085	0085	0085
	+0.075°	0001	0001	0001
	0°	0000	0000	0000
	–0.075°	FFFF	FFFF	FFFF
	–10°	FF7B	FF7B	FF7B
	–30°	FE70	FE70	FE70
	–90°	FB50	–	FB50
	–150°	F830	–	–
	–170°	F725	–	–

A + indicates a pan to the right and a tilt upward on the monitor screen.
A – indicates a pan to the left and a tilt downward on the monitor screen.

Others

R,B gain	00~FF
Aperture	00~0F

Title Setting

Vposition	00 to 0A	
Hposition	00 to 17	
Blink	00: Does not blink	
	01: Blinks	
Color	00	White
	01	Yellow
	02	Violet
	03	Red
	04	Cyan
	05	Green
	06	Blue

Pan/Tilt Status Code List

P	Q	R	S	
0 - - -	- - - -	0 - - -	- - - 1	A Pan movement all the way to the left
0 - - -	- - - -	0 - - -	- - 1 -	A Pan movement all the way to the right
0 - - -	- - - -	0 - - -	- 1 - -	A Tilt movement all the way up
0 - - -	- - - -	0 - - -	1 - - -	A Tilt movement all the way down
0 - - -	- - - -	- - 0 0	- - - -	Pan movement is correct
0 - - -	- - - -	- - 0 1	- - - -	Pan position cannot be detected
0 - - -	- - - -	- - 1 0	- - - -	The Pan mechanism is abnormal
0 - - -	- - 0 0	0 - - -	- - - -	The Tilt movement is correct
0 - - -	- - 0 1	0 - - -	- - - -	The Tilt position cannot be detected
0 - - -	- - 1 0	0 - - -	- - - -	The Tilt mechanism is abnormal
0 - - -	0 0 - -	0 - - -	- - - -	No movement instructions
0 - - -	0 1 - -	0 - - -	- - - -	In the midst of a Pan/Tilt
0 - - -	1 0 - -	0 - - -	- - - -	Pan/Tilt completed
0 - - -	1 1 - -	0 - - -	- - - -	Pan/Tilt failed
0 - 0 0	- - - -	0 - - -	- - - -	Not initialized
0 - 0 1	- - - -	0 - - -	- - - -	Initializing
0 - 1 0	- - - -	0 - - -	- - - -	Initialization completed
0 - 1 1	- - - -	0 - - -	- - - -	Initialization failed

(- : optional)

Pan/tilt position

Pan position

Cameras	Parameters
EVI-D30/D31	FC90h (–100 degrees) to 0370h (+100 degrees)
EVI-D70/P	FACBh (–100 degrees) to 0535h (+100 degrees)

Tilt position

Cameras	Parameters
EVI-D30/D31	FED4h (–25 degrees) to 012Ch (+25 degrees)
EVI-D70/P	FEB3h (–25 degrees) to 014Dh (+25 degrees)

Translation of commands

Accepting parameters	Translation
Pan position	Multiplies received parameters by 50/33
Tilt position	Multiplies received parameters by 50/45

D30		D70	
Pan/Tilt		Pan	Tilt
FC90h	→	FACBh	–
…		…	–
FED4h	→	FE3Ah	FEB3h
…		…	…
FFFDh	→	FFFCh	FFFDh
FFFEh	→	FFFDh	FFFEh
FFFFh	→	FFFFh	FFFFh
0000h	→	0000h	0000h
0001h	→	0001h	0001h
0002h	→	0003h	0002h
0003h	→	0004h	0003h
0004h	→	0006h	0004h
0005h	→	0007h	0005h
0006h	→	0009h	0006h
0007h	→	000Ah	0007h
0008h	→	000Ch	0008h
0009h	→	000Dh	000Ah
000Ah	→	000Fh	000Bh
000Bh	→	0010h	000Ch
…		…	…
012Ch	→	01C6h	014Dh
…		…	–
0370h	→	0535h	–

Translation when answering inquiry commands

The camera sends back values that are calculated by inverse conversion performed when the camera accepts commands.

Note

Repetitive use of the Absolute position command may increase the variance that is accumulated from translation.

Example

When you execute the Relative position command 88 times, one step after another to turn the camera to the right side:

Cameras	Results
EVI-D30/D31	Turns to the right side by 10 degrees.
EVI-D70/P whose D30/D31 mode is ON	Turns to the right side by 6.6 degrees.

For Absolute Position commands, the permissible range for drive settings are the same as those for the EVI-D30/D31.
Pan direction: –100 degrees to + 100 degrees.
Tilt direction: –25 degrees to + 25 degrees.

Appendix B

Source Code

This appendix contains full code listings for the hardware calibration function and the demonstration function created for the two project demonstrations.

```
/* This function tests various values for pan, tilt and zoom in order to acquire accurate minimum and maximum values that properly reflect the connected hardware */
int SonyPTZ::CalibrateCamera()
{
    std::cout << "Calibrating..." << std::endl;

    /* Query camera for maximum speed settings, this sets maxPanSpeed and maxTiltSpeed in the sony_ptz_cam_t struct accordingly */
    GetMaximumSpeeds();

    /* Increment varies by model for the quickest and most accurate calibration */
    int testIncrement;

    if (ptzCamConfig.modelID == D70_MODEL_HEX)
    {
        testIncrement = 1;
    }

    else
    {
        testIncrement = 10;
    }

    /* Set initial test positions for pan and tilt */
    int testPositionPan = 0x0600; // Value discovered through experimentation, this should ideally be a value close to an upper or lower limit but still within range for all supported cameras
    int testPositionTilt = 0x3E8; // Set test position for tilt to a value close to the expected maximum value but still within range for every supported camera

    /* Move to the start positions for testing pan and tilt */
    SetAbsolutePanTiltPosition(testPositionPan, testPositionTilt, true); // Passing a value of true means the hardware will wait until the command has completed before processing any new movement commands

    bool stopPanning = false;
    bool stopTilting = false;

    bool incrementPan = true;
    bool incrementTilt = true;

    while (true)
    {
        SetAbsolutePanTiltPosition(testPositionPan, testPositionTilt, true);
        GetPanTiltStatus(); // This updates the state of the panTiltStatus variable

        /* The hexadecimal values here relate to the pan/tilter status codes provided in the technical manuals */
        /* VISCA is most-significant-bit first so 0x02 is equivalent to an S value of --1-, see pg. 51 EVI-D70P manual */
        if (panTiltStatus[3] == 0x02 || panTiltStatus[3] == 0x06)
        {
            ptzCamConfig.maxPanAngleCU = testPositionPan;
            printf("Maximum Pan Angle: 0x%2X\n", ptzCamConfig.maxPanAngleCU);
            incrementPan = false;

            if (ptzCamConfig.modelID == D70_MODEL_HEX)
            {
                testPositionPan = 0xF800;
```

```
            }
        }

        if (panTiltStatus[3] == 0x04 || panTiltStatus[3] == 0x06)
        {
            ptzCamConfig.maxTiltAngleCU = testPositionTilt;
            printf("Maximum Tilt Angle: 0x%2X\n", ptzCamConfig.maxTiltAngleCU);
            incrementTilt = false;

            /* Optimisation for the D70 */
            if (ptzCamConfig.modelID == D70_MODEL_HEX)
            {
                testPositionTilt = 0xFEDD;
            }
        }

        if ( (panTiltStatus[3] == 0x01 || panTiltStatus[3] == 0x09) && !stopPanning)
        {
            ptzCamConfig.minPanAngleCU = testPositionPan;
            printf("Minimum Pan Angle: 0x%2X\n", ptzCamConfig.minPanAngleCU);
            stopPanning = true;
        }

        if ( (panTiltStatus[3] == 0x08 || panTiltStatus[3] == 0x09) && !stopTilting)
        {
            ptzCamConfig.minTiltAngleCU = testPositionTilt;
            printf("Minimum Tilt Angle: 0x%2X\n", ptzCamConfig.minTiltAngleCU);
            stopTilting = true;
        }

        if (stopPanning && stopTilting)
        {
            break;
        }

        if (incrementPan && !stopPanning)
        {
            testPositionPan+= testIncrement;
        }

        else if (!incrementPan && !stopPanning)
        {
            testPositionPan-= testIncrement;
        }

        if (incrementTilt && !stopTilting)
        {
            testPositionTilt+= testIncrement;
        }

        else if (!incrementTilt && !stopTilting)
        {
            testPositionTilt-= testIncrement;
        }
    }

    /* Print the values discovered by calibration so the user can copy these into the config
     ↪  file for future use */
```

```
108     std::cout << "Calibrated values are as follows (you may copy these into the config file):
        ↪  ";
109
110     /* Variables to store configuration data */
111     std::string configFileContents;
112     std::string configFileLine;
113     std::string calibrationData = std::string(ptzCamConfig.modelName) + " ";
114
115     /* Build a tuple for minPanAngleCU, maxPanAngleCU, minTiltAngleCU, maxTiltAngleCU,
        ↪  maxPanSpeed and maxTiltSpeed for potential insertion into the file */
116     calibrationData +=  "[";
117     calibrationData += TOSTR(ptzCamConfig.minPanAngleCU,  " "); // Use the macro function
        ↪  defined at the top of this file
118     calibrationData += TOSTR(ptzCamConfig.maxPanAngleCU,  " ");
119     calibrationData += TOSTR(ptzCamConfig.minTiltAngleCU, " ");
120     calibrationData += TOSTR(ptzCamConfig.maxTiltAngleCU, " ");
121     calibrationData += TOSTR(ptzCamConfig.maxPanSpeed,    " ");
122     calibrationData += TOSTR(ptzCamConfig.maxTiltSpeed,    "");
123     calibrationData += "]";
124
125     std::cout << calibrationData << std::endl;
126
127     SetAbsolutePanTiltPosition(CENTER, CENTER, true); // Move camera back to the centre
128
129     return 0;
130 }
```

```
1  /* Demo method used to test basic pan, tilt and zoom functionality */
2  void SonyPTZ::PanTiltDemo()
3  {
4      std::cout << std::endl << "Hit enter to start the hardware demo function..." << std::endl
       ↪  << std::endl;
5      std::cin.ignore();
6
7      /* Demonstrate speed control functionality */
8      std::cout << "Setting speed..." << std::endl;
9      SetAbsolutePanTiltSpeed(0x12, 0x12);
10
11     /* Basic movement */
12     std::cout << std::endl << "Minimum Pan..." << std::endl;
13     SetAbsolutePanTiltPosition(ptzCamConfig.minPanAngleDegrees, CENTER, true);
14     std::cout << "Maximum Pan..." << std::endl;
15     SetAbsolutePanTiltPosition(ptzCamConfig.maxPanAngleDegrees, CENTER, true);
16     std::cout << "Minimum Tilt..." << std::endl;
17     SetAbsolutePanTiltPosition(CENTER, ptzCamConfig.minTiltAngleDegrees, true);
18     std::cout << "Maximum Tilt..." << std::endl;
19     SetAbsolutePanTiltPosition(CENTER, ptzCamConfig.maxTiltAngleDegrees, true);
20
21     std::cin.ignore();
22
23     /* Combined movement */
24     std::cout << "Maximum Pan, Minimum Tilt..." << std::endl;
25     SetAbsolutePanTiltPosition(ptzCamConfig.maxPanAngleDegrees,
       ↪  ptzCamConfig.minTiltAngleDegrees, true);
26     std::cout << "Minimum Pan, Minimum Tilt..." << std::endl;
27     SetAbsolutePanTiltPosition(ptzCamConfig.minPanAngleDegrees,
       ↪  ptzCamConfig.minTiltAngleDegrees, true);
28     std::cout << "Maximum Pan, Maximum Tilt..." << std::endl;
29     SetAbsolutePanTiltPosition(ptzCamConfig.maxPanAngleDegrees,
       ↪  ptzCamConfig.maxTiltAngleDegrees, true);
```

```
30      std::cout << std::endl;
31
32      /* Demonstrate how the camera can accurately report current positions */
33      short currentPanPos, currentTiltPos;
34
35      if (GetAbsolutePanTiltPosition(&currentPanPos, &currentTiltPos) != 0)
36      {
37          perror("SonyPTZ::PanTiltDemo: Error getting current pan / tilt position,
            ↪  terminating");
38          exit(1);
39      }
40
41      std::cout << "Requesting position data from camera..." << std::endl;
42      std::cout << "Current Pan Position: "  << std::dec << currentPanPos << " degrees" <<
        ↪  std::endl;
43      std::cout << "Current Tilt Position: " << std::dec << currentTiltPos << " degrees" <<
        ↪  std::endl;
44
45      std::cin.ignore();
46
47      /* The camera will threshold values above or below the minimum supported by the hardware
        ↪  */
48      std::cout << "Thresholding Invalid Values..." << std::endl;
49      SetAbsolutePanTiltPosition(ptzCamConfig.maxPanAngleDegrees + 100,
        ↪  ptzCamConfig.minTiltAngleDegrees - 100, true);
50      std::cin.ignore();
51
52      SetAbsolutePanTiltPosition(CENTER, CENTER, true);
53
54      /* Demonstrate Zoom Functionality */
55      short currentZoomPos;
56
57      std::cout << "Telescopic Zoom..." << std::endl;
58      SetTeleZoom();
59      std::cin.ignore();
60
61      GetAbsoluteZoomPosition(&currentZoomPos);
62      std::cout << "Zoom: " << (currentZoomPos / ptzCamConfig.maxZoomCU) * 100 << "%" <<
        ↪  std::endl;
63      std::cin.ignore();
64
65      std::cout << "Wide Angle Zoom..." << std::endl;
66      SetWideZoom();
67      std::cin.ignore();
68      GetAbsoluteZoomPosition(&currentZoomPos);
69      std::cout << "Zoom: " << (currentZoomPos / ptzCamConfig.maxZoomCU) * 100 << "%" <<
        ↪  std::endl;
70      std::cin.ignore();
71
72      std::cout << "Zoom to arbitrary level..." << std::endl;
73      SetAbsoluteZoomPosition(ptzCamConfig.maxZoomCU * 0.45);
74      std::cin.ignore();
75      GetAbsoluteZoomPosition(&currentZoomPos);
76      std::cout << "Zoom Position: " << round( (currentZoomPos / (float) ptzCamConfig.maxZoomCU)
        ↪  * 100) << "%" << std::endl;
77
78      std::cout << std::endl << "Wide Zoom with speed..." << std::endl;
79      SetWideZoom(1);
80      GetAbsoluteZoomPosition(&currentZoomPos);
```

```
81      std::cin.ignore();
82
83      SetAbsolutePanTiltSpeed(ptzCamConfig.maxPanSpeed, ptzCamConfig.maxTiltSpeed);
84      std::cin.ignore();
85
86      std::cout << "Returning to Home Position..." << std::endl << std::endl;
87      SetAbsolutePanTiltPosition(CENTER, CENTER, true);
88
89      MainQuit();
90  }
```

Annotated Bibliography

[1] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

Ken Beck was heavily involved in the conception of agile methodologies, in particular *Extreme Programming*. This book provided an early description of the methodology.

[2] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mallor, K. Shwaber, and J. Sutherland, "The Agile Manifesto," The Agile Alliance, Tech. Rep., 2001.

The Agile Manifesto produced by the Agile Alliance, this document forms the standard by which all agile development is measured.

[3] W. R. Cheswick and S. Bellovin, *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

This book defines the concept of a network firewall and descrbes one of the first major firewall deployments at AT&T in the early 1990s. Defensive coding practices are discussed in relation to safety-critical systems.

[4] M. Cohn, *User Stories Applied*. Addison-Wesley, 2004.

Book detailing the application of user stories to agile development, written by the well known agile advocate Mike Cohn.

[5] T. H. J. Collet, B. A. MacDonald, and B. Gerkey, "Player 2.0: Toward a Practical Robot Programming Framework," in *Proceedings of the Australasian Conference on Robotics and Automation (ACRA)*, 2005.

Conference article describing version 2.0 of the *Player* project.

[6] D. Douxchamps, "libVISCA home page," http://damien.douxchamps.net/libvisca/, 2014, accessed December 2014.

The official homepage of *libVISCA*, the open source *VISCA* implementation used for reference in this project.

[7] E. Freeman, B. Bates, K. Sierra, and E. Robson, *Head First Design Patterns*. O'Reilly, 2004.

A well-regarded book providing a primer on object-oriented design patterns.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley, 1994.

The venerable "Gang of Four" book which laid down the fundamentals of object-oriented design, still relevant today.

[9] B. Gerkey and B. Tonkes, "sony-ptz," *Player* Driver http://playerstage.sourceforge.net/doc/Player-cvs/player/group__driver__sonyevid30.html, 2011 May, included with *Player* at `/server/drivers/ptz/sonyevid30.cc`.

The *sonyevid30* driver is the driver currently included with *Player* for use with Sony *EVI* cameras, this driver forms the basis of the new system and elements of it are taken verbatim or modified for use in the new driver with full attribution and thanks.

[10] F. Labrosse, "Sony PTZ Cameras," Mailing List http://sourceforge.net/p/playerstage/mailman/message/31468338/, September 2013.

This mailing list post helped define a significant amount of the project requirements.

[11] B. Livshits, "Improving Software Security with Precise Static and Runtime Analysis," Ph.D. dissertation, Stanford University, 2006.

This PhD thesis presents an analysis of methods for improving the security and stability of software. Section 7.3 ”Static Techniques for Security” describes issues related to static analysis.

[12] R. Mattes, “Player and CMake,” (Wiki) http://playerstage.sourceforge.net/wiki/Player_and_CMake, January 2012, accessed January 2015.

Project wiki page providing justification and details for using *CMake* with *Player*.

[13] R. Mattes and E. Walters, “Writing a Player Driver,” (Wiki) http://playerstage.sourceforge.net/wiki/index.php?title=Writing_a_Player_driver&action=history, February 2012, accessed January 2015.

The project wiki page describing how to write a simple *Player* driver, including the required functions and methods.

[14] B. Petersen and J. Fonseca, “Writing Player/Stage Drivers: A how-to for ESRP Player Driver Source Package,” University of Copenhagen, http://image.diku.dk/mediawiki/images/e/e2/Driverhowtodoc_HandedIn.pdf, Tech. Rep., 2006.

This technical report outlines the steps to create a simple *Player* driver, it was extremely useful during the first iteration of development.

[15] W. W. Royce, “Managing the development of large software systems,” in *Proceedings of IEEE WESCON*, vol. 26, no. 8. Los Angeles, 1970.

One of the first articles to ever describe a software development methodology similar to the *waterfall* method. It was initially presented as a flawed software design.

[16] J. Shore and S. Warden, *The Art of Agile Development*, 1st ed. O’Reilly, 2007, 9780596527679.

This book is a practical guide for developing software using agile methodologies and includes a defintion of what ”done done” means.

[17] A. Sidky and J. Arthur, “Determining the Applicability of Agile Practices to Mission and Life-Critical Systems,” in *Software Engineering Workshop, 2007. SEW 2007. 31st IEEE*, March 2007, pp. 3–12.

A thought-provoking paper on the suitability of agile methods for mission and life-critical systems.

[18] Y. Sugimori, K. Kusunoki, F. Cho, and S. Uchikawa, “Toyota production system and Kanban system Materialization of just-in-time and respect-for-human system,” *International Journal of Production Research*, vol. 15, no. 6, pp. 553–564, 1977.

This journal article provides a description of how the *Kanban* scheduling system was first conceived at Toyota.

[19] Unknown, “scan-build: running the analyzer from the command line,” http://clang-analyzer.llvm.org/scan-build.html.

Official *clang* project page for the *scan-build* static analysis tool. Describes how the tool works and how to use it.

[20] ——, “VISCA RS-232C CONTROL PROTOCOL,” https://pro.sony.com/bbsc/assetDownloadController/EVID70_D70P_Technical_Manual.pdf?path=Asset%20Hierarchy$Professional$SEL-yf-generic-153703$SEL-yf-generic-153738SEL-asset-116847.pdf&id=StepID$SEL-asset-116847$original&dimension=original, Sony Inc., accessed January 2015.

A mirror to the official specification of the *VISCA* protocol, unfortunately there is no provided author or date information.

[21] D. van Heesch, “Doxygen: Main Page,” http://www.stack.nl/~dimitri/doxygen/manual/starting.html, January 2015.

The official homepage of *Doxygen*, this tool was used for adding comments to the source which could then be output as LaTeXor HTML, as well as for generating diagrams from code automatically.

[22] Various, "ConfigFile Class Reference," http://playerstage.sourceforge.net/doc/Player-1.6.5/player-html/classConfigFile.php, May 2005, accessed February 2015.

Autogenerated documentation for the *ConfigFile* class, demonstrating the methods available.

[23] ——, "Technical Report on C++ Performance," International Standards Organization, Tech. Rep., February 2006.

An in-depth technical report from the International Standards Organization with a section discussing the potential issues of using pure C++ for low-level systems development.

[24] ——, "GNU General Public License," http://www.gnu.org/licenses/gpl.html, Free Software Foundation, June 2007.

Version 2 of the open source GPL license, one of the most widely-used software licenses in the world.

[25] ——, "Writing configuration files," (Wiki) http://playerstage.sourceforge.net/wiki/Writing_configuration_files, January 2011, accessed January 2015.

Project wiki page describing the structure of *Player* configuration files and how to write and use them.

[26] ——, "Intelligent mobile robotic platforms for research, development, rapid prototyping," http://www.mobilerobots.com/Mobile_Robots.aspx, 2014, accessed January 2015.

The page for *Adept Mobilerobots*, creators of the *Pioneer* robots commonly used for research and student projects at Aberystwyth University and elsewhere.

[27] ——, "Player Project," http://playerstage.sourceforge.net, February 2014, accessed December 2014.

The official homepage of the *Player* project, which includes the eponymous robot device interface, the *Stage* multiple robot simulator and the *Gazebo* 3D multiple robot simulator.