Robotic Control of a Photometric Telescope

Hendrik Petrus van Heerden

July 2011

A project submitted in partial fulfilment of the requirements for the degree M.Sc.
in the Department of Astronomy, as part of the National Astrophysics
and Space Science Programme
UNIVERSITY OF CAPE TOWN

Supervisors: Dr. Patricia A. Whitelock (UCT, SAAO) and Dr. Peter Martinez (SAAO)

Abstract

The objective of this project was to design and implement the telescope control software for the Alan Cousins Telescope (ACT), a robotic telescope at the Sutherland, South Africa, site of the South African Astronomical Observatory (SAAO). Numerous challenges were encountered in terms of the design and implementation of the software and novel approaches had to be employed to overcome them. The approach adopted involved a high level of modularisation of the software and the use of software simulators in an effort to foster rapid software development and enable more rigorous software testing.

Signficant attention was also paid to new and emerging technologies and how these may be used within the specific context of robotic telescope control software. The software now acts as a demonstration platform for those technologies that were implemented. Furthermore, seeing as the infrastructure for testing new technologies is well implemented in the software, the software may in future be used as a testbed for the development or implementation of new technologies.

The software was mostly developed off-line at the SAAO offices in Cape Town, South Africa, followed by a lengthy period in Sutherland in order to do in-situ testing of the software at the telescope. Although neither the testing nor the commissioning was complete at the time of completing this thesis, the small number of software components that had not undergone rigorous in-situ testing had been demonstrated to be functional by simulated tests in the laboratory in SAAO Cape Town.

During the in-situ testing of the software, a number of problems were identified in the optical alignment and mechanical systems of the telescope, which fall outside the scope of this thesis and will be addressed during the commissioning of this telescope in late 2011.

Acknowledgements

The author would like to thank the National Research Foundation for a bursary supplied through the National Astrophysics and Space Science Programme as well as the South African Astronomical Observatory for financial support for the project.

Furthermore, the author would like to thank Stephen Potter (SAAO), Therese Betita (former SAAO intern), Peter Martinez (SAAO), Luis Balona (SAAO) and Greg Cox for supplying software source code that several components of the ACT telescope control software are based on or inspired by.

The author would also like to thank Peter Martinez (SAAO) and Dave Kilkenny (University of the Western Cape) for their useful insights and assistance related to technical aspects of the telescope, as well as several other SAAO staff members (namely Pieter Fourie, Dave Carter, Pieter Swanevelder, Geoff Evans, Reginold Klein, John Stoffels and Johnny Klein) and Southern African Large Telescope staff members Francois Stumpfer and Janus Brink for their work on the telescope and for providing technical support during the validation phase of the project.

Lastly, I would like to extend my heartfelt gratitude to my family and friends for their support during this project.

Plagiarism Declaration

I, Hendrik Petrus van Heerden, know the meaning of plagiarism and declare that all of the work in the document, save for that which is properly acknowledged, is my own.

Contents

List of Figures

List of Tables

Chapter 1

Robotic Telescopes

This chapter outlines the advantages of robotic telescopes and the motivation for building them as well as the hardware and software requirements for robotic telescopes. The chapter concludes with a brief summary of existing robotic telescopes around the world.

1.1 Advantages of Robotic Telescopes

The main advantage of robotic and remote-access telescopes is that they require no human operators, thus reducing the overall operational costs of the telescopes (for instance, travel expenses for astronomers and operators are removed and they require no lodgings at the telescopes). Remote-access telescopes and automated telescopes have enabled the construction of completely unmanned observatories. Although an observatory of exclusively automatic and remote-access telescopes should ideally be completely unmanned, in reality (at least) technical support staff must be on-site. In theory, a completely unmanned observatory would introduce the opportunity for further cost-cutting measures, including the fact that no accommodation, amenities and infrastructure is required. Unmanned observatories can also be built in more inhospitable and inaccessible parts of the world such as Antarctica and high up in the Himalayas.

Robotic observatories are ideal for long-term projects, such as the long-term monitoring of variable stars, searching for eclipsing extra-solar planetary systems and monitoring for transient events. Without a robotic telescope, such a project would either require dedicated human observers or suffer from the effects of gaps in the data.

By positioning a remote-access telescope at an appropriate longitude, astronomers can observe during office hours. The astronomer's attention can then be diverted to other tasks, such as data analysis, while the telescope is busy with a lengthy operation, such as data acquisition. In the case of long-term projects with robotic telescopes, the astronomer can simply retrieve the data after a few days, months or years and perform data analysis.

A more recent application of robotic telescopes is to provide rapid response to transient events - for example, to collect crucial data shortly after a supernova as well as monitoring

the supernova afterglow (which can last several days). In this case, the telescope control software is linked to a network of stations monitoring for transient events. When a transient event occurs, an alert is distributed across the network, which then triggers the telescope(s) to observe the object of interest.

Robotic and remote-access telescopes have also made it easier to observe from several sites during a multi-site campaign. A number of telescopes can be built by a single organisation at various observatories around the world. These telescopes can then be operated from a central point in order to gain continuous coverage of a target.

Telescope time at various observatories are usually allocated in slots spanning several days to several weeks. It is therefore difficult to get coverage of a particular target for only a few minutes to a few hours per night. This problem can be easily solved with a robotic telescope or a remote-access telescope, as the slots can be as short as several minutes. This can, in turn, simplify the matter of slot allocation and division of available telescope-time, which can simplify the formation of consortia of astronomical institutions.

1.2 Hardware and Software Requirements

The primary design goal for the end-to-end development of a robotic telescope system should be reliabity. Ideally, robotic and remote-access telescopes should be able to function for months to years at a time without any direct, human interaction. This requires that both the hardware and software operate in exactly the same way during the course of a night and through all seasons.

When developing a robotic or remote-access telescope, it is necessary to assume that the user is either absent or very distant and that technical support is several days away (even if that is not the case). This requires a measure of situational and self-awareness. The system needs to determine when it is safe to open the dome and needs to respond to changes in its environment as well as determining whether a particular observation is safe and viable. The system must also be able to detect hardware and software failures and must be able to respond appropriately. As an example, if a software error occurs, it may be necessary to restart the software. Hardware errors may require a that technician be sent out to resolve the problem, which requires that the software be able to sense the error and submit a support request. In more extreme situations, a moving part driven by a motor may get stuck, in which case the motor could continue driving, which can (in turn) cause damage to the motor, moving part(s) or neighbouring components - the software should sense such a failure, stop the motors and request support.

Working with the assumption that the user is absent demands that the system be designed such that potentially hazardous operations are avoided in both the hardware and the software. In general, it is common practice to prevent the most serious and critical damage in the hardware and then to prevent other, lesser potential hazards in the software. The assumption should always be that commands that may cause damage to the system were sent in error. In most (if not all) cases, this means reducing the freedom of the user

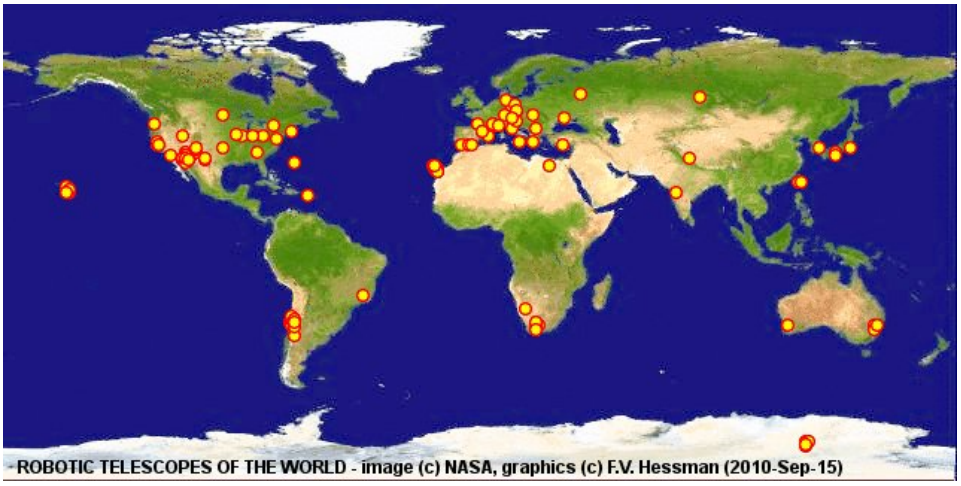

Figure 1.1: Map of the World showing the robotic and remote-access telescopes. [10]

and developers are frequently faced with catch-22 situations where user freedom should be weighed against the potential for accidental damage to the system.

1.3 Robotic Telescopes Around the World

There are dozens of robotic telescopes around the World - with a wide variety of aperture sizes, instruments and so forth between them [9] (see Figure 1.1). These telescope systems were designed and built with particular goals in mind, such as:

- Long-term monitoring of variable stars.
- Conducting astronomical surveys.
- Searching for Near-Earth Objects and Solar-system planetoids.
- Searching for extra-Solar planets.
- Monitoring the afterglows of gamma-ray bursts.
- Doing education and/or outreach activities.
- Collecting weather and atmospheric data.

The Las Combres Observatory Global Telescope network* is one of several currently-running projects intended to be used for outreach and education. The LCOGT is in the process of building a vast network of 0.4-2 m class telescopes that will stretch across the Earth. Apart from being used for scientific observations, the network will be used as an outreach and education tool. The longitudinal spread of the network means that members

*http://lcogt.net

of the public can operate an astronomical telescope on the other side of the side of the World during office hours (or school hours).

Robotic telescopes are extensively used by surveys to detect near-Earth objects and other Solar-system objects (for instance the LIncoln Near-Earth Asteroid Research[†] project (LINEAR)). Such surveys are usually conducted using a number of telescopes with large-format CCD imagers that scan selected parts of the sky a number of times per night.

Several robotic telescopes have been operated for longer than a decade (see [9]), which demonstrate how effective robotic telescopes are in the study of long-term variations in astronomical objects and monitoring astronomical objects in general. A more recent development is the practice of using robotic telescopes both for monitoring purposes and to observe transient events (such as supernova afterglows). Such telescopes are usually connected to a transient notification network so the telescope can automatically switch from the regular monitoring programme to observations of the transient event.

[†] `http://www.ll.mit.edu/mission/space/linear/` and `http://neo.jpl.nasa.gov/programs/linear.html`

Chapter 2

The Alan Cousins Telescope

Although the development of the ACT's hardware falls beyond the scope of the project, a brief summary of the hardware will be given in this chapter. The development of the software interfaces with the various hardware components is given in detail in Appendices A-D.

Section 2.1 gives a broad overview of the hardware as well as some of the ACT's history. Section 2.2 describes how the telescope is moved and how feedback of the current telescope orientation is produced and received. Section 2.3 describes the ACT's photometer (including the scientific instrument, acquisition system and supporting hardware). Section 2.4 gives a brief introduction to the PLC, which is used to manage a wide variety of hardware. Section 2.5 explains how the control computer gets access to accurate time.

2.1 Overview

The Alan Cousins Telescope was formerly known as the Automatic Photometric Telescope (APT). More details on the initial development of the APT can be found in Martinez et al. [14]. As described in Martinez et al. [14], the telescope is a Dall-Kirkham reflector with a 0.75-m f/2.5 primary mirror. The mount is an equatorial horseshoe mount in popular use with automated telescopes.

The telescope and structure were designed and built such as to maximise hardiness and long-term stability and to minimise size while still providing easy access for maintenance. The ACT facility therefore has none of the creature comforts usually found in a manned telescope (such as a warm-room, kitchen, lavatory, etc.). In various orientations, the telescope clears parts of the interior of the dome by no more than a few centimetres. There is only one known orientation of the telescope and dome where the telescope and dome can collide. Such a collision can only occur when the telescope is pointing due North and at high zenith distance and the dome is pointing South. This scenario cannot conceivably occur during normal operation and, in fact, it was discovered purely by accident.

After several years of operation, a major upgrade of several telescope systems was done.

Figure 2.1: The workbench at the SAAO electronics laboratory in Cape Town where the software interfaces with the hardware were tested. From left to right are the PLC, signal generator (to simulate a PMT), telescope drive electronics, telescope pointing encoder electronics and acquisition CCD with supporting electronics.

The upgrade included telescope pointing encoders to improve pointing, a programmable logic converter (PLC) to simplify interacting with various hardware components and new control computers and software.

During the course of the development of the software interfaces with the hardware, the relevant hardware (see 2.2) was put in the electronics laboratory at the SAAO offices in Cape Town and connected to artificial inputs to simulate feedback from the electronics connected to the telescope.

2.2 Telescope drives

The ACT is driven by a pair of metal-on-metal friction drives - one for right ascension and one for declination. The RA horseshoe rests upon a pair of rollers, of which one is driven by a stepper motor through a system of pulleys - thus effecting mobility in RA. The declination horseshoe is also in contact with a roller, similar to the rollers supporting the RA horseshoe, which is also driven by a stepper motor through a series of pulleys. The motors and the electronics controlling them were not upgraded since spare parts are readily available and the current system has been in use since the telescope was built and has proven to be very robust. The telescope did not initially contain pointing encoders; instead the software counted motor steps from a fiducial point in order to establish the telescope orientation. This method was fairly reliable, but the pointing accuracy suffered from the fact that the

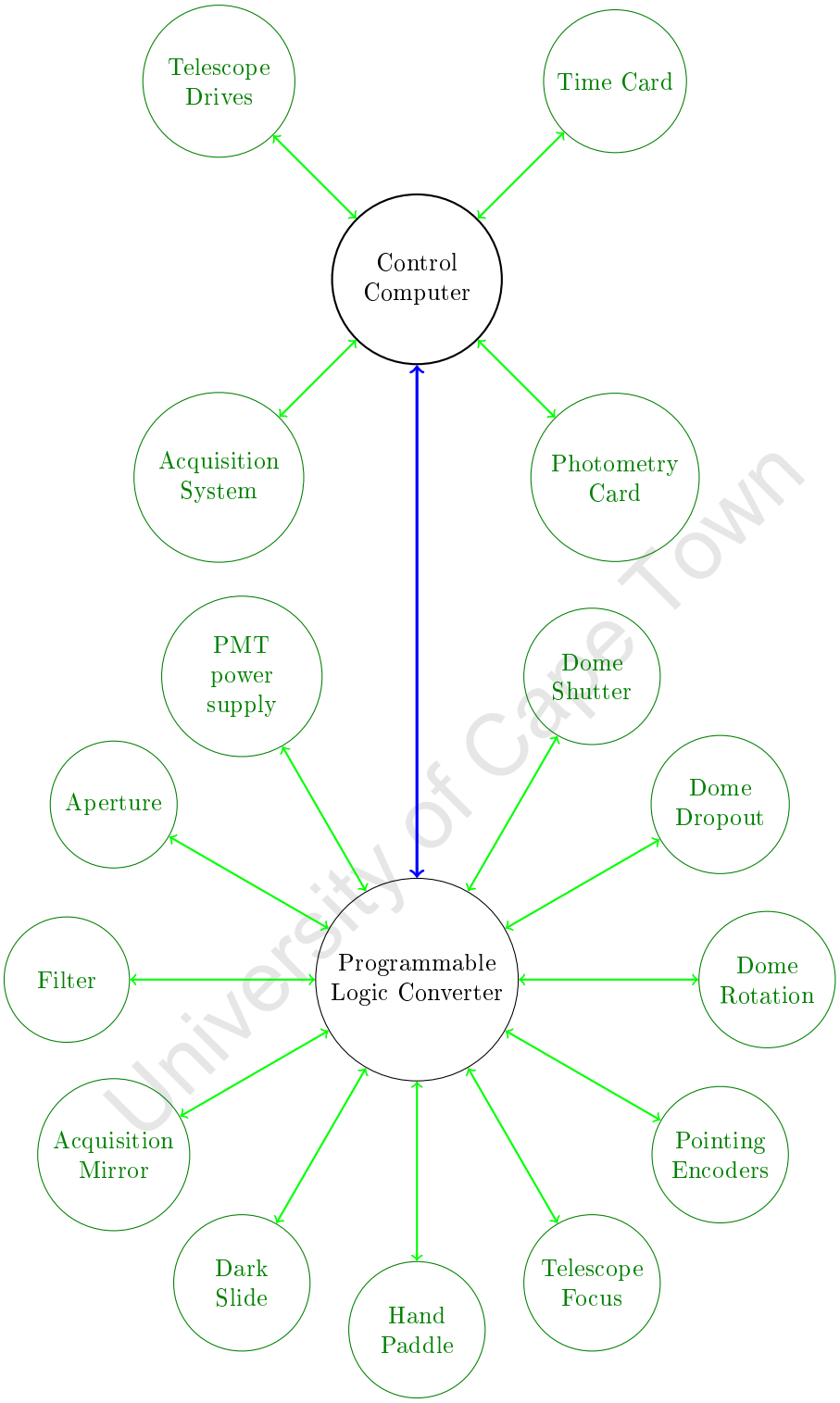

Figure 2.2: Overview of the ACT hardware. The control computer controls all hardware either directly (through special PC expansion cards) or via the Programmable Logic Converter (PLC).

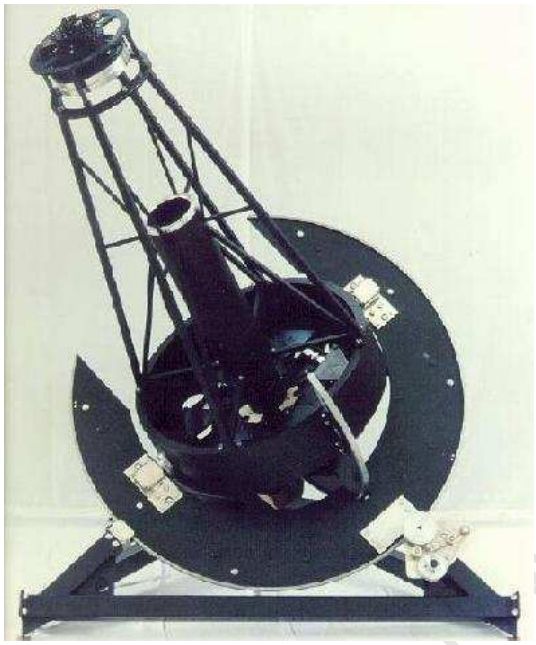

Figure 2.3: The telescope without the mirror, before it was installed in the dome. [11]

drives occasionally slipped. For this reason, it was decided that the upgrade should include encoders on both the RA and Dec axes.

2.2.1 Telescope limits

Each horseshoe has an upper- and a lower electronic limit and an upper- and a lower mechanical limit. The electronic limits are effected by thin metal plates mounted on the sides of the horseshoes which trigger optical switches mounted on the supports of the mount. When one of the electronic limits are triggered, the corresponding telescope drive stops but remains charged, thus holding the telescope in its current orientation on that axis. This condition needs to be cleared by moving the telescope in the opposite direction. Should the electronic limits fail, the mechanical limits are able to prevent the telescope from sustaining damage. The mechanical limits are effected by pins inserted perpendicularly into the horseshoes, which prevent any motion beyond the designated points by the pins hitting supports of the mount. The control software should be able to recognise when either the mechanical or electronic limits have been triggered and respond accordingly and should enforce certain additional limits as well (see 8.2.1).

2.2.2 Pointing encoders

In an effort to improve pointing accuracy, the telescope upgrade included a pair of relative encoders with sufficient resolution to meet the ACT's operational pointing requirements. The fiducial points for the encoders are the Northern and Western electronic telescope limits.

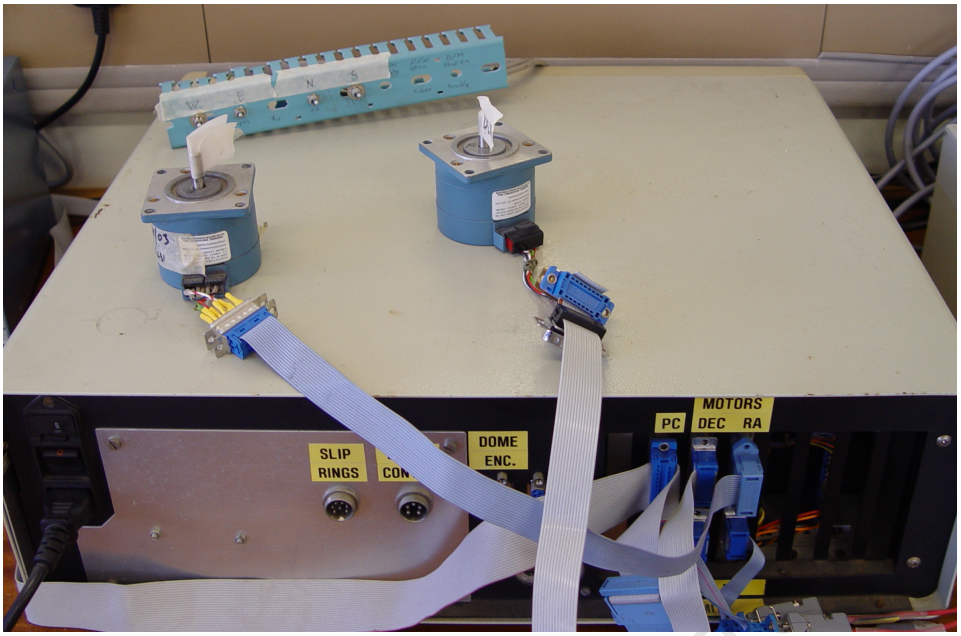

Figure 2.4: The telescope drive electronics (contained within the case), the RA and dec motors themselves (on top of the case, near the front) and the electronic limit simulators (the switches on the blue strip on top of the case, near the back).

The electronics managing the encoders were produced at the SAAO and integrated with the PLC. The encoder readouts are accessed through the PLC and the encoders are zeroed by the encoder electronics at the fiducial points. Due to the fact that relative encoders are used (as opposed to absolute encoders), the encoder readout is set to zero if an encoder loses power at any time. In this case, the encoders must be initialised by moving the telescope to the fiducial points. The encoders should be initialised periodically to correct for any drifts in the encoder readouts (due to the mechanical slipping of the encoder shaft), although it should not be necessary to do so more than once per night. Initialising the encoders is therefore part of the telescope start-up procedure.

2.3 Photometer

The photometer consists of several modules built into a chassis, which allows modules to be easily accessed for servicing under operational conditions.

Figure 2.7 shows a schematic view of the photometer. This section is subdivided according to the light path within the photometer:

1. Light entering the photometer reaches the view mirror (labelled `A` - also known as the acquisition mirror) first, which has two positions. When in the out-beam position, the light is reflected toward the acquisition system and in the in-beam position, the on-axis light passes through a hole in the acquisition mirror toward the photometric system. See §2.3.1.

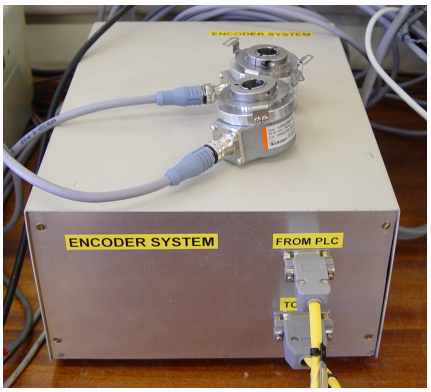

Figure 2.5: The telescope pointing encoders and supporting electronics.

2. Light passing through the acquisition mirror and bound for the photometric system is reflected by the fixed mirror (B) and passes through the filter and aperture wheels (C and D, respectively) - §2.3.2.

3. The instrument shutter (or dark slide - E) follows after the aperture wheel - §2.3.3.

4. The target is imaged on the photomultiplier (G - see §2.3.4) by a Fabry (field) lens.

5. Light may be diverted by the acquisition mirror toward the acquisition system (instead of passing through to the photometric system). In this case, the light path ends at the acquisition CCD camera (F) - §2.3.5.

2.3.1 Acquisition Mirror

The beam can be diverted from the PMT toward to the acquisition CCD by a rectangular 45° mirror. One half of the mirror diverts the entire field to the CCD (out-beam position), the other half has a hole in it to allow on-axis light through to the PMT and diverts only the outer edges of the field to the CCD (in-beam position). The thin annulus of the field that is visible on the CCD when the mirror is in-beam can (in principle) be used for auto-guiding, although in practice it was found that the annulus rarely contains stars bright enough for auto-guiding.

2.3.2 Filter and Aperture Wheels

Both the filter wheel and the aperture wheel have 10 available slots and both are controlled by the PLC. The filter wheel currently has Johnson U, B and V filters, Cousins R and I filters and a beta source (for calibration purposes). The aperture wheel currently contains 780″, 90″, 60″, 45″, 35″, 30″, 25″, 20″ and 15″ diameter apertures.

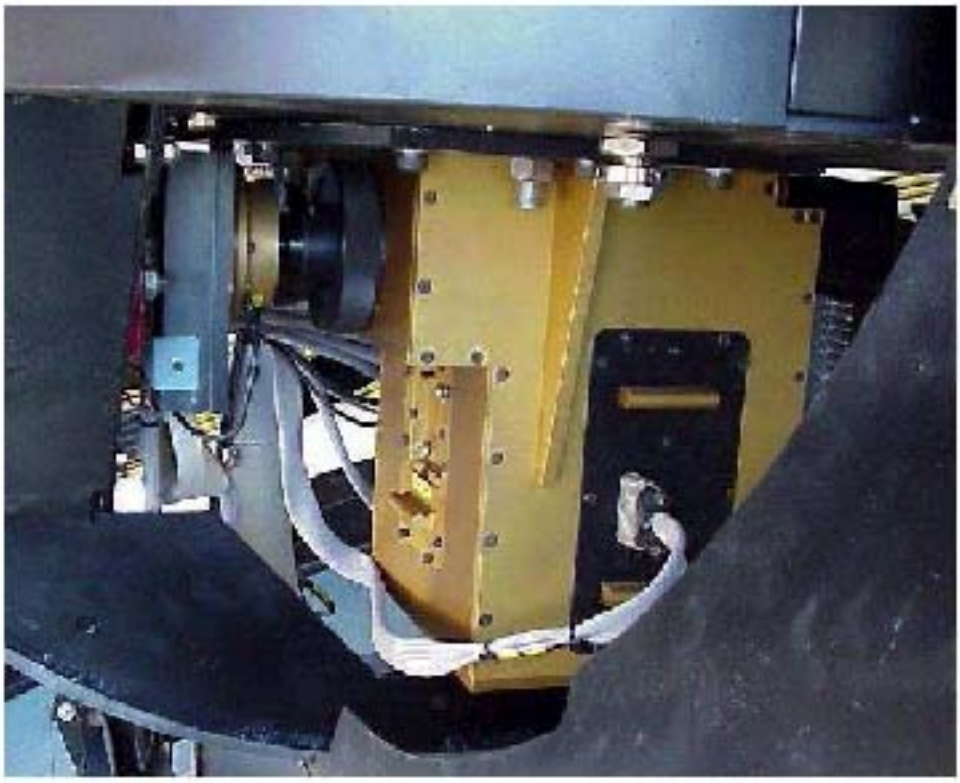

Figure 2.6: The photometer, shown here mounted on the telescope. [11]

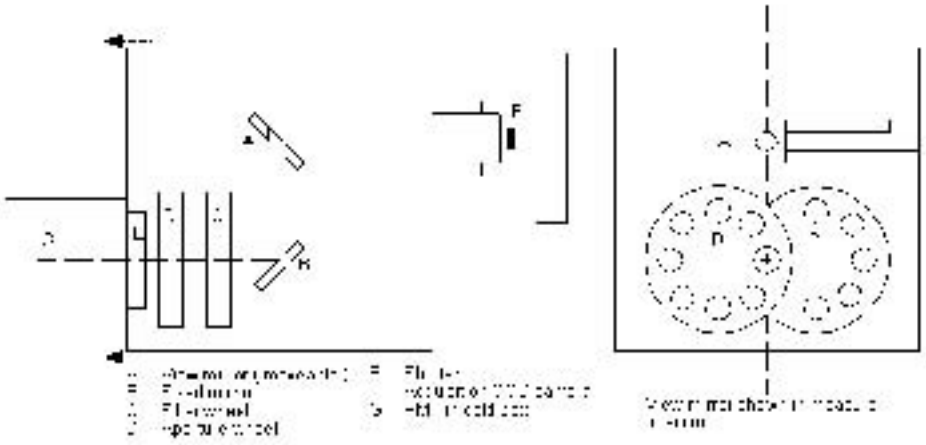

Figure 2.7: Diagram of the photometer. [11]

2.3.3 Dark Slide

A Hall-effect iris diaphragm is used as a "dark slide" to cover the PMT when the PMT is not in use and to protect the PMT in case of over-illumination. As the diaphram requires electric power to remain open, it closes in the event of a power failure.

2.3.4 Photometric Detector

The ACT uses a thermoelectrically-cooled Hamamatsu R943-02 photomultiplier tube. Due to the limited space available in the photometer, a special right-angle "folded" cold-box had to be manufactured.

The PMT is connected to an SAAO PC expansion card ("photometry card") specifically designed to interface with PMTs and a high-voltage power source controlled by the PLC.

During the development of the software interface with the photometry card in the laboratory, a signal generator was used to simulate counts from a PMT.

2.3.5 Acquisition System

The acquisition CCD and its supporting electronics (the "MERLIN" crate) attach to the side of the photometer chassis. The MERLIN crate interfaces with the control computer via an SAAO PC expansion card. The CCD is a Peltier-cooled engineering-grade frame-transfer EEV CCD with 385×288 pixels (in full-frame mode). A focal reducer is employed to project an f/2.8 image onto the CCD, yielding a field of view of $15' \times 11'$ and a pixel resolution of $2.3'' \times 2.3''$.

2.4 PLC

A Programmable Logic Converter (PLC) was designed and built at the SAAO to manage various aspects of the dome, telescope and instrument. The PLC acts as an intermediary between the control computer and all the hardware connected to the PLC. Without a PLC, the control software would need to have specific interfaces implemented for each item of hardware. The electronics of the PLC is programmed with interfaces to all the specific hardware and thus allows the control software to issue commands to and get feedback from various pieces of hardware in a generic and easily-accessible manner.

The PLC accepts a control string from the control computer and returns a status string to the control computer (when requested). Both the control string and the status string consist of an array of C **chars** where each **char** represents a particular command to be issued to a hardware component or the current status of a particular hardware component. The **chars** do not represent human-readable text, instead the array of **chars** is interpreted as a series of binary digits.

The following hardware components are managed by the PLC:

- Dome shutter

- Dome dropout
- Dome rotation (both manual and automatic)
- Telescope pointing encoders
- Telescope focus
- Hand paddle (to move telescope in cardinal directions and adjust the telescope focus manually - status only)
- Dark slide
- Acquisition mirror
- Filter
- Aperture
- PMT high-voltage power-supply

2.5 Time

The control computer has access to millisecond-accurate time through the SAAO time service. Each control computer has a specialised SAAO time service expansion card ("time card") which connects to the time service and makes the information available to the computer. The current time can be read from the time card by any privileged programmes running on the computer and the time card can issue Interrupt ReQuests (IRQs) which can be used to schedule regular tasks for the Central Processing Unit (CPU) of the control computer.

2.6 Weather Data

The ACT does not currently have its own weather sensors. Instead, the weather sensors of other facilities at the SAAO in Sutherland must be relied upon. The following facilities record weather data and publish them in real time, either on the SAAO intranet or globally through the internet:

- Southern African Large Telescope* (SALT)
- Wide Angle Search for Planets* (SuperWASP)
- South African Geodynamic Observatory Sutherland† (SAGOS)

*`http://www.salt.ac.za/~saltmet/weather.txt`
*`http://swaspgateway.suth/`
†`http://sg1.suth/tmp/kan11.htm` and `http://sg1.suth/tmp/kan16.htm`

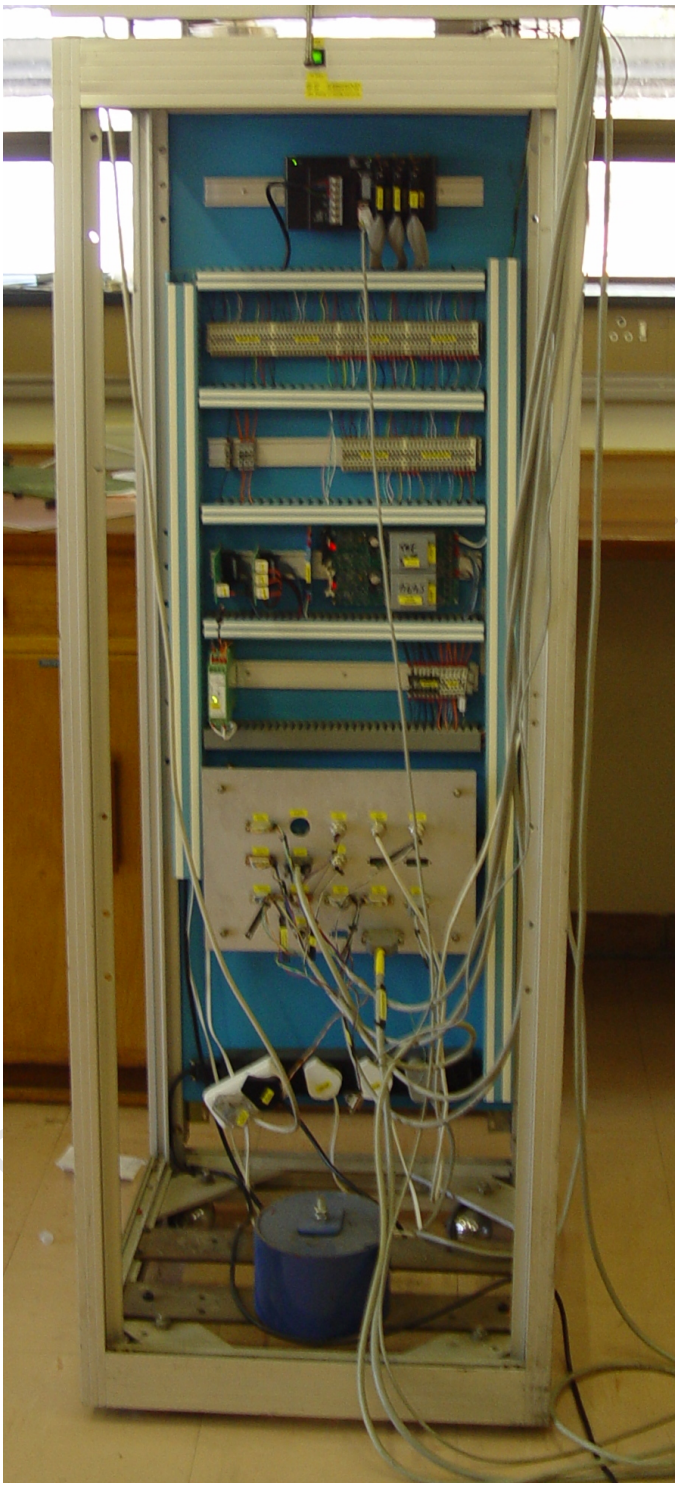

Figure 2.8: Programmable Logic Converter, shown here with its cover panels removed. During the development of the software interface with the PLC at the SAAO electronics lab in Cape Town, the PLC was connected to switches which were used to simulate feedback from all the hardware connected to the PLC.

Chapter 3

Design of a Robotic Control System for the ACT

In the simplest description of the problem, the control software must be able to collect photometric data of astronomical targets. This chapter refines this singular, simple goal and discusses various aspects of it in greater detail as well as software design strategies employed in order to accomplish this objective and appropriate, currently available software technologies. Since the ACT software should ideally be capable of robotic control in such a way that manual control of the telescope is unhindered, aspects of this dual-purpose will be discussed as well.

Section 3.1 describes the functional requirements of the telescope - i.e. the functions a robotic telescope control system are expected to do - and §3.2 describes the design strategies and ideals that must guide the development process in order to deliver a result that meets those requirements. Section 3.3 gives a brief overview of currently-available software technologies that may be employed either during the development process or during the regular operation of the telescope control software. Section 3.4 gives a list of parameters by which the successfulness of the telescope control software may be gauged.

3.1 Functional Requirements

The tasks of telescope control software are often underestimated. Before the development of the software could begin, these tasks had to be defined and investigated.

3.1.1 Data Collection

The collection of photometric data is the primary purpose of the telescope. At the most basic level, this entails capturing photometric measurements via some hardware connected to the telescope and recording it.

The problem becomes exponentially more complex when it is considered that the software must also be able to guarantee the safety of the telescope and its instrumentation and the correctness of the collected data. The software will produce this result if it can ensure that the correct target is being observed, take atmospheric effects into account and remove (or at least identify and flag) effects introduced into the data by the hardware or the software.

3.1.2 Instrument Control

Apart from collecting photometry, the control software is also tasked with controlling various aspects of the instrument - this includes setting the filter and aperture, controlling the acquisition CCD, moving the acquisition mirror, opening and closing the instrument shutter and so forth. Simply issuing commands to the relevant hardware components is insufficient - the control software must also monitor these components and must be able to detect incorrect and/or dangerous behaviour. Since most of these components have moving parts, they can get "stuck" and the parameters by which the control software is able to judge whether a command was successfully executed or an error has occurred depends on the design of the hardware and the hardware sensors available to detect such an error. The action the control software should take also depends on the particular hardware component.

A particularly dangerous scenario is that of an over-illumination of the photomultiplier tube. In such a situation, the control software should close the instrument shutter within a fraction of a second. Should the instrument shutter not close properly, the photomultiplier tube may be damaged. For this reason, it may be necessary to put in place additional measures to protect the PMT, such as moving the filter wheel to an appropriate filter and moving the aperture wheel to the smallest available aperture. It is highly unlikely that all three of these components would fail simultaneously. However, should the over-illumination condition persist, several steps may be taken to prevent damage to the photomultiplier tube - such as moving the telescope away from the target, closing the dome shutters or defocusing the telescope.

3.1.3 Telescope and Dome Control

The control software should naturally be able to move the telescope to the desired target, to align the dome aperture with the telescope and to open and close the dome shutters.

Moving the telescope is simple enough - although doing so safely and accurately is a much more complex matter. The software must prevent a collision between the telescope and parts of the dome or structure. These preventative measures must be implemented both during robotic and manual control, however during manual control the appropriate action could be limited to merely displaying a warning message to the user and noting the event in the log. The software must also be able to identify when a collision has occurred (by, for instance, checking that the telescope pointing encoders are turning when the telescope motors are turning) and immediately stop all involved hardware components to prevent further damage. However, the telescope protection measures should not impede the normal

operation of the telescope and the extent to which these measures reduce the capabilities of the telescope should be minimal. The software must also smooth out any requested speed or direction changes in the requested motion of the telescope - failing that could cause increased wear on the telescope drives and supporting hardware. The software must also be able to identify any inaccuracies or inconsistencies in the feedback it receives from the telescope positional encoders - for instance, the software should identify any systematic drifts in the encoder feedback and take them into account.

3.1.4 Situational Awareness

Whether the software is under manual control (either locally or remotely) or under robotic control, it must respond to external stimuli from a variety of sensor inputs, such as metereological sensors. In the case of manual control, the current weather conditions should be displayed to the user and appropriate warnings issued if the user attempts to perform an action that is potentially unsafe under the current weather conditions (e.g. high humidity, high wind speed, rain and snow) or fails to secure the telescope under changing weather conditions. In the case of robotic control, the control software should emulate a sensible human observer in terms of taking action to guarantee the safety of the telescope.

Beyond meteorological conditions, factors such as the positions of the Sun and Moon must be taken into account. The telescope should not be in operation while the Sun is up (specifically, higher than a particular maximum altitude) and should not be pointed toward the illuminated part of the Moon, as this may cause damage to the photomultiplier tube. Observing targets near the Moon may also result in degraded data quality (due to very high sky background brightness) and should therefore be avoided.

3.1.5 Remote Access

Ideally, a user should be able to access the system from anywhere in the world. This is fairly easily attainable through the internet, but requires that special considerations be made for system security and access control. A suitable alternative is to allow access to the control system only through the local SAAO intranet. Another important factor is the issue of latency, which is related to the time span between a command being issued by a remote user and the telescope control software responding to that command and, conversely, the time span between an event occurring in the control software and the notification of that event being received by the remote user. All forms of remote access would necessarily introduce a latency greater than the (negligible) latency for a local user at the telescope, however the latency generally increases with increasing distance between the remote computer and the control computer and therefore (in terms of latency) shorter-distance remote access is preferable. A viable option is to only allow remote access within the intranet at the SAAO in Sutherland, although this mostly defeats the purpose of enabling remote access.

As will be discussed in §3.3.4, there are very effective software technologies available at the moment that have been used with great success to secure connections over a public

network (such as the internet). Furthermore, a combination of currently available technologies and software design strategies could be employed to reduce the negative effects of remote-access through a high-latency network on the usage experience of the remote user.

3.1.6 Robotic Operation

Most of the aspects of the robotic functions of the telescope have been covered in previous sections of this chapter. Ultimately, the robotic facilities of the control software must emulate and replace a human observer such that data are collected and recorded accurately and the telescope is operated safely. There is much overlap between the robotic functions of the telescope and those functions of the telescope that are designed to ensure the telescope's safety during manual operation and to simplify manual control of the telescope. These functions should therefore be implemented in a general fashion so that they may be accessed from many different aspects of the control software.

3.2 Design Strategies

The development of the control software should be guided by several design ideals. Most notably:

- Reliability
- Usability
- Stability
- Maintainability

Aspects of software design strategies are explained in this section with these goals in mind.

3.2.1 Reliability

The concept of reliability pertains to both the measures employed to protect the hardware (see §3.1.3) and the integrity of the data collected (see §3.1.1). The software would be deemed reliable if it can guarantee consistent and safe operation of the telescope system or take appropriate action should that not be the case.

In order to realise this goal, the software would necessarily have to be implemented with a large measure of redundancy. Redundancy would have to be built into all parts of the software and at all levels - ranging from checks to confirm the validity of data passed between different components of the software to verifying that a particular command that was issued (either by the user or a particular component of the software) was not issued erroneously and regular checks to ensure that all aspects of the software are operating normally.

3.2.2 Usability

Usability primarily refers to the tasks the software is designed to complete - i.e. whether the telescope system (both hardware and software) can be effectively used to collect astronomical data. This notion entails both aspects related to the internal operation of the telescope system (the hardware, interfaces between hardware and software and operations within the software that are invisible to the user) and interface with the user (both in terms of a user controlling the telescope manually and issuing commands to be executed by the telescope robotically). The greatest burden in this context is deciding what level of control the user should have. More experienced users would prefer a greater level of control, although this could needlessly complicate the user interface, thus detracting from the overall user experience and making learning to use the telescope system a cumbersome task. At the other extreme, the system could be developed such that the user has less control over the system and the system "fills in the blanks" by assuming the way a task is always done is the way the task is most commonly done. The latter approach would result in a telescope system that is only useful in particular circumstances and lacks versatility.

The golden mean is then to develop the low-level, internal operations of the telescope software such that they are highly configurable, but then implement some concessions in terms of user control at higher levels and design a user interface that allows common tasks to be done quickly and simply while also granting a high level of control to more experienced users.

3.2.3 Stability

Software instabilities arise when errors in the software cause unexpected and/or dangerous behaviour in the telescope system. The possibility of such errors occurring can be greatly reduced by developing the software consistently. Consistent development practices should be observed during all stages and at all levels of the development: from the high-level task of structuring the software to the low-level task of defining a naming convention for items in the software source code that is comprehensive and effective.

Another way to reduce the effect of software instabilities is to check frequently for such instabilities and correct for them. Defining the software architecture in a very modular fashion would increase the number of points at run-time at which these checks could be naturally done and any corrective actions be taken. Furthermore, modules could be used to perform validity checks on the data passed between each other and to check the operation of other modules.

3.2.4 Maintainability

Maintainability in this context refers to how simply particular parts of the software source code can be upgraded (to implement new features in the software) as well as any errors in the source code can be be traced and removed. The maintainability of the software is therefore directly linked to the readability of its source code. Easily maintainable source code is

highly structured, rich with annotations and comments, follows a strict naming convention for methods and objects and is accompanied by comprehensive and easy-to-read developer documentation.

3.3 Software Technologies

3.3.1 The Linux Kernel

Most (if not all) modern desktop computers and operating systems support multi-tasking. Multi-tasking can ruin the real time response of the system as a whole, which is critical for the correct operation of the low-level hardware interfaces of the telescope software. Real time responsiveness is required in order to collect photometry with absolute timing and to interface with the acquisition CCD - in both cases the system needs to respond to a trigger (such as an event occurring that requires a response from the system or the time at which a scheduled task is to be executed, being reached) within tens of nanoseconds after the trigger. If the system resources (most notably the CPU) are occupied with another task in a multi-tasking environment, the system may not react to the trigger within a reasonable length of time.

The *Linux kernel* is the open-source core of all flavours of the Linux operating system. As such, third-party developers have created ways of guaranteeing real time responsiveness while using the Linux kernel. The *Real Time Application Interface** (*RTAI*)[13] is such a package and is currently used in several SAAO telescope control systems. However, the *RTAI* interface is cumbersome to use and there is a general lack of good documentation for it. At any rate, the Linux kernel is (to some extent) capable of providing real time responsiveness to processes that require it without any third-party additions [1]. This concept is implemented in the Linux kernel in the form of different levels of *pre-emption*. The default for desktop operating systems is to disable *pre-emption*, although voluntary *pre-emption* can be activated when the kernel is compiled. Voluntary *pre-emption* allows a privileged process that is idle and needs to become active to jump to the front of the CPU scheduling queue and replace the currently active process (thus halting the currently active process) at a point in the thread of execution of the active process that is labelled as being a natural point for the active process to be halted. Voluntary *pre-emption* is usually sufficient to guarantee real time responsiveness to privileged processes (such as the process managing the collection of photometry and the process controlling the acquisition CCD). Should voluntary *pre-emption* be insufficient, a higher level of *pre-emption* (real time *pre-emption*) can be activated by applying a third-party patch to the Linux source code before compiling it [7].

3.3.2 Graphical User Interfaces

Given the level of complexity of the software's user interface, the implementation of a *graphical user interface* (*GUI* - as opposed to a command-line or text-based user interface) is

*See `www.rtai.org`

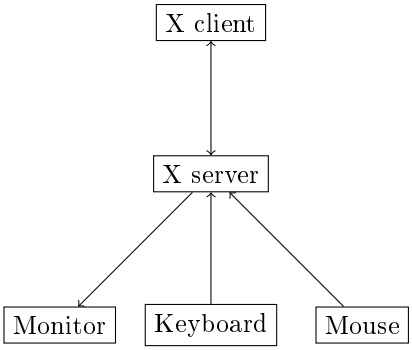

Figure 3.1: Schematic of the X Window System.

absolutely necessary. The availability of methods by which the creation of a *GUI* can be simplified have increased dramatically with the ever-increasing use of *GUIs*.

The ***X Window System***[†] (***X11***) is a suite of programmes designed to simplify the matter of developing and managing *GUIs*. Various flavours of ***X11*** are available on most UNIX-like operating systems. The ***X11*** server programme (X server) acts like a server programme that manages input and output hardware connected to the computer and serves the needs of the graphical programmes connected to it (which act like clients to the server). Graphical programmes can therefore produce output to any connected output device (such as a computer monitor) without any consideration for the nature of the output device and can expect standardised input from any connected input devices (such as a keyboard, mouse, trackball, touchpad or touchscreen). Clients (graphical programmes) connect to the X server through a ***network socket*** and communicate with it using a protocol specifically designed for this purpose (***XLib***). The nature of the client-server connection means that the client need not necessarily be run on the same computer as the server. See Figure 3.1.

Working with the XLib protocol can be cumbersome, since it is a very low-level communication protocol. For this reason, several graphical libraries or "toolkits" have been developed to further simplify the development of *GUIs*.

Given the complexity of the software, a well-documented graphical toolkit that is comprehensive, easy to use (from a development point of view) and in wide use must be chosen. There are only two toolkits that meet these criteria: *Qt* and *GTK+*[‡]. *Qt* is in some regards simpler to use because its natively used in C++ programmes and makes extensive use of the object-oriented functionality enabled by C++[§]. *GTK+* is natively used in C programmes and implements some object-oriented functionality using *GLib*[¶].

[†]See `http://www.x.org/`

[‡]GTK is an abbreviation of "GIMP ToolKit". GIMP is an abbreviation of "GNU Image Manipulation Program" and is the name of the image processing software for which GTK was initially developed. GNU is an infinitely recursive abbreviation of "GNU's Not Unix". The "+" in *GTK+* is to indicate that it implements some object-oriented features even though its native language (C) lacks such capabilities.

[§]See `http://qt.nokia.com/`

[¶]See `http://www.gtk.org/`

3.3.3 Data Storage

All currently available forms of data storage media are fallible. The useful life of a particular data storage device can be extended by reducing the frequency with which data is read from or written to the device (thus reducing wear). However, a more effective strategy to ensure accurate data storage is to implement a measure of redundancy.

The first method is to use a *Redundant Array of Independent Disks* (*RAID*), by which two (or more) storage devices are used in tandem (as opposed to one device). When data is written to the storage device, it is in fact written to both devices - thus two copies of all data are available at any given time. Should one of the storage devices fail, the defective device can be replaced and populated with the data from the intact device.

An effective data storage strategy should go beyond simply duplicating data at the same geographical location as the original. In the case of a robotic telescope, a fire or similar incident could destroy not only the telescope and structure, but also years worth of data if a remote backup is not kept. An effective and efficient means of keeping a remote backup would be to upload all data collected since the last backup to the backup server on, for instance, a daily basis. This can be easily implemented using a programme called *rsync*[‖], which can be used to scan the master copy of the data and a backup copy of the data and modify the backup such that it is identical to the master copy.

3.3.4 Remote Access

As mentioned in §3.1.5, remote access should ideally be available globally through the internet. This functionality requires that measures be taken to prevent unauthorised access to the control software. Although an authorisation and/or authentication scheme could be developed from scratch, several software technologies are currently available that are designed for this purpose.

Secure Shells

The *Secure SHell* (*SSH*)[**] suite of applications can be used to establish secure connections over insecure computer networks. *SSH* consists of a server and a client component. The *SSH* server runs on a computer and allows remote users that have a user account on that computer to log into it. The *SSH* server is highly configurable and the configuration should be set according to the required level of security. The default server configuration usually requires that all communication between the client and the server (even the authentication process) be encrypted, which should be sufficient in this context.

As mentioned in §3.3.2, a *GUI* programme can connect to a remote X server. In the context of remote access to the ACT software, the ACT software (the X client) would connect to the X server running on the remote machine from which the user has logged into the ACT control computer. However, this X server feature is usually disabled by default for

‖See `http://www.samba.org/ftp/rsync/rsync.html`
**See `http://www.openssh.com`

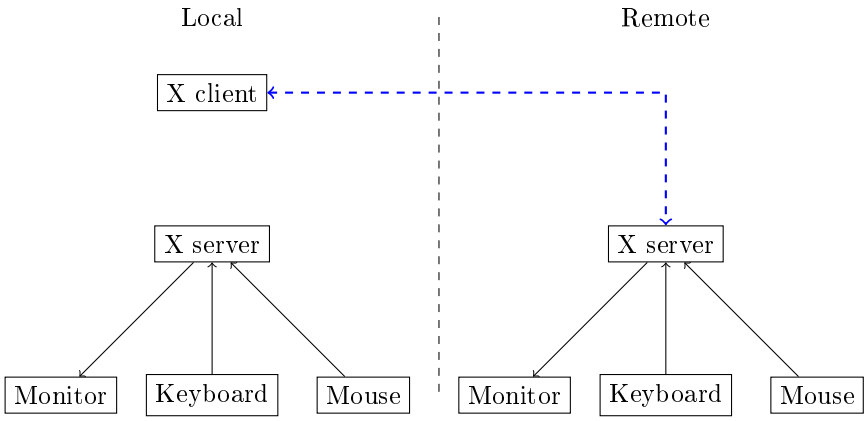

Figure 3.2: Schematic of *SSH* with ***X11*** forwarding (cf. Figure 3.1).

security reasons. Remote connections between X servers and X clients can be made secure by ***tunnelling*** the ***X11*** connection through the *SSH* connection, which would automatically encrypt all communication between the X server and the X client. Most popular *SSH* clients (OpenSSH, PuTTY, etc.) are capable of enabling such a ***tunnel***. Figure 3.2 gives a schematic representation of how graphical remote access is enabled using *SSH* (cf. Figure 3.1).

One of the major disadvantages of *SSH* is that it cannot provide access to the user interface of a programme that is already running on the server - a new instance of the programme must be started.

Virtual Network Connection

VNC[††] is a protocol that is in wide use to enable remote access to a computer. Many software programmes and suites are available that provide the client and/or server functionality that utilise the *VNC* protocol.

VNC uses much more of the server computer's system resources because, instead of simply forwarding XLib communications, the *VNC* server programme must effectively render the on-screen output in the background, take a screenshot, compress it and send it to the client programme. This usually also produces greater network traffic between the client and the server.

Furthermore, preliminary tests showed that using *VNC* for remote access introduces unacceptably large lags in user interactions.

NX

NX[‡‡] is a protocol developed and maintained by NoMachine (a division of MediaLogic S.p.A.) which allows remote access to Linux-based computers [21]. The *NX* server pro-

[††] See `http://www.hep.phy.cam.ac.uk/vnc_docs/index.html`
[‡‡] See `www.nomachine.com`

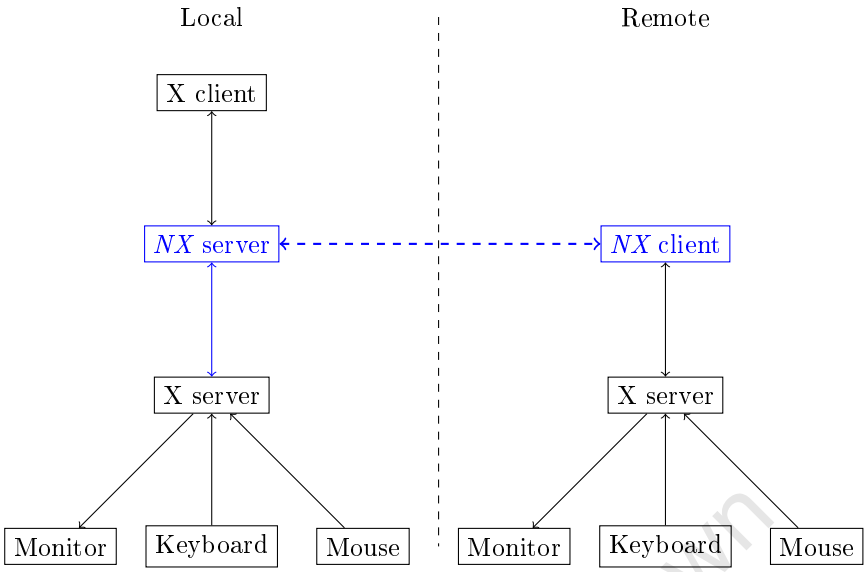

Figure 3.3: Schematic of remote access using *NX* (cf. Figure 3.1 and Figure 3.2)

gramme acts like a proxy between the X server and any *NX* clients (a remote access client programme capable of using the *NX* protocol) connected to it. The *NX* server programme duplicates the XLib output from all X clients running within a particular working session on the server computer, sending one copy of the output to the X server running on the server computer and one copy to each *NX* client programme connected to the *NX* server programme. The remote *NX* client receives the XLib output from the *NX* server programme and (like a normal X client) sends that output to the X server to which it is connected. The *NX* client also accepts input from the X server to which it is connected and forwards it to the *NX* server, which multiplexes that input with the inputs forwarded to it by the other *NX* clients connected to it. Figure 3.3 depicts how remote access is granted using the *NX* protocol.

NX ***tunnels*** through *SSH*, meaning that all communication is encrypted. *NX* is also documented as being very effective in reducing the effects of remote access through a high-latency connection [21].

3.3.5 Telescope Control Software

Due to the variety of robotic telescopes around the world, several attempts have been made to standardise certain components of the telescope control software in order to simplify the development of control software. The Automatic Telescope Instruction Set [8] , AStronomy Component Object Model [15] and Remote Telescope Markup Language [19] are examples of such standards. Although each of these standards has merit within the context for which it was developed, their usefulness as a multi-purpose telescope control system is doubtful.

There are also dozens of software packages in use on robotic and/or remote access telescopes (see Hessman[9]), although these are usually either tailor-made for a specific telescope

by the institution that builds it or provided as a commercial product. Some of these software packages come close to providing the functionality required by the ACT (see §3.1) and follow similar design ideals as those set out in §3.2. However, none of these software packages are general enough to cater for the specific needs of the ACT (in terms of available hardware and the functional requirements of the software) or easy enough to expand (within a reasonable length of time) such as to meet these requirements. One of these projects, namely the Remote Telescope System [12], is open-source, so the source code can be referred to for guidance if a particularly difficult problem is encountered during the development of similar telescope control software.

3.4 Gauge of Successful Completion

The quality of the telescope control software can be gauged according to the answers to the following questions:

1. Can the system record astronomical data?
2. Are the collected data absolutely reliable?
3. Does the software control the hardware effectively, comprehensively and safely?
4. How stable is the software and how well does it respond to instabilities?
5. How maintainable and expandable is the software source code?
6. How simple is it to use the telescope software?

The answer to each of these questions should be considered in the context of each mode of operation of the telescope software:

- Local, manual control
- Remote, manual control
- Robotic or automatic control

Chapter 4

The ACT Software System Architecture

This chapter outlines the structure of the ACT control software. Section 4.1 discusses the modular approach that was employed in designing the software architecture with an emphasis on the advantages, disadvantages and potential pitfalls involved with such an approach. Section 4.2 describes in finer detail how the modular approach was implemented in the specific case of the ACT control software, with a strong emphasis on communication between various modules of the software.

4.1 A Modular Architecture

Given the many and complex requirements for the software (see §3.1), it was decided to employ to a modular software design approach. The modular design meant that a number of independent programmes would be developed, each with its own set of tasks, capabilities and requirements.

4.1.1 Advantages of Modular Design

There are several advantages to a modular approach. Most notably, it allows for a tidier division of labour, as all functions related to a particular task, set of tasks or piece of hardware can be contained within a particular programme.

A modular approach simplifies the matter of resolving conflicts within the software. Such conflicts arise from simultaneous attempts to access hardware and from unexpected modifications of variables (especially global variables). A good example of such a conflict would be exposure commands sent to the image acquisition system. Given the limitations of the hardware, if an exposure is requested before a previously requested exposure is complete and its data read out, the result is unpredictable (in most test cases on the ACT this caused a hardware error which required a hard reboot of the image acquisition system).

A modular approach also allows for compartmentalisation of hardware-software interfaces - a particular programme can be assigned to handle all input and output from a particular piece of hardware. Communications with the Programmable Logic Converter (PLC) is a good example, as the command strings sent to the PLC can be difficult to construct and the status strings returned by the PLC can be difficult to interpret. Furthermore, some commands only need to be issued once, some need to be sustained until the desired effect is reached and some need to be sustained until the effect opposite to that caused by the command is desired. With a modular design, only a single programme needs the capability of constructing PLC command strings and interpreting PLC status strings. Commands to be issued to the PLC can then be sent to the programme responsible for PLC communications in a more accessible and/or human-readable form and that programme can disseminate PLC status information to the other programmes in a similarly accessible and/or human-readable form.

This division of labour also makes it simpler to locate bugs in the software, as incorrect behaviour of a piece of hardware or incomplete or incorrect execution of a command would almost certainly point to an error in the programme associated with it. The code that could possibly have caused the error is then significantly reduced and contained in a more contiguous subset of the code. In the case of errors from which a programme cannot recover which cause the programme to become unresponsive or exit abnormally, only the tasks assigned to that programme are interrupted and the normal operation of the software continues undisturbed.

A modular design enforces a certain minimum level of structure in the software. With a modular design, the high-level structure of the programme can be easily documented and communicated to future developers and can be more easily inferred from the code itself. It also allows for each programme to be more easily structured.

Should it be decided that a particular programme in the modular scheme is no longer adequate or that a better programme can be created, the old programme can simply be replaced with a new one.

4.1.2 Disadvantages of Modular Design

By far the biggest disadvantage of a modular software system is the complexity introduced by the need for communication between modules - Interprocess Communication (*IPC*).

It is an unfortunate fact that all methods of *IPC* are more resource-intensive than accessing data from different parts of the same programme - that is to say that any means of *IPC* increases the overall software system overhead. In some cases, such as ***shared memory blocks*** and ***kernel-level message queues***, the extra overhead is practically negligible, whereas the effect may be significantly larger and more noticeable with methods such as virtual files on the disk (i.e. ***pipes***) or communication via ***network sockets***.

There is also a finite lag introduced into the normal sequence of execution of the software due to the fact that the various programmes run asynchronously. This means that a programme expecting an incoming message would need to poll its *IPC* resources periodically in

anticipation of the arrival of the message. This lag can (in theory) be made short enough to go unnoticed, but this would, in turn, further increase the system overhead and needlessly waste system resources.

It is absolutely critical that the *IPC* mechanism, *IPC* structure and the division of programmes work in harmony, which would require many considerations concerning each of these aspects. As an example, a high-overhead *IPC* mechanism should be used if a smaller number of larger and more complex programmes is desired (and vice versa) in order to reduce overhead. The definition of the *IPC* structure should then be designed such that programmes can easily extract the data relevant to them from interprocess messages and not be bombarded with unnecessary information. Such a definition would need to include things such as the various types of messages to be sent from any programme to any other programme (e.g. message types to communicate telescope coordinates, the current date and time, programme status information, etc.). Naturally, this will depend on the *IPC* mechanism and how the various tasks are divided between the programmes.

Given that many of the programmes would need to output information to the screen and receive input from the user, the *IPC* specification would need to be defined in such a way that each programme receives its required input and that each programme's output is displayed to the user. This would cause more *IPC* traffic and hence more overhead.

Lastly, a modular design has some minimum, irreducible element of duplication of code and functionality. This is most likely to happen in cases where some small, easily determined result is derived from some information shared between various programmes, in which case it is possibly less efficient to perform a calculation in one programme and then send this result to the other programmes requiring it rather than simply performing the same calculation in each programme that requires the result. When the overall structure of the modular design is defined, this kind of situation should be avoided (although some occurrences are inevitable) while also ensuring that there is no overlap between the tasks of the various programmes that may cause serious conflicts or any unsatisfied dependencies among the programmes (i.e. a situation where a particular programme requires a particular piece of information or a particular task to be executed on its behalf, but no programme exists to fulfil this requirement).

4.2 Modular Design in the Context of the ACT Software

4.2.1 Interprocess Communication Mechanism

As is evident from the disadvantages of a modular design, choosing an appropriate *IPC* mechanism is absolutely critical. Moreover, the structure of the software must be closely aligned with the *IPC* mechanism. The Linux platform offers many *IPC* mechanisms, each having advantages and disadvantages [22]. It was decided to use ***UNIX sockets***, through which Internet Protocol (*IP*) packets are exchanged between computers. ***UNIX sockets*** are simple to use, very flexible and universal. Even though they add significant overhead to

the system, the expense is justified because it enables communication between programmes running on different computers without the need for a proxy.

The *IP* works at the internet layer, which works on top of the link layer (the protocol of the physical link, Ethernet in this case) and requires an additional layer (the transport layer) to be useful in the context of *IPC* (see Figure 4.1). The Transmission Control Protocol (*TCP*) is ideal in this case, as connections between computers are established before first use and broken after final use and the accurate transmission of a message from one computer to another is guaranteed provided that the connection is established.

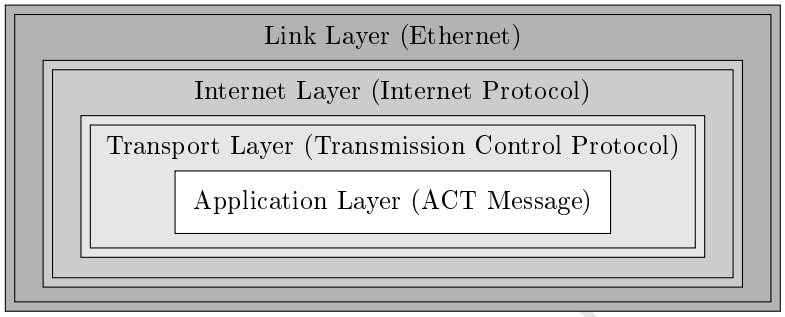

Figure 4.1: A simplified diagram of the layers involved in network access.

TCP/IP networking requires one process (the *server*) to open a ***UNIX socket*** on which it can *listen* for incoming connections and another process (the *client*) to open a ***UNIX socket*** on which it can connect to the *server* process. Such a *server* process can also be used to manage any *client* processes connected to it.

IPC Security

TCP/IP networking is not as secure as many of the other *IPC* mechanisms available on the Linux platform. Fortunately, with the advent of Wide Area Networks (WANs) like the internet and the associated security issues, many effective means and strategies for securing *TCP/IP* networks have evolved. Several factors were taken into account when the security of the *IPC* mechanism was considered. First and foremost, the control computers would not connect to the internet directly and the local network is not publicly accessible. The *firewall* on each control computer can be fine-tuned such that only the programmes of the control software running on one of the other control computers are granted access. Lastly, fairly standard authentication and encryption schemes can be implemented within the programmes with relative ease.

4.2.2 Division of Tasks

In order to compensate for the increased overhead involved with using *UNIX sockets*, a coarser division of tasks between programmes was devised - thus creating a smaller number of larger and more complex programmes. This division, in its current form, is summarised in Figure 4.2 and consists of three components: the main controller, a number of *children* of the

main controller and a number of low-level software interfaces to hardware components. The specifics of each of these programmes will be discussed in subsequent chapters. The main controller manages all *child** programmes and also acts as the *server* for all communication between *clients**.

The *IPC* architecture was devised in parallel with the division of the system tasks into modules. Although the *IPC* was designed from scratch, the result was broadly akin to a Service-Oriented Architecture (SOA) - a design paradigm often implemented with web services. In SOA, internal *clients* require particular input and produce particular output without any "knowledge" of the origin of the input, how the input is produced, the destination of the output or how the output is used. In terms of modular telescope software, this means that a particular programme might need to have the current time and can get it without knowing how the time is read from the hardware. Similarly, the module reading the time can provide the time to the main control programme without any consideration as to which programmes require it or for what purpose.

4.2.3 Graphical User Interfaces

As mentioned before, one of the requirements of the *IPC* architecture is that it must allow for user input to be relayed to the relevant programmes and for the programmes to display their output on the screen. One of two approaches may be used in designing a *graphical user interface* (*GUI*) management scheme:

1. Each module manages its own *GUI*
2. The main controller manages a global *GUI*

The former option is preferable, as the latter would require that the main controller have intimate knowledge of the ideal *GUI* of each *child*. Furthermore, having the main controller as an intermediary for all *GUI* activities would increase *IPC* overhead and increase the response time of the *client* to user input and of the graphical output to changes in the client. However, if each client manages its own *GUI*, they will appear separately, which would make the system appear scattered and thus degrade the user experience.

An ideal solution to this problem exists in the form of ***XLib plugs*** and ***sockets***. ***XLib*** is the library of drawing primitives on the ***X Window System***. All major graphical toolkits (such as GTK+ and Qt) use this library at the most basic level. Furthermore, ***XLib*** drawing commands to a particular window can be diverted to a different window using ***XLib plugs*** and ***sockets***. The main controller can have an ***XLib socket*** for each *client* that has a *GUI* and each client can send all ***XLib*** drawing commands into an ***XLib plug*** instead of directly into a window on the screen. It will appear to the user as if all output is produced by the main controller and all input goes to the main controller. Simplified means of establishing

*The difference between a child and a client is subtle but definite: *child* refers to any computer application that is initiated by the main controller; *client* refers to a programme in the ACT software suite that makes use of services provided by other programmes in the ACT software suite.

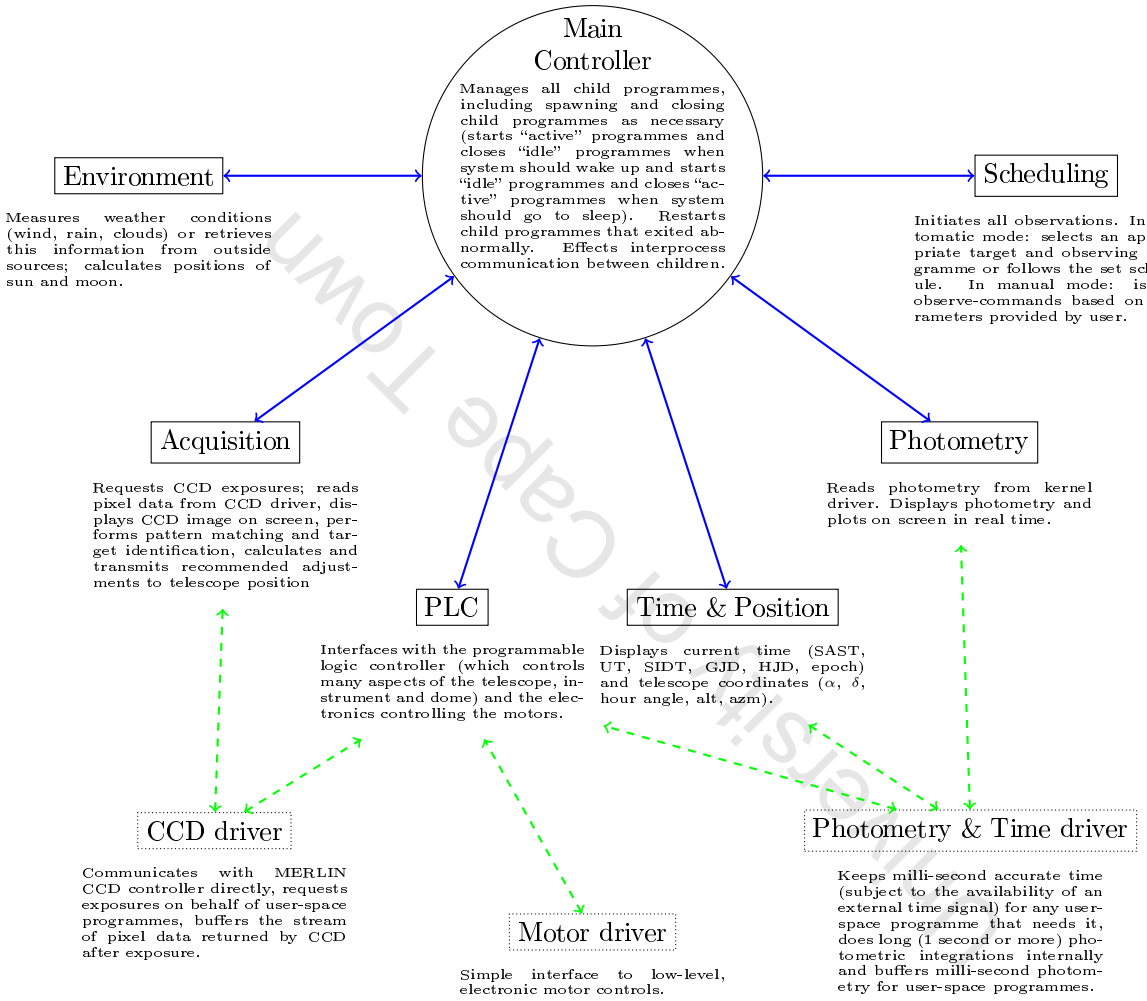

Figure 4.2: The general structure of the suite of programmes and kernel drivers that constitute the ACT software system. The blue arrows represent network connections while the dashed green arrows represent communication via a kernel driver's *character device*.

connections between ***XLib plugs*** and ***sockets*** have been implemented in both GTK+ and Qt. Furthermore, the ***XLib socket*** does not need to reside on the same computer as the ***XLib plug***. Therefore any number of clients on any number of computers can, in theory, have their *GUIs* on a single screen. For security reasons, the *XLib connections* between computers are usually ***tunnelled*** through a Secure SHell (*SSH*) connection.

4.2.4 IPC Structure

The use of ***UNIX sockets*** as the *IPC* mechanism requires the implementation of an *abstracted* set of message structures, implemented (in the case of the ACT software) using C *structs*. These *structs* were defined to be 100 bytes in size, as this is the recommended maximum to ensure accurate data transmission and is in practice much larger than should be necessary given the information to be relayed via the *IPC* mechanism. The first item in the message **struct** is a flag indicating what type of message it is.

During start-up, each *child* reports what information it is capable of providing, what information it requires and what role it plays in observations. Each class of information that needs to be communicated between *clients* has a corresponding **struct** that is similar in structure to the base message **struct** and has exactly the same size (100 bytes). Figure 4.3 shows the structure of this *IPC* message template. Table 4.1 lists the classes of information that the *IPC* provides for. New classes of information may be added to this, as long as they are assigned unique message types and the main controller is recompiled after introducing the new class.

Client requirements and capabilities	What information the *client* requires (time, telescope coordinates, etc.), what information the *client* can provide and what role the programme plays in the system.
XLib socket information	Information required to have the programme's output displayed as part of the main controller's output.
Client status	Used for programmes to report normal operation or an error to the main controller.
Current time	Used to communicate the current local time and current sidereal time as well as several derived quantities, such as the current universal time, heliocentric Julian date and geocentric Julian date.
Telescope coordinates	Used to communicate the current telescope hour angle and declination as well as several derived quantities, such as the current right ascension, altitude and azimuth.
Environmental conditions	Used to communicate the current environmental conditions (relative humidity, cloud coverage, wind speed and direction etc.)
Observational detail	Used to communicate details of the current observation (target coordinates, filter, aperture, integration time, repetitions etc.)

Table 4.1: Classes of information that need to be relayed between programmes.

General Message Structure	
Message Type	integer - 1 byte
Message Content	99 bytes

Figure 4.3: Base data structure of interprocess communication messages. All message types in the specification have a data structure based on this template. The data a particular message type is designed to communicate is stored in the `Message Content` part and all message structures contain an amount of padding depending on the total size of the message content in order to bring the total size of each message structure to 100 bytes. In almost all cases the padding is completely ignored by both sender and recipient.

Client Requirements and Capabilities

As the main controller acts as the central hub for all interprocess communication, each *client* needs to inform the main controller of all data it requires in order to function properly. Similarly, the main controller needs to be aware of what capabilities each *client* adds to the system as a whole - more specifically, what data each *client* is able to provide. Lastly, the controller needs to know what role each *client* has during an observation. This information is relayed to the main controller during the initial handshaking procedure when the main controller spawns a *client*. Even if a *client* requires no data from the other *clients*, is unable to provide any data and doesn't take part in any observations, it still needs to complete the handshaking procedure. The procedure is initiated by the main controller sending a message of type `Client requirements and capabilities` (see Table 4.1) to a newly-started *client* - the content of the message at this point is ignored. The *client* responds by filling in all fields of the message structure appropriately and returns it to the main controller. The data structure for `Client requirements and capabilities` messages is given in Figure 4.4.

Requirements and Capabilities Message Structure	
Message Type	integer - 1 byte
Capabilities	integer - 4 bytes
Requirements	integer - 4 bytes
Observation Stage	integer - 1 byte
Padding	90 bytes

Figure 4.4: Data structure for requirements-and-capabilities messages. All *clients* need to provide these details to the main controller during the initial handshaking procedure. `Capabilities` refers to the classes of data the *client* is capable of providing. `Requirements` refers to the classes of data the *client* requires in order to function. `Observation Stage` refers to those stages of a typical observation the *client* is equipped to complete.

GUI Information

The only information required to establish an ***XLib*** connection between a *client* and the main controller is the unique integer identifier of the ***XLib plug*** and (if the *client* is not

running on the same computer as the main controller) the identifier for the ***X window system*** display on which the plug resides. Each *client* needs to provide this information to the main controller during the initial handshaking procedure, which is initiated by the main controller sending a message with this type to each *client* after it is started. If the *client* has a *GUI* to export to the user, it should fill in all fields of the message data structure (see Figure 4.5) appropriately and return the message to the main controller. If a *client* has no *GUI* to export, the *client* can simply ignore the *GUI* information request message from the main controller or (preferably) indicate this fact by setting the `Plug ID` in the message data structure to 0 and returning the message to the main controller.

GUI Information Message Structure	
Message Type	integer - 1 byte
Plug ID	integer - 4 bytes
Display ID	string - 20 bytes
Padding	75 bytes

Figure 4.5: Data structure for GUI information messages (only for clients with a GUI). `Plug ID` is the unique identifier for the client's GUI plug (virtual window), assigned to it by the windowing system. `Display ID` is the string identifier for the X window system display on which the GUI plug resides.

Status Information

Status information about the *clients* also need to be relayed to the main controller. This should contain a flag indicating whether an error has occurred and, if so, whether the error is fatal and requires that the entire software suite be restarted or whether only the *client* producing the error must be restarted. The error flag is positive if no error has occurred and negative otherwise. Although the *IPC* specification currently makes no provision for particular positive or negative values of the status flag, such a modification could potentially be implemented in future. Instead of padding this **struct** to the appropriate size, the additional space can be used to relay an error message for the user's convenience. Figure 4.6 depicts the data structure designed to relay this information. The main controller should periodically prompt the *clients* to supply this information by sending a message of this type to the *clients*.

Time

A means of communicating the current time (in the several ways of reckoning time used in astronomy) is an absolute necessity. The following times are included in the specification:

- South African Standard time and date (SAST)
- Universal time and date (UT)

Status Information Message Structure	
Message Type	integer - 1 byte
Client Status	integer - 1 byte
Fatal Error	boolean - 1 byte
Restart Client	boolean - 1 byte
Error Message	96 bytes

Figure 4.6: Data structure for client status information messages. If a client receives such a message, it should fill the structure with appropriate values and return the message to the main controller.

- Sidereal time (SIDT)
- Geocentric Julian date (GJD)
- Heliocentric Julian date (HJD)

SAST and UT are stored as a pair of **structs** within the message **struct** - one time **struct** and one date **struct** each. The time **struct** stores the hours, minutes, seconds and milli-seconds as integers (see Figure 4.7). Similarly, the date **struct** stores the year (it is assumed to be *A.D.*), month of the year (where 1 represents January), day of the month (1 is the first day of the month) as integers (see Figure 4.8). SIDT is also stored in a time **struct** . GJD and HJD are stored as 64-bit floating-point numbers. A summary of the time message data structure is given in Figure 4.9.

Time Data Structure	
Hours	integer - 1 byte
Minutes	integer - 1 byte
Seconds	integer - 1 byte
Milli-Seconds	integer - 2 bytes

Figure 4.7: Data structure for storing and communicating time information.

Date Data Structure	
Years (since 0 A.D.)	integer - 2 bytes
Month of the year	integer - 1 byte
Day of the month	integer - 1 byte

Figure 4.8: Data structure for storing and communicating date information.

Telescope Coordinates

Telescope coordinates must be communicated between *clients* regularly. The following quantities are stored within the coordinates **struct** (some of which are stored merely for conve-

Requirements & Capabilities Message Structure	
Message Type	integer - 1 byte
Local Time	Time **struct** - 5 bytes
Local Date	Date **struct** - 4 bytes
Universal Time	Time **struct** - 5 bytes
Universal Date	Date **struct** - 4 bytes
Sidereal Time	Time **struct** - 5 bytes
Geocentric Julian Date	floating-point days - 8 bytes
Heliocentric Julian Date	floating-point days - 8 bytes
Padding	60 bytes

Figure 4.9: Data structure for time messages. All clients requiring one or more of the quantities contained within this structure will receive a message with this structure, containing the current time in all of the forms listed above.

nience):

- Hour angle (HA)
- Right Ascension (RA)
- Declination (Dec)
- Epoch
- Altitude (Alt)
- Azimuth (Azm)

All the values stored in the coordinates **struct** are stored as 64-bit floating-point numbers. The telescope coordinates data structure is depicted in Figure 4.10.

Telescope Coordinates Message Structure	
Message Type	integer - 1 byte
Hour angle	floating-point hours - 8 bytes
Right Ascension	floating-point hours - 8 bytes
Declination	floating-point degrees - 8 bytes
Epoch	floating-point years - 8 bytes
Altitude	floating-point degrees - 8 bytes
Azimuth	floating-point degrees - 8 bytes
Padding	51 bytes

Figure 4.10: Data structure for telescope-coordinates messages. All clients requiring one or more of the quantities contained within this structure will receive a message with this structure, containing the telescope coordinates in all of the forms listed above.

Environmental Conditions

The `Environmental Conditions` message structure (as depicted in Figure 4.11) contains data on environmental conditions which determine whether or not it is safe for the telescope to operate. The message structure additionally contains a flag the `Active/Idle` flag which is true (`Active`) if all aspects of the environment are safe for the operation of the telescope and false (`Idle`) otherwise.

Telescope Coordinates Message Structure	
Message Type	integer - 1 byte
Active / Idle	integer - 1 byte
Relative Humidity	integer percentage - 1 byte
Cloud Coverage	integer percentage - 1 byte
Rain	integer boolean - 1 byte
Altitude of Sun	floating point degrees - 4 byte
Wind Velocity	integer (km/h) - 2 bytes
Wind Direction	integer degrees (0°-360°) - 2 bytes
Seeing	integer arcseconds- 2 bytes
Padding	51 bytes

Figure 4.11: Data structure for messages relaying data on environmental conditions.

Observational Parameters

Most *clients* would need information concerning the current observation. This was implemented such that two closed loops (the target acquisition cycle and the data acquisition cycle) are formed by the *clients* having tasks to complete at some point during an observation, with the message being passed from one *client* to the next (in the sequence dictated by the *IPC* specification) thus signalling the appropriate time for a *client* to complete its assigned task. This implementation means that each *client* needs to know only what role it plays during an observation and that no programme (not even the main controller) needs to know what the proper sequence of events is - the sequence is inherent to the *IPC* specification.

An observation message originates from the *client* responsible for scheduling, is sent to each child that forms part of the target acquisition cycle and is returned to the scheduling *client*. The scheduling *client* then sends the observation message to the *clients* comprising the data acquisition cycle, after which the message is returned to the scheduling *client*. In both cases, the observation message is returned to the scheduling *client* whether or not the observation was completed successfully. When a *client* receives an observation message, it completes all tasks (if any) it was assigned to complete, fills in all necessary information (if any) into the message **struct** and returns the message to the main controller.

The *IPC* specification enforces the proper sequence of activities of the various *clients* by

Observation Message Structure	
Message Type	integer, 1 byte
Target Name	string, 20 bytes
RA	floating-point hours, 8 bytes
Dec	floating-point degrees, 8 bytes
Epoch	floating-point years, 8 bytes
Auto/Manual mode	boolean, 1 byte
Observation Stage	integer, 1 byte
Filter No.	integer, 1 byte
Aperture No.	integer, 1 byte
Star (or sky)	boolean, 1 byte
Focus position	integer, 2 bytes
Integration Time (ms)	integer, 4 bytes
No. of Repetitions	integer, 4 bytes
Padding	40 bytes

Figure 4.12: Structure of interprocess communication message containing all data relevant to a particular observation.

means of an `observational stage` flag. The flag is an integer where the current observational stage is derived from the binary representation of the integer. The least significant bit (bit 1) represents the scheduling stage, the next least significant bit represents the stage following the scheduling stage, etc. Only one bit in the binary representation of the integer may be 1, the others must be 0. The various stages comprising the observation cycle are given in Table 4.2. The main controller keeps track of the observational stage by advancing the non-zero bit of the observation stage flag by one position (this is tantamount to multiplying the integer by 2). Once the position of the non-zero bit exceeds that of the last stage in the observation cycle, the flag is reset to 1 and the message is returned to the scheduler.

The observation cycle is shown in Figure 4.13.

4.2.5 IPC Forward Look

The *IPC* specification outlined above should be fairly simple to expand and update. However, if any of the existing **structs** need to be changed or if any of the defined capability/requirement identifiers need to be changed, every programme will need to be updated.

New capabilities can be added to the specification by assigning a unique capability number to it and defining a structure designed to contain the relevant information without modifying or even recompiling any of the other programmes. Similarly, unused capability numbers may be removed from the specification along with their defined **structs** without affecting any of the other programmes. Future revisions of the *IPC* specification should cater for situations where a *client* programme has a number of sets of acceptable requirements and capabilities (as opposed to the current specification, which allows for only one set of

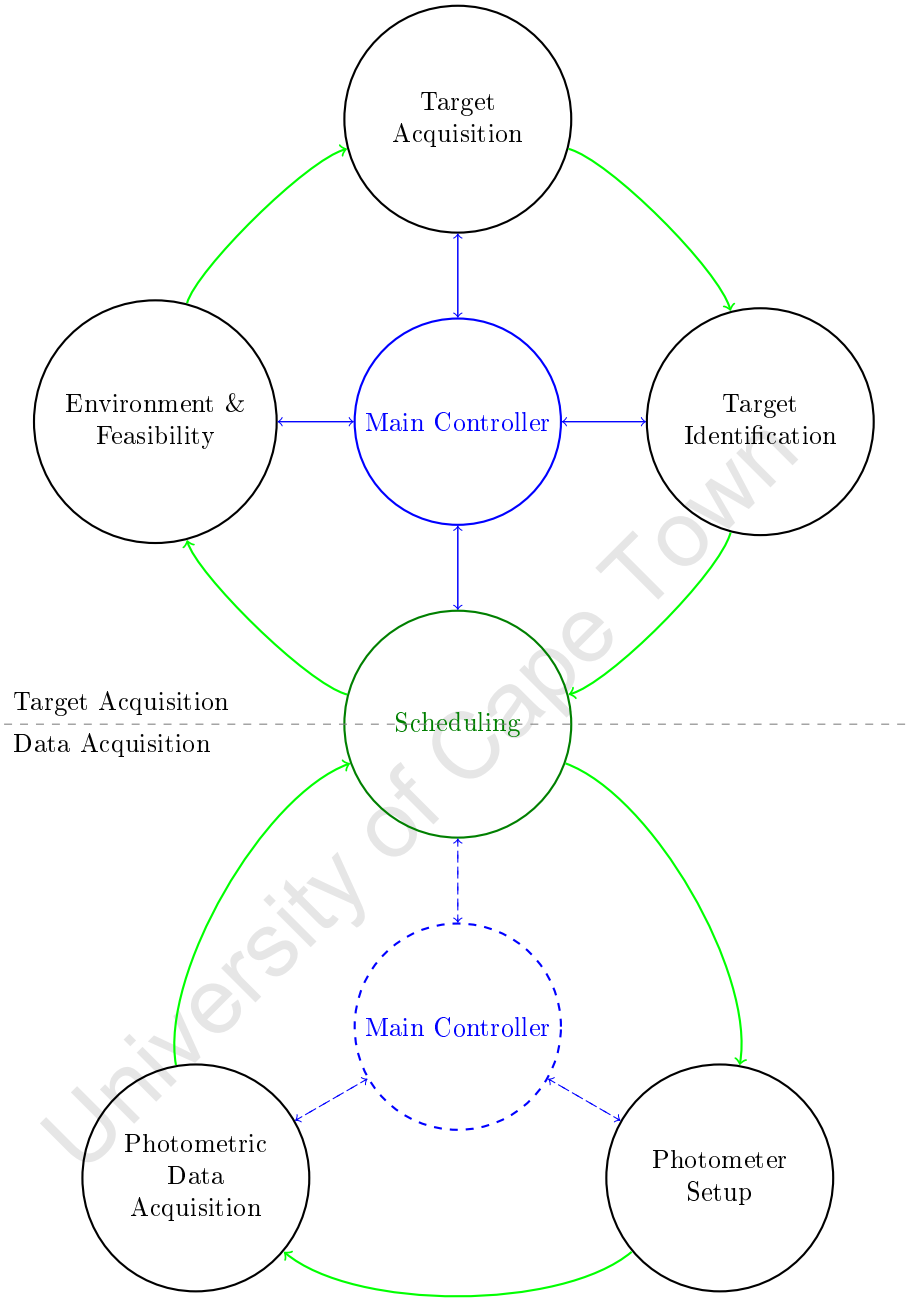

Figure 4.13: Diagram of the ACT observation cycle. The dashed lines indicate messages being passed through the main controller between consecutive stages of the cycles.

Bit	Name	Action
1	Target Acquisition Scheduling	Select a target to be observed (either under manual control or automatically).
2	Environment	Check environmental conditions (humidity, clouds, proximity to Moon).
3	Target Acquisition	Slew to target and (optionally) start tracking. Additionally, if the system is in automatic mode, the acquisition mirror is moved in-beam.
4	Target Identification	In manual mode: Return message to main controller. It is assumed that the user will centre the target on the aperture. In automatic mode: Do pattern matching. If target is centred on the aperture, return message to main controller; otherwise calculate new coordinates and reduce stage such that #3 is repeated.
5	Data Acquisition Scheduling	If target acquisition was successful, continue to data acquisition stage otherwise log an error and either halt observations or choose a different target depending on the nature of the error.
6	Photometer Setup	Set filter and aperture; In automatic mode: move acquisition mirror out-beam and open instrument shutter.
7	Photometric Data Acquisition	Collect photometry with integrations and repetitions specified in the observation message (0 repetitions means repeat forever).

Table 4.2: Summary of the observation stages as provided for in the *IPC* specification.

requirements and one set of capabilities). An example of where this would be useful is if a programme can accept either the local time or the universal time as its input - in the current specification, such a programme would have to specify both as a requirement even though only one will be used and this sort of behaviour might hamper efforts to resolve all dependencies within the suite of programmes. Lastly, the current *IPC* specification does not allow for easy expansion of the observational cycle - new stages may be appended to the list (although this would require that the control programme be recompiled), but if new stages are added within the current cycle (i.e. between stages 1 and 6), all programmes will have to be recompiled.

Chapter 5

Main Controller

This chapter describes the operation of the main controller (controller for short). Section 5.1 briefly outlines the purpose of the controller. Sections 5.2 and 5.3 explain how the controller starts *child* processes and the initial *IPC* handshake. Sections 5.4 and 5.5 outline the configuration files required by the controller. Sections 5.6 and 5.7 explain in greater detail the role of the controller in terms of *IPC*, managing clients and *graphical user interface* (*GUI*) management. Section 5.8 explains how the controller enables remote access and section 5.9 outlines potential future improvements of the controller.

5.1 Introduction

In chapter 4 it was explained that the controller is the central hub or "switchboard" of all interprocess communication (*IPC*). The necessity of this hub allows for several interesting potential developments such as (in broad terms) using the main controller to manage all applications that form part of the software suite. A diagram of the controller's intended inputs, tasks and outputs (in broad terms) is given in Figure 5.1.

In general, a central controller could include tasks such as *spawning* the various *child* applications upon start-up, managing the sections of the screen devoted to each *child* producing graphical output, regularly polling the *children* to verify their continued successful execution, restarting *children* that exited abnormally or reported errors which require the *child* or *children* to be restarted, checking that the dependencies of all children are met, etc. The set of *children* the controller can manage should preferably not be static, but arbitrary and dynamic.

5.2 Spawning a Child Process

On UNIX systems, this task has to be done in two steps. The first step is to issue the *fork*[*] command. The *fork* function duplicates the calling programme exactly, including the

[*] `http://linux.die.net/man/2/fork`

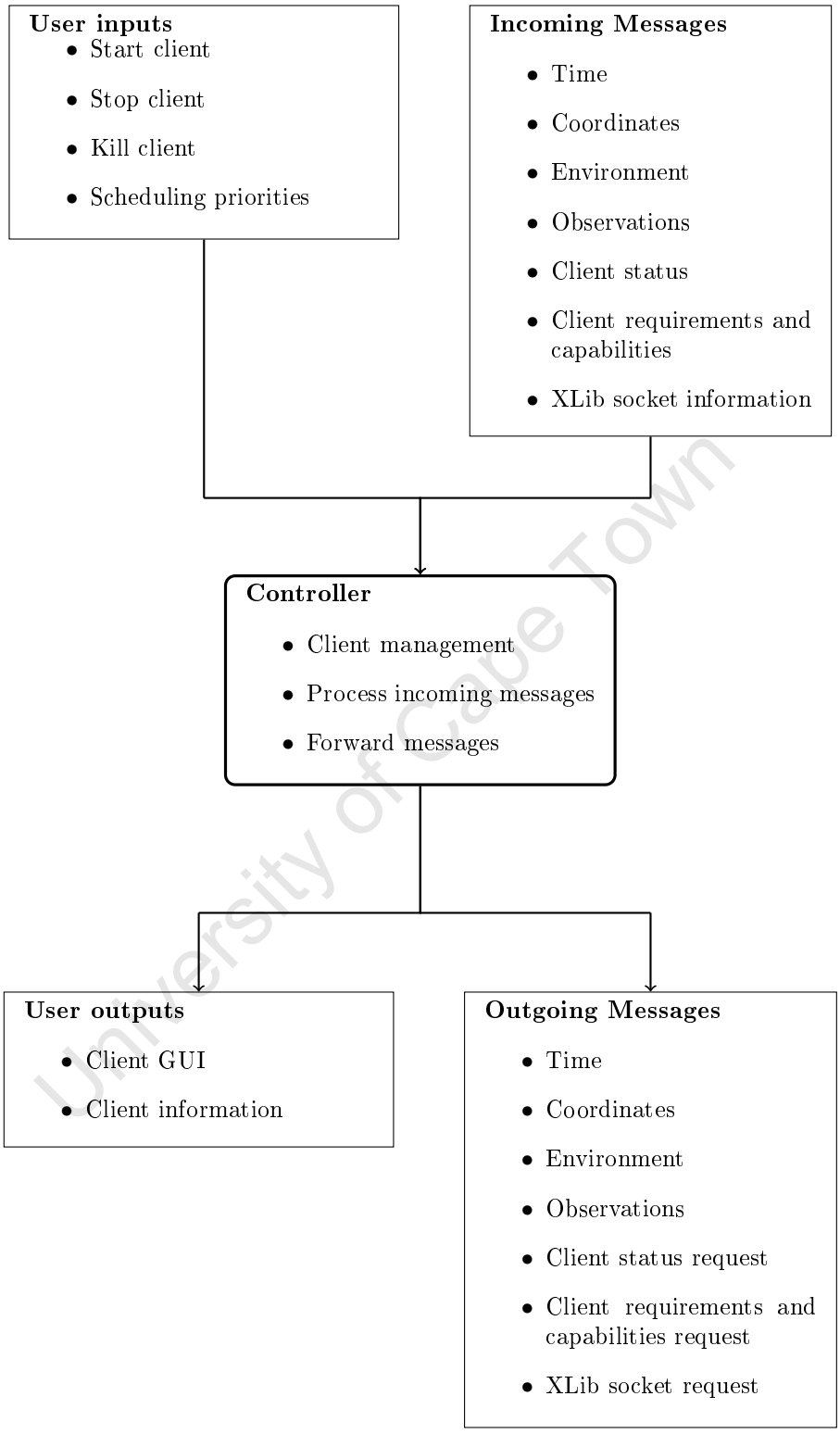

Figure 5.1: Diagram of controller inputs, tasks and outputs.

computer memory that has been allocated to the calling programme, file descriptors, etc. except, of course, if the call was unsuccessful (in which case *fork* returns a negative number). The original programme then becomes the parent of the newly-*spawned* copy of itself and both the *child* and the *parent* continue execution from the point where *fork* was called - with one minor difference: The result of the call to *fork* in the *parent* process returns the process identifier (PID) of the *child* process and the result is 0 in the *child* process. This difference allows us to distinguish between the *parent* and *child* processes in the source code.

In the case of the *parent* (the controller), no further action is strictly necessary and (aside from a few minor tasks), the normal thread of execution of the controller resumes.

After the *child* is *spawned*, it is still an exact copy of the controller. The programme intended to run in this thread has to be started at this point. There are several ways of doing this, which can be divided into two broad classes, namely pausing the current thread of execution while the intended client programme is executed or replacing the copy of the *parent* (i.e. the controller) with another programme. Generally, the one major advantage of pausing the current thread of execution while the *client* is executed, is that the paused programme is returned to after the client exits - which is actually a disadvantage in this regard, as we would need to kill this process after the *client* programme exits. A further disadvantage to this approach is that the resources of the duplicate of the controller are retained while the *client* is running. Therefore, in this case, the alternative is much more appropriate - i.e. replacing the current process with the intended *client* programme. This can be achieved through any of the *exec*[†] family of functions. All necessary command-line arguments may be specified as part of the call to any of the *exec* family of functions.

5.2.1 Command-line arguments to clients

A programme *spawned* by the controller would necessarily require the *hostname* of the computer on which the controller is running (even if the child is run on the same computer as the controller) as well as the network *port* which the controller is monitoring for incoming connections. As the *child* and controller will be unable to communicate with each other without these two data, they should either be specified as command-line arguments to the *child* executable or be specified in a configuration file accessible by the child program. The former option is preferable because the controller would need to have the hostname and port anyway. Furthermore, some clients may require a configuration file in order to start up. It was decided to implement the specification of the global filesystem path and filename of a global configuration file as a command line argument to the children. This way, each *child* would by design have the necessary information to start communicating with the controller and have access to at least the global configuration file.

[†] `http://linux.die.net/man/3/exec`

5.3 Client Start-up

When a *client* is started, it would initially have no means of communicating with the controller. The first course of action should be to establish a network connection with the controller. The information required to do so will have been passed to the *client* as command-line arguments. Once the arguments are processed, the *client* can start establishing the link. There is a proper handshaking procedure that must be followed in order to completely establish the basic connection. Should the *parent* not properly follow this procedure, the *client* should log an error and exit. Similarly, if the *client* does not follow the procedure correctly, the controller should stop the *client* and log an error. Once the connection is established, the *client* should parse the global configuration file. If an error occurs while the global configuration file is parsed, the client should exit.

After establishing a connection with the controller and all necessary configuration options extracted from the configuration file(s), the *client* should be ready to proceed with its normal operations. Apart from completing the tasks the *client* was designed to do, the *client* should also regularly check the network connection with the *parent* - that includes removing all waiting messages from the queue, processing them and (if necessary) responding appropriately.

There are two issues that need to be jointly addressed by both the *client* and the controller whenever the *client* is started, they are the specification of ***XLib plug*** and ***XLib socket*** information so the *client's* output can be exported to the screen and the specification of the capabilities and requirements of the client. Either matter may be addressed first and in both cases, the controller initiates the process. If the client receives a `GUI information` message, the correct response would be to enter the appropriate information into the corresponding fields of the message structure (once this information is known) and return the message. If the client receives a `Client requirements and capabilities` message, the appropriate response is to enter its capabilities and requirements (as laid out in the system specification, see Chapter 4) into the corresponding fields of the message structure and return the message.

5.4 Global Configuration File

The basic structure of the global configuration file is quite simple: Comments start with the `#` character and everything else should be in the form `key = value`. Any leading or trailing white space should be ignored, as well as any white space between `key` and `=` and between `=` and `value`. The keys are identifiers for the properties programmes need to extract from the file and the values are the properties. New key-value pairs may be added to the file in order to cater for potential expansion of the software, provided that the new keys are unique. This rule does not apply in situations where, by design, a particular, unique key can have several values associated with it. In order to keep the global configuration file short and easy to maintain, any programmes requiring large amounts of configuration data should be assigned one or more separate configuration files. These files should be linked to from

the global configuration file using key-value pairs, where the values should preferably be the global filesystem paths to the additional configuration files or otherwise a filesystem path the intended programme will understand.

5.5 Client Configuration File

Each *client* that the controller is intended to manage should have an entry in the client configuration file, which is linked to from the global configuration file (see section 5.4). This file is similar in structure to the global configuration file (comments start with a #, configuration options are specified by key-value pairs) with one modification: The file is divided into a number of blocks, with each block specifying the options of a single *client programme*. A block is started by specifying a unique identifier for the *client* within square brackets (`[IDENTIFIER]`). This identifier is only used within the controller and is an alphanumeric string optionally also containing dashes ('-') and underscores ('_'). Even though any unique identifier may be chosen, the identifier should be descriptive of the *client* it represents. Any key-value pairs in the file refer to the last specified *client*. Each block should contain the global filesystem path to the *client* executable and the name of the *client* executable - all other information is optional. Should a *client* need to export any information to the screen (by means of an XLib socket), the rectangular section of the screen (in the form *column start, column end, row start, row end*) should be specified as part of that *client's* block. Otherwise, a *client's* inability to export information to the screen can be indicated by setting either the *column start* equal to *column end* or the *row start* equal to *row end* - this is the default behaviour. The `activetime` key specifies whether the *client* should be active when the software system is idle (`idle`), active (`active`), either active or idle (`both`) or neither active nor idle (`neither`) - see §5.6. In future versions of the controller, *client* executables may be started on different computers. Such a client should have the *hostname* of the computer on which it is to run specified as part of its block.

```
[TIMECOORD]
executable = time_coord
path = /home/actuser/act_control/time_coord/
host = localhost
guicoords = 0,3,0,1
activetime = both
```

Listing 5.1: Example definition in the client configuration file.

5.6 Client Management

One of the cornerstones of the control software suite is the *client* management options enabled by the modular approach taken. Most notably, the ability to start and close *clients*

gracefully, to kill *clients* that have stoppped responding and to restart *clients* that exited abnormally.

As mentioned before, each *client* has a unique identifier (as specified in the *client* configuration file). Each *client* has a corresponding button on the *controller's* window, which doubles as a status light for the *client* - see Figure 5.2. When the button is clicked, information concerning that *client* is displayed in a popup window along with buttons for starting, closing and killing the *client*. Should a *client* perform unusually, it may be necessary to restart the *client* by first closing then starting the *client*. This can also be done in order to replace a currently running *client* with a newer version. The `Kill` button is useful if the *client* stops responding or if it could not be closed successfully. Should *killing* the *client* not successfully stop the *client*, the computer will likely have to be rebooted. This function has been in place for several months (after extensive testing), but it has not yet been necessary to reboot the computer due to a failure of one of the clients.

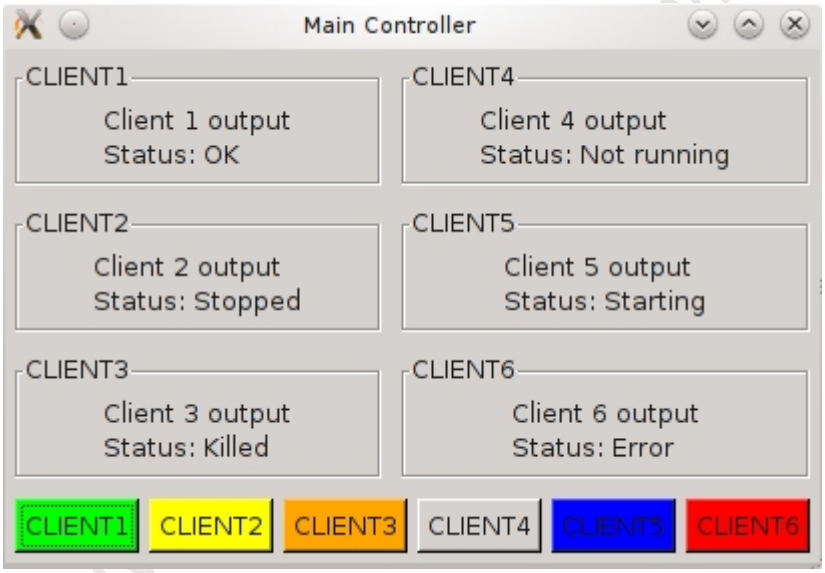

Figure 5.2: Mock-up of controller the status of the various clients.

The controller is also equipped to restart *clients* that exit abnormally. Such a situation is identified by catching all ***SIGCHLD signals*** - these ***signals*** are sent to the *parent* of a *child* process when it exits. Whenever such a signal is caught, the controller checks each *client's* execution status against its previous execution status (saved within the controller) until it finds a *client* that is no longer running, but should be. The controller then restarts this *client*.

5.6.1 Active and Idle Mode

The telescope software always operates in either `active` or `idle` mode, depending on the environmental conditions. The telescope software is `idle` when it is not possible to observe

astronomical targets due to environmental conditions (e.g. during the day or while it is raining - see Chapter 10.3) and is `active` when it is possible to observe.

Clients that have a role to play during observations must be running when the software is in `active` mode, but need not be running when the software is in `idle` mode. Similarly, some *clients* may not be necessary during observations (such as off-line data reduction programmes) and should only be running when the software is in `idle` mode. Other *clients* may be needed during both `idle` and `active` states.

The controller is able to start and stop the relevant *clients* while the software is running. Although the user may start and stop individual *clients* at will, whenever the software switches between modes the controller starts and stops *clients* as appropriate.

The `activetime` parameter in the client configuration file specifies which *clients* need to run in which modes. Additionally, *clients* may be labelled in the client configuration file as running in neither `active` nor `idle` mode, in which case the controller will neither start nor stop that *client* when the software mode changes.

5.7 GUI Management

A *GtkTable*[‡] object was chosen to contain the graphical content of all *clients*. Therefore, each *client* is allocated a rectangular section on the screen (as specified in the *client* configuration file) which contains the *GtkSocket*[§] created for that *client*. Such a rectangle may span several rows and/or columns in the *GtkTable* object, however this does not necessarily mean that the *client's* output will occupy a larger section on the screen than that of a client spanning fewer cells of the *GtkTable* due to the size allocation mechanism of GTK. The ability to arbitrarily specify the on-screen position of *client* output may seem like an unnecessary function, however it does add to the flexibility of the system and did immensely simplify and speed up the task of finding space on the screen for all necessary display items.

As stated in section 5.5, if a *client's* on-screen allocation is omitted in the *client* configuration file, the default is used: the trivial allocation from $(0, 0)$ to $(0, 0)$. The controller interprets any allocation where either the start column is the same as the end column or the start row is the same as the end row as no allocation, in which case the controller will not create a *GtkSocket* for the *client's* output and will not request the *client's* ***XLib plug*** identifier. This should be the case only for *clients* that have no graphical output to be displayed on the screen. If a *client* producing graphical output is given no valid allocation on the screen, that output will not be displayed and no error will be reported.

It should be noted that the controller will automatically allocate the space specified in the client configuration file to the corresponding *clients* - even if that space is already occupied by another application. This will cause several *GTK* errors printed to the console, the visual

[‡]A *GTK+* graphical element that may contain several other graphical elements and is used only to structure the GUI by placing the graphical elements it contains on a rectangular (but not necessarily regular) grid.

[§]The GTK+ graphical element that allows the *GUIs* of the *clients* to be displayed as part of the controller's *GUI* using ***XLib plugs*** and ***XLib sockets***

effect will be one *client* overlaid on top of another. This is obviously an undesirable effect and would probably have been caused by an incorrect space allocation in the *client* configuration file. Future versions of the controller should check for such errors and either correct for it automatically (if possible) or log an error and alert the user.

The space allocation for *client* output is done when the controller starts up and parses the client configuration file. This implies that the allocation is arbitrary, but not dynamic. In other words, the entire suite of applications needs to be restarted when the allocations for the *clients* are modified in the client configuration file. As this is an undesirable effect, although relatively minor, a means of dynamic allocation should be implemented.

5.8 Remote Access

Remote access to the system is enabled using the *NX* software package (see 3.3.4). Initially, a scheme was conceived by which the ***XLib*** communications between the telescope control software and the ***X*** server on the control computer could be diverted to the ***X*** server on a remote computer, thus allowing the remote user to use the control software as if he/she were at the telescope control computer. However, preliminary testing showed that this scheme would have been cumbersome to implement effectively and it was decided to use *NX* instead, since *NX* provides all the functionality required of the remote access facility of the controller.

The *NX* server does not merely divert the stream of ***XLib*** communication, but in fact replicates the user output produced by the software (sending a full copy of all communications to each connected *NX* client) and multiplexes user input to the software from all connected *NX* clients. The advantage of this is that several users may control the telescope at the same time, although the *NX* server can be configured to limit the number of authorised users that may view and/or control the software simultaneously. In the context of the ACT software, it is best to configure the *NX* server such that multiple simultaneous access is forbidden.

NX also provides several features that are convenient (in terms of the remote access facilities of the software) and some that are unfortunately completely unnecessary (such as the ability for printers to be shared between *NX* servers and clients). A particularly convenient feature is the ability to stream audio between the *NX* server and the *NX* clients - a feature that would have been implemented anyway so that remote users can monitor the telescope by the sound it makes.

The only disadvantage to using *NX* in this case is that it provides features that are not required and it therefore consumes unnecessary amounts of system resources (CPU, memory, etc.). However, performance tests were conducted and it was found that the *NX* server programme consumes comparatively little system resources and that its use does not adversely affect the performance of the telescope control software.

5.9 Forward Look

As mentioned in the previous sections, future versions of the controller should be capable of starting *clients* on different computers, checking unmet dependencies among the *clients*, checking for overlaps between *client* output sections and dynamically managing the *client* output sections on the screen.

Future versions of the software should also have a means of starting *clients* that aren't specified in the client configuration file. At the moment, if a new *client* needs to be added to the system, the entire software suite needs to be restarted.

Chapter 6

Scheduler

The scheduler is responsible for initiating each observation command that is sent to the other programmes in the ACT software suite. In manual mode, this is done upon the user's request; in automatic mode an observation command is initiated according to the scheduler's built-in algorithms that dictate when and how targets are to be observed.

This chapter outlines the development and implementation challenges encountered as well as the operational features of the software and some operational features of the scheduler. Section 6.1 outlines issues related to differences between automatic and manual mode and how other ACT *clients* should recognise commands issued in either mode. Section 6.2 describes the scheduler's configuration files and section 6.3 describes observing queues. Section 6.4 outlines potential future upgrades to the scheduler.

6.1 Automatic and Manual Mode

From the user's perspective, switching between automatic and manual mode is as easy as flicking a switch. At the software level, the situation is more complex. In automatic mode the scheduler can simply issue an observation message and trust that all clients are performing their duties correctly (unless an error is reported). In manual mode the user must have the opportunity to operate all elements of the telescope and instrument in preparation for target acquisition when switching from one target to another or switching from the target to the sky and to operate all elements of the telescope and instrument in preparation for data acquisition.

The problem was eventually solved by dividing the observation stages (which mimic the tasks that must be performed by all components of the ACT software in order to observe a target) between two loops instead of having them all in sequence in one loop (see Figure 4.13). With this revision of the *IPC* specification, the scheduler can pause the sequence of operations after the `target acquisition cycle` is complete and before the `data acquisition cycle` commences in order to give the user time to make the appropriate adjustments to the telescope and instrument during manual operation. Similarly, the scheduler can pause between

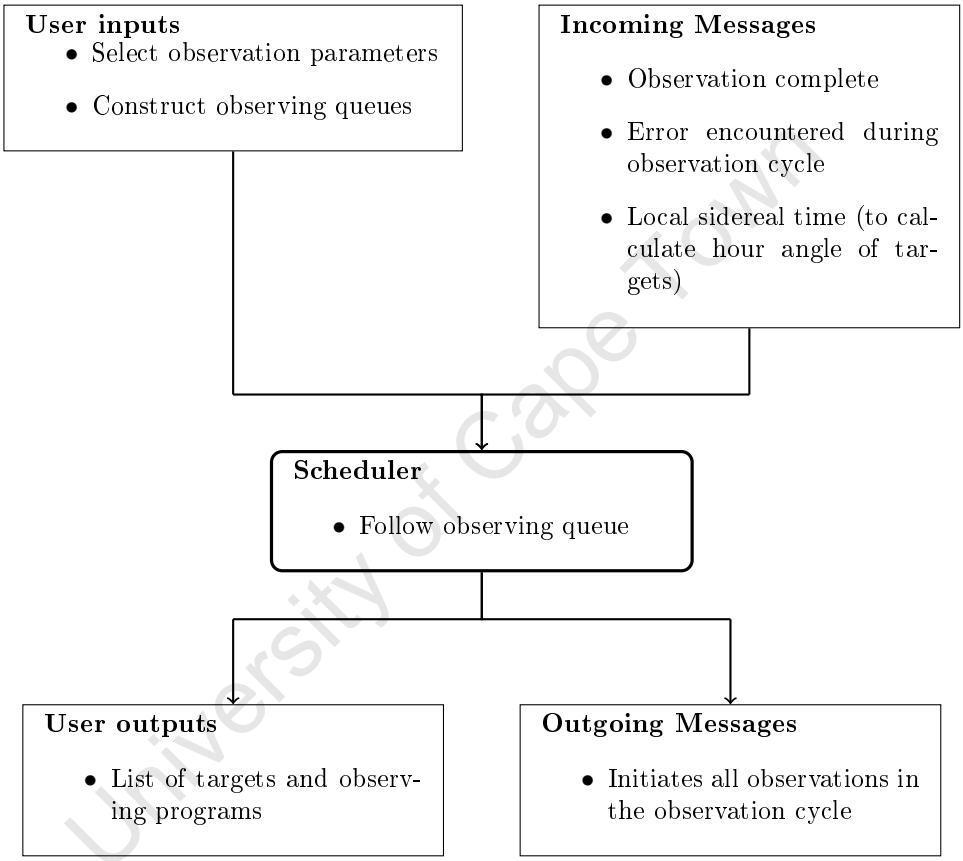

Figure 6.1: Diagram of scheduler inputs, tasks and outputs.

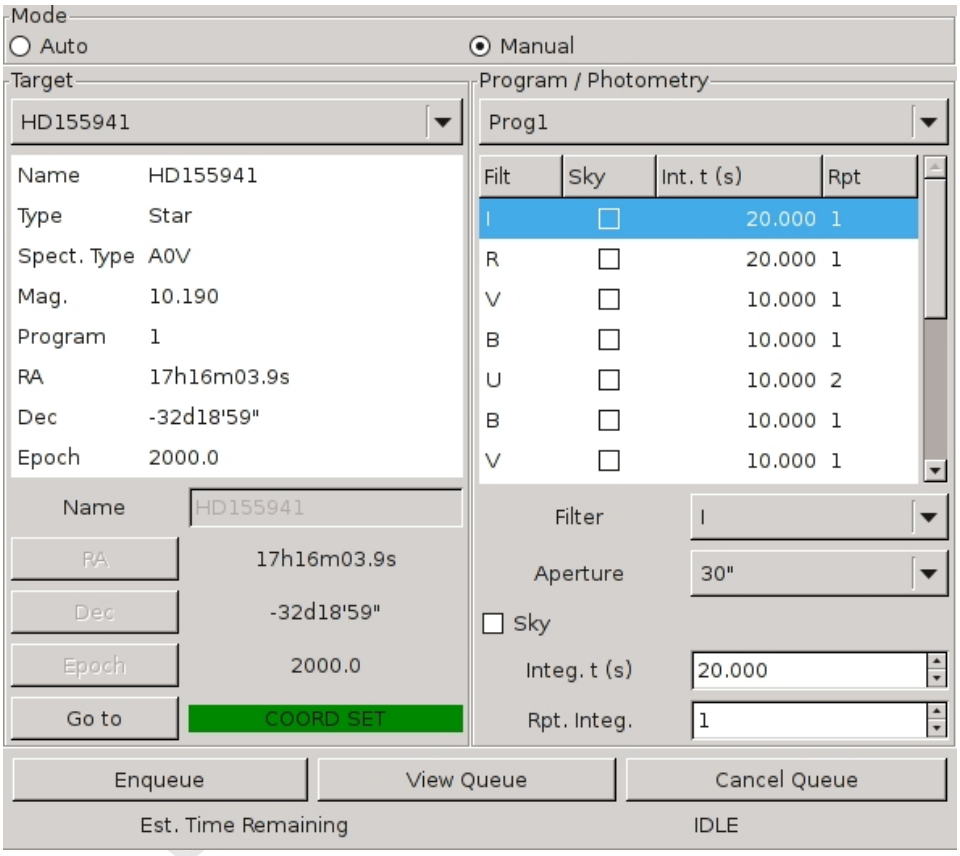

Figure 6.2: The scheduler user interface.

Key	Description	Default
`filters`	List of filters	*CRITICAL*
`apertures`	List of apertures	*CRITICAL*
`soft_lim_W`	Software Western limit of telescope	*CRITICAL*
`soft_lim_E`	Software Eastern limit of telescope	*CRITICAL*
`soft_lim_N`	Software Northern limit of telescope	*CRITICAL*
`soft_lim_S`	Software Southern limit of telescope	*CRITICAL*
`min_elevation`	Recommended minimum elevation	0.0
`programmes_file`	File containing convenient observing programs	*CRITICAL*
`stars_file`	File containing data of frequently-used targets	*CRITICAL*
`macrofile`	Name of file containing an observing macro*	None
`autostart`	Initiate macro (if specified) on start-up	No

Table 6.1: List of keys recognised by the scheduler in the global configuration file. The entries where the default value is listed as being *CRITICAL* indicate those keys which may not be omitted.

the `data acquisition cycle` related to a particular target and the `target acquisition cycle` of the next target to give the user the chance to make appropriate adjustments to the telescope and instrument. The scheduler does not pause during automatic operation, since the various components of the ACT software will make the appropriate adjustments to the telescope and instrument.

Furthermore, the scheduler provides a "go to" feature that may be used to have the telescope automatically slewed to the selected target, but no other tasks (such as setting various parameters of the instrument) performed - these tasks are left to the user. When the user issues a "go to" command, the scheduler sends an observation command on the `target acquisition cycle` but does not start the `data acquisition cycle` after the target has been acquired.

6.2 Configuration Files

6.2.1 Global Configuration File

The keys recognised by the scheduler in the global configuration file are listed in Table 6.1

The list of filters is a space-separated list of human-readable filter-names and (similarly) the list of apertures is a space-separated list of human-readable aperture-names (see Listing 6.1).

```
filters = U B V R I CLR CLR CLR CLR CLR
apertures = 750" 90" 60" 45" 35" 30" 25" 20" 15"
```

Listing 6.1: Example filter configuration lines in the global configuration file.

The list of filters should contain exactly as many entries as there are slots on the filter wheel, whether or not all slots are occupied. The same rule holds for the number of entries

*A macro in the context of the scheduler defines an observing queue which the scheduler should initiate upon start-up, thus implementing automated operation of the telescope.

in the list of apertures.

The scheduler will not allow any observation of a target that falls beyond the telescope software limits. During manual operation, the scheduler will warn the user if the target is too low in the sky - as specified by the `min_elevation` parameter.

The `programmes_file` parameter should have a value that is the global filename of a file containing details of frequently-used observing programmes. Similarly, the `stars_file` parameter should have a value that is the global filename of a file containing details of frequently-observed targets. The `macro_file` parameter should have a value that is the global filename of a file containing details of the targets (from the stars configuration file) to be observed on a particular night (in order) as well as which apertures (from the list of apertures) and observing programmes (from the programmes configuration file) to use with each target. The `autostart` parameter only has an effect when a macro file is specified and is ignored otherwise. The `autostart` parameter makes the scheduler start in automatic mode and starts the macro specified in the macro configuration file (see §6.2.4). If a valid macro configuration file is specified, but autostart is disabled, the scheduler will prompt the user, asking whether the macro should be ignored or the scheduler should initiate the macro. This function is useful if an observer simply needs to ensure that the software starts up correctly, then lets the software go about its business.

6.2.2 Stars file

The stars file is used to store the details of frequently-observed targets. Furthermore, if a particular target is referenced within the macro file (see §6.2.4), it must have a corresponding entry in the stars file.

The stars file has a structure similar to that of the controller's client configuration file (see §6.2.1). A comment line begins with a `#` and any leading or trailing space on any line is ignored. The file is divided into blocks, with each block representing a frequently-observed target. The start of a block is indicated by a descriptive name for the target that block represents, encapsulated in square brackets (`[` and `]`). The star's HD number, E region number or other designation in some well-known catalogue should be used, as the user should be able to identify the target uniquely from this identifier and the target may be referenced in other files (such as the macro configuration file) using this identifier.

Each block consists of a number of key-value pairs (as in the controller's client configuration file). The keys recognised by the scheduler are listed in Table 6.2.

Only the target's right ascension, declination and the epoch of the coordinates need to be specified in every block. Should the right ascension, declination or epoch be omitted in a particular block, that entire block will be ignored. For the moment, all the other parameters are merely for the user's convenience. Both the programme identifier specified for the `prog` key and the aperture identifier specified for the `aper` key need to be exactly identical to their definitions in the programmes configuration file and apertures option in the global configuration file, respectively.

Key	Description
type	Type of target (e.g. δ-Scuti, Cepheid)
spect	Target's spectral type
C1	Recommended comparison star
C2	Recommended comparison star
mag	Magnitude of star
prog	Identifier of recommended observing program
aper	Identifier of recommended photometric aperture
ra_h	Target's right ascension (in the format hh:mm:ss.s)
dec_d	Target's declination (in the format dd:mm:ss)
epoch	Epoch in which target's coordinates were specified (e.g. 2000.0)

Table 6.2: Keys contained in the stars configuration file.

Only the keys specified in Table 6.2 are recognised by the scheduler. In future versions, all additional information may be stored in an abstracted way (such as in a string) and displayed along with the other information for the user's convenience.

6.2.3 Programmes file

The programmes configuration file defines an arbitrary number of observing programmes. These observing programmes specify sequences of filters, integration times, etc. and can (in theory) be applied to any star. For example, the user may define an observing programme appropriate for observing rapidly varying stars, which could consist of a 10 s integration on the target in each of the Johnson UBVRI filters followed by a 10 s integration on the sky.

As in most of the other configuration files, the file is divided into blocks. A comment line begins with a # and any leading or trailing space on any line is ignored. The blocks span from one programme identifier to the next, where a programme identifier is given within square brackets. The programme identifiers should be meaningful to the user and may contain any combination of alpha-numeric characters, as well as hyphens (-) and underscores (_).

Each block contains a number of lines, with each line representing a particular observation. Each observation line consists of a space-delimited list of the identifier for the filter in which that observation is to be done, a flag indicating whether that observation is of the target or of the sky, the integration time in milliseconds and the number of repetitions of that observation. An example programme specification is given in Listing 6.2.

The filter identifier given in the first column must have an identical definition in the list of filters specified in the global configuration file (see Listing 6.1). Any invalid lines are ignored (including lines having incorrect filter identifiers).

The integration time must be specified as an integer number of milli-seconds. Internally, this value is stored as an **unsigned int** and so may not exceed 4294967296 (an integration time of 4294967296 ms is equivalent to 49.7 days) - this an unreasonably high limit, but no smaller internal *data type* was sufficient. A 0 integration time is meaningless and will be ignored. A negative integration time will roll down from 4294967296. The number of repetitions in the final column of the observation line is also stored as an **unsigned int** and

```
[Prog1]
I 1 20000 1
R 1 20000 1
V 1 10000 1
B 1 10000 1
U 1 10000 2
B 1 10000 1
V 1 10000 1
R 1 20000 1
I 1 20000 1
U 0 10000 1
B 0 10000 1
V 0 10000 1
R 0 10000 1
I 0 10000 1
```

Listing 6.2: Example of an observing programme. The name `Prog1` was arbitrarily chosen. This programme specification represents 20 s integrations of the target in the `I` and R filters, followed by 10 s integrations of the target in the `V`, `B` and `U` filters. This sequence is then repeated in reverse order. This is followed by sky measurements in the `U`, `B`, `V`, `R` and `I` filters, each lasting 10 s.

so also allows for 4294967296 repetitions. Furthermore, 0 repetitions has a special, internal meaning - repeat until the user orders the scheduler to discontinue the observations. This is also allowed and interpreted as such within the programmes configuration file, but doing this with automatic scheduling is not recommended, as the observations will continue until they are stopped by one of the other clients - most likely because the Sun is rising or to protect the photomultiplier tube from being over-illuminated.

The programmes configuration file may contain any number of programmes with any number of lines - the upper limit is dictated only by the amount of available computer memory.

6.2.4 Macro file

The macro file is only used when the scheduler is started in automatic mode. Each line of the macro file represents an observation run of a particular target using a particular observation programme and must contain three space-separated items. The first is a valid target identifier from the stars configuration file. The second is a valid aperture identifier from the apertures line in the global configuration file. The third is a valid programme identifier from the programmes configuration file. Each item must be present and valid, otherwise that line of the macro file is ignored.

6.3 Observing Queues

The scheduler enables the user to construct an observing queue. The user can construct such a queue by selecting the desired target, selecting an appropriate aperture, either selecting

```
E708 30" Prog1
HD9357 15" ProgUBVRI
HD5446 15" ProgStroemgren3
E708 30" Prog1
```

Listing 6.3: Example macro file. `E708`, `HD9357` and `HD5446` must have been defined in the stars configuration file (see §6.2.2). `30"` and `15"` must have been defined in the apertures line of the global configuration file (see Listing 6.1). `Prog1`, `ProgUBVRI` and `ProgStroemgren3` must have been defined in the programmes configuration file (see §6.2.3).

an appropriate observing programme or setting the parameters of a single observation and then clicking on the "Enqueue" button (see Figure 6.2). If the queue is currently empty, this will also place the item at the head of the queue and start the observation.

The "View Queue" button displays a popup window containing all observations currently in the queue. The queue display is updated regularly so the user can keep track of progress. The user can also remove any observation or any number of observations from the queue in real time.

Internally, the observing queue is stored as a *singly-linked list* implemented as C **structs** , where each **struct** contains all the relevant parameters concerning the current observation as well as a link to the next observation in the queue. The scheduler then simply requires pointers to the start or head of the queue and the end or tail of the queue. This is a convenient and effective data structure in this case, as typically only the head and tail will be used in the code - if necessary, access to any item in the queue can be gained by searching for it recursively through the queue from the head. New items are added to the queue by appending them to the tail and the next observation in the queue is accessed by taking the observation at the head of the queue and replacing it with the next observation (which is referenced by the former head). The advantage of using a *singly-linked list* is that it is completely dynamic and that the amount of CPU overhead does not increase with the number of observations in the queue (which is the case with a *vector*, a suitable alternative dynamic data structure). The biggest disadvantage of using a *singly-linked list* is that it involves more CPU overhead to access an item that isn't at either the head or the tail of the queue, but, as explained above, this should rarely be necessary.

If the system is in manual mode and a situation is encountered where the telescope needs to be moved (such as when a new target reaches the front of the queue or when the telescope needs to be moved from the star to the sky or from the sky to the star), the user is prompted and the scheduler halts the queue until the user orders the scheduler to continue with the queue (presumably after the necessary adjustments have been made) - see Figure 6.4. In automatic mode the user is not prompted. The telescope is moved and target acquisition is done automatically. The user may switch between automatic and manual mode while the queue is being processed.

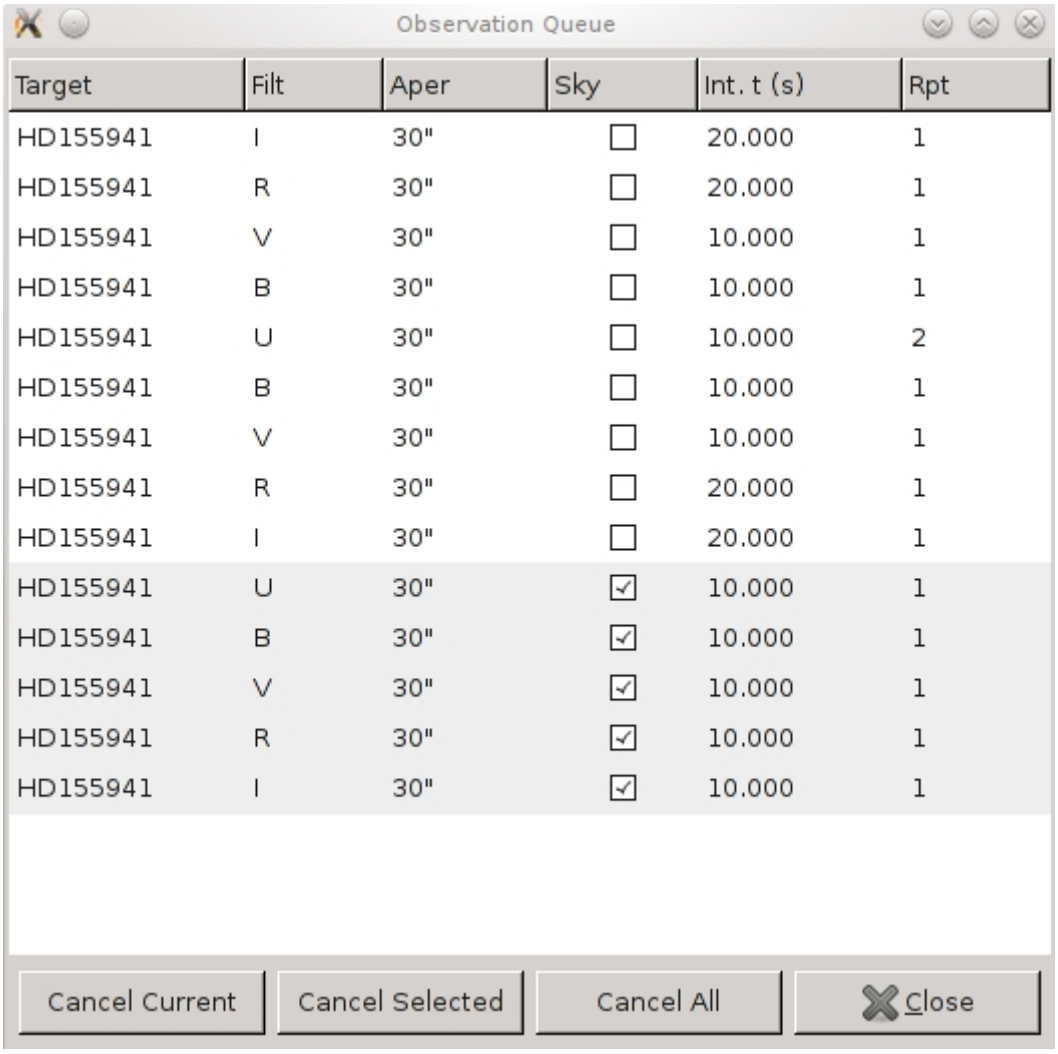

Figure 6.3: The scheduler popup window for displaying the observing queue.

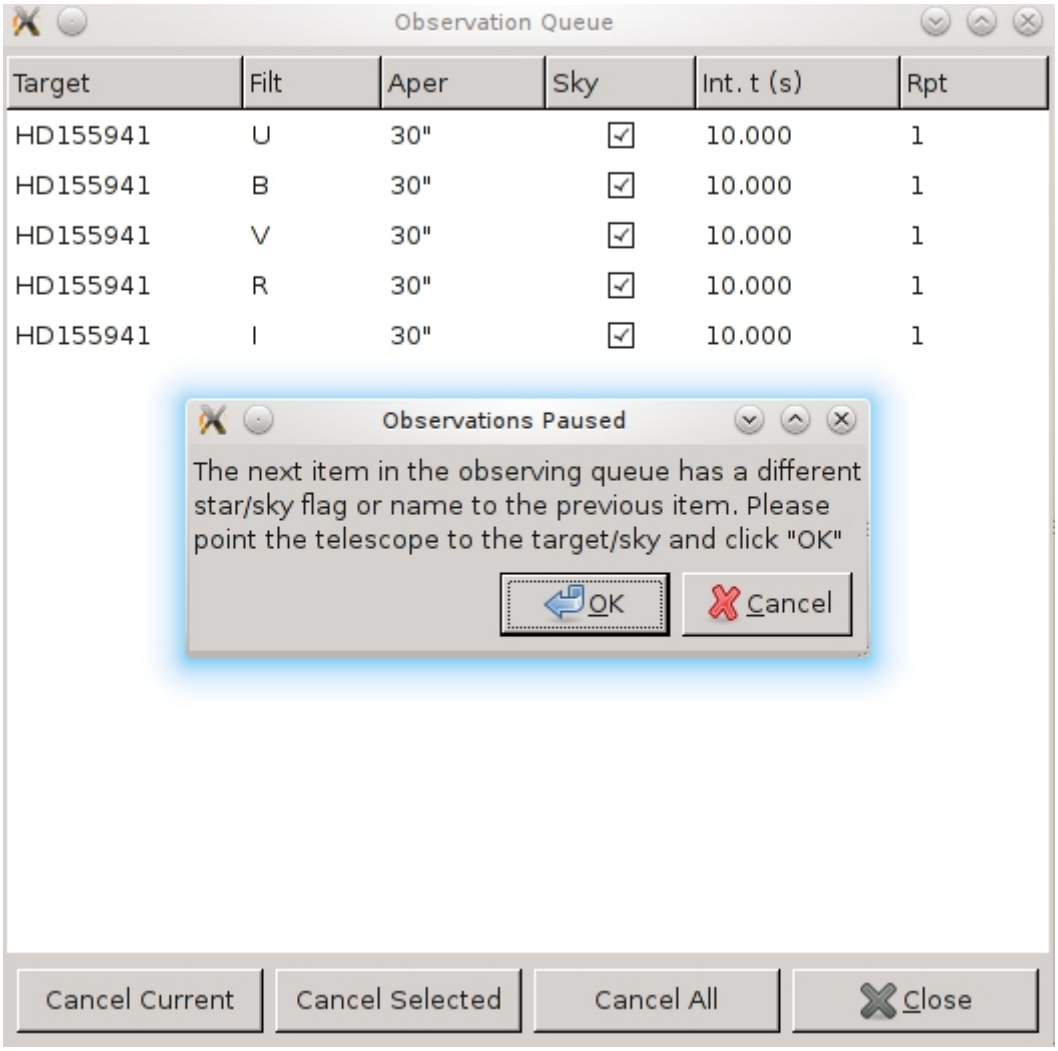

Figure 6.4: An observing queue that has been paused so the user can move the telescope. While the queue is paused, the user is free to control all aspects of the telescope and instrument through the other components of the ACT software.

6.4 Forward Look

Future versions of the scheduler could include a higher level of "intelligence" which would define the night's observing programme automatically, based on sky conditions and target priority.

Another welcome addition to the scheduler would be the ability to reorder items within the observing queue.

Finally, future versions of the scheduler should also implement a means of connecting to global transient alert networks.

Chapter 7

Photometry

The photometry programme is intended to collect and permanently store photometry. In the context of the ACT software layout, this task implies that the photometry programme needs to interface with the photometry-and-time driver to collect accurate photometry, display recently-collected photometry to the user and store all photometry permanently in such a format that it may be accessed by the established SAAO data reduction software.

Section 7.1 describes the functional requirements of the photometry programme. Section 7.2 describes how the photometry programme collects photometric data as well as aspects related to ensuring the safety of the photometric hardware. Section 7.3 describes the ways by which photometric data are displayed to the user as they are collected. Section 7.4 outlines various aspects related to how the collected data are permanently stored. Section 7.5 describes potential future developments of the photometry programme.

For a description of the software interface with the photometry and time hardware, refer to Appendix B.

7.1 Functional Requirements

In the context of astronomical software, the photometry programme needs to collect photometry, display it to the user in a convenient way and store it permanently in such a format that the data reduction software can access it.

7.1.1 Collection of Photometry

The photometry programme needs to be able to collect photometry with integration times of as low as 1 millisecond and as high as several minutes. This requirement is more an issue with the photometry-and-time driver rather than the photometry programme (see Appendix B).

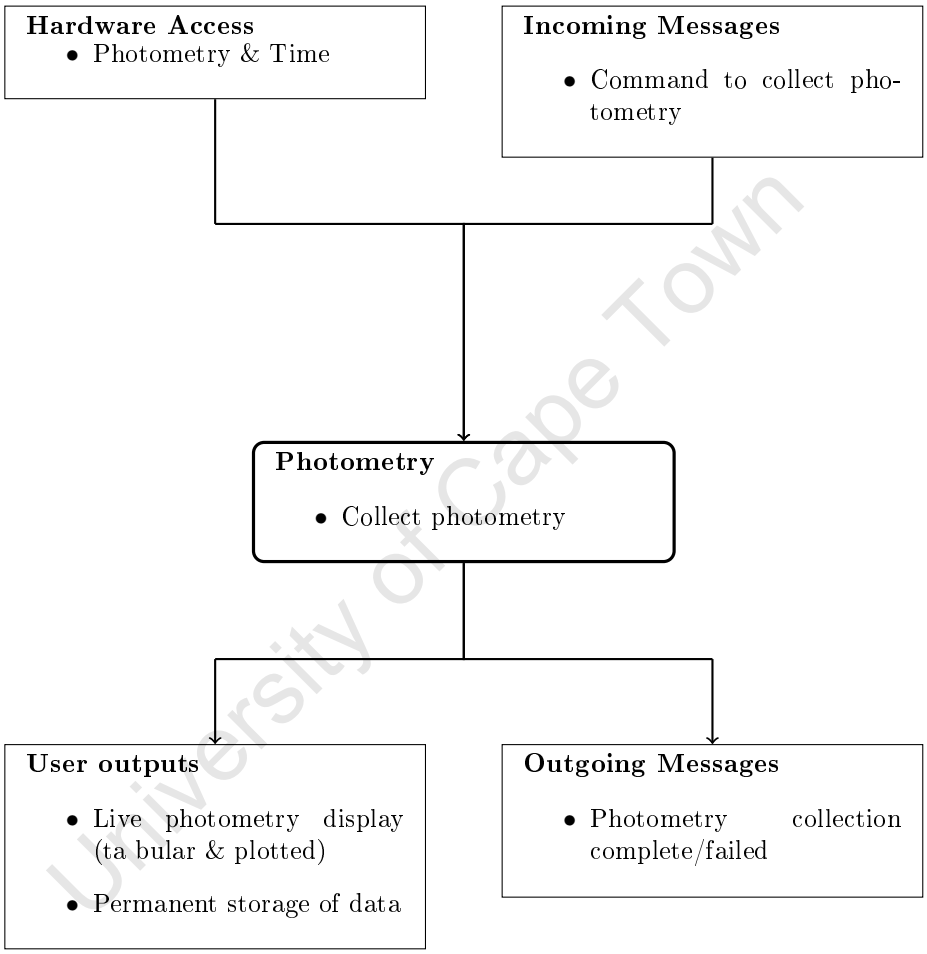

Figure 7.1: Diagram of photometry programme inputs, tasks and outputs.

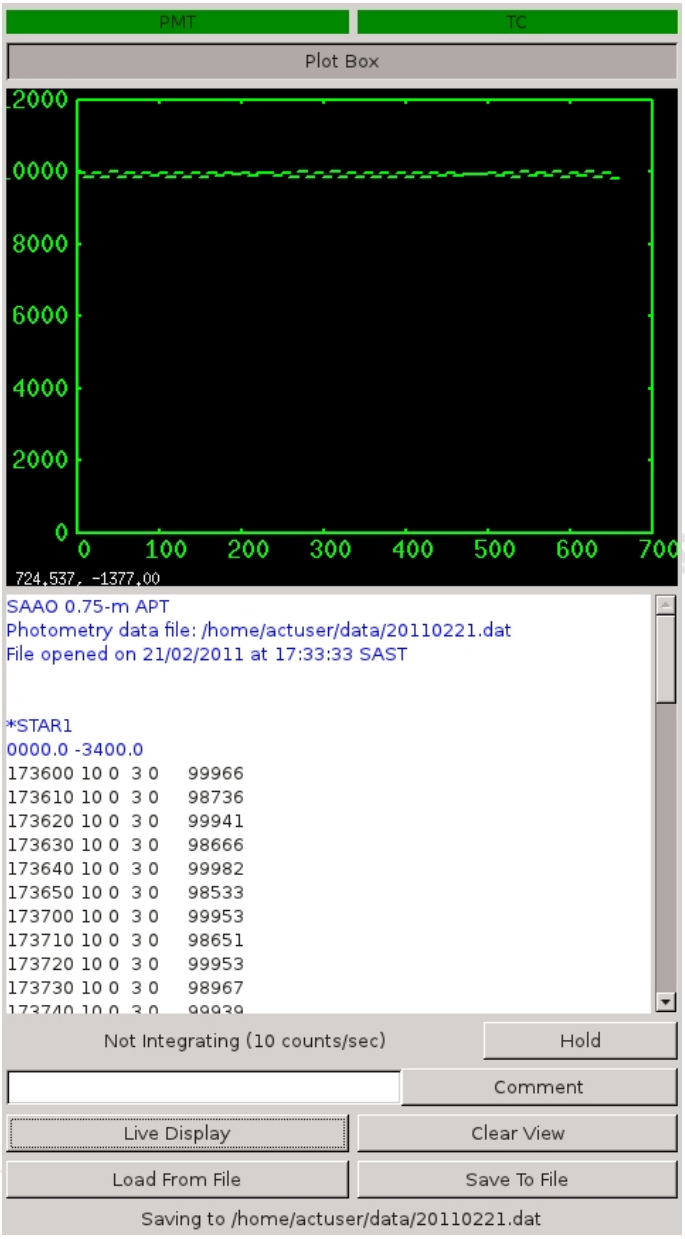

Figure 7.2: Screenshot of the photometry programme. The photometry and time hardware indicators are at the top (green means that the hardware is functioning and red means an error has occurred). The plot window can be activated using the `Plot Box` button, which appears immediately below the button if enabled - §7.3.3. The tabular data display follows after and displays all data as they are collected (in black) as well as any comments to be stored in the data file (in blue) and error messages stored in the data file (in red - not shown here) - §7.3.2. The line of text below the tabular data display shows the current photometric count rate. The start of an integration may be postponed using the `Hold` button - §7.2. Comments are added by typing the message in the text entry and pressing `Comment`. The `Live Display` and `Client View` buttons control how tabular data are displayed - §7.3.2. Buttons that control where photometric data are stored are at the bottom of the screen.

7.1.2 Photometry Display

Photometry needs to be displayed to the user as soon as it is collected. The display should consist of a numeric, tabular component displaying the counts collected during each integration and a graphical component showing a plot of all integrations collected during the course of the night and/or a particular observing run.

7.1.3 Data Storage

The photometry programme needs to ensure accurate storage of the collected photometry. A particular requirement is securing data on permanent data storage media as they are recorded - failure to do so could cause loss of data in case of a power failure.

The data also need to be stored in such a way that they can be processed by the standard data reduction programmes used at the SAAO. If this is not done, an additional programme would have to be written to reformat the data such that the data reduction programmes can access them.

7.2 Collection of Photometry

When the photometry programme receives an order to collect photometry, it sends a command to the photometry-and-time driver which causes the driver to set an internal reference point at the next turn of a second (i.e. in the future) within its internal photometry buffer. Any data collected after this point in time (i.e. any data stored after the reference point in the photometry buffer) the driver makes available to the photometry programme. The photometry programme periodically checks whether any new data are available. The photometry programme bins all new data to compose an integration of the required length of time.

Currently, the period at which the photometry programme polls the photometry-and-time driver for new data is set to 1 second*. The period can be safely increased in the photometry programme's source code, as long as it can be practically guaranteed that the driver will not overwrite data not yet read by the photometry programme, which happens once every 5 seconds (with the parameter values currently in use in the photometry-and-time driver). The period can also be reduced, although this will cause greater overhead in the photometry programme.

The photometry programme does not reset the driver's reference point for subsequent integrations, meaning that subsequent integrations are initiated the moment the current integration is complete. Therefore, if the integration time is not an integer number of seconds, any subsequent integrations will not start at the turn of a second. This is a desirable effect in the case of rapid photometry and may or may not be desirable in the case of integrations of several seconds. Integrations lasting several seconds would likely last

*The period with which the photometry programme is independent of the period with which the photometry-and-time driver reads photometry from the hardware. Every time the photometry programme polls the photometry-and-time driver, it reads all data from the driver's buffer that has not yet been read by the photometry programme. See Appendix B

an integer number of seconds anyway. This design decision was made because resetting the reference point would mean that all data collected during the remainder of the current second would be discarded. With such an alternative implementation, if the photometry programme were commanded to collected a continuous series of 1 millisecond integrations, it would in fact record a 1 millisecond integration once per second. For integrations lasting an integer number of seconds, this alternative implementation would have caused 1-second gaps between subsequent integrations.

The user may opt to delay a particular integration by means of the `Hold` button (see Figure 7.2). If the button is activated (depressed), any integration that would have started at the turn of the next second (whether it is a newly-received integration order or a repetition of an integration currently underway), would be halted. This feature is particularly handy in cases where the user wishes to start an integration at the turn of a minute or at the turn of an hour, given the unpredictable delay between the command to start an integration sent from the scheduler and the response to the command by the photometry programme.

The photometry programme can fail to collect photometry due to an over-illumination condition or if the instrument reports 0 counts.

7.2.1 Over-illumination Failure

If the counts-per-millisecond read from the photometry hardware by the photometry-and-time driver exceeds the maximum specified in the global configuration file[†], the driver will trigger the closure of the instrument shutter to protect the instrument and will set the driver status to a value indicating an over-illumination (see Appendix B). This probably means that the telescope is pointing at a target that is too bright to be safely observed with the photometer. If the photometry programme detects the over-illumination condition reported by the driver (by means of the driver status), it will assume that the instrument shutter is closed and that the integration cannot continue - in which case the photometry programme will report an error and return the observation message to the controller (with the status flag set to indicate that a different target should be selected).

7.2.2 Zero-counts Failure

If the photometry hardware reports zero counts, it usually means that the photometer's link to the computer's photometry hardware is broken or that the photometer's power source is deactivated. If an integration is underway or is about to start when such a condition is raised, the photometry programme will respond by reporting an error in the logs and returning the observation message to the controller (with the error flag activated to indicate that all observations should be stopped).

[†]The maximum allowable counts-per-millisecond on the photomultiplier tube is temporarily set to 100. A more appropriate value (given the instrument maximum of 1000 counts per millisecond) will be chosen during commissioning.

7.3 Live Data Display

The photometry programme can display data to the user in three ways: an approximate count rate and rolling integration counter, a tabular data display and a data plot.

7.3.1 Count Rate and Integration Counter

While the programme is not integrating, it displays an approximate count rate (counts per second) below the tabular data display (see Figure 7.2), which is in fact the instantaneous counts-per-millisecond (multiplied by 1000) at the time the photometry programme read the current countrate from the photometry-and-time driver.

While the programme is integrating, it displays the latest total counts of the current integration.

7.3.2 Tabular Data

The tabular data display shows the current contents of the file to which all the photometry is being recorded. The display is not regularly updated and new data added to the display unless the `Live Display` toggle button (see Figure 7.2) is active (i.e. depressed). Photometry with longer integration times (several seconds) can be conveniently displayed with this feature, however rapid photometry (with integration times of hundreds of milliseconds or less) cause the photometry programme to become less responsive and can in (in theory) lead to data loss under extreme conditions, which is why the feature is disabled by default.

The user may also clear the tabular data display using the `Clear View` button, however clearing the display will not affect the data stored on disk.

7.3.3 Data Plot

In order to see a live plot of all collected data, the user must press the `Plot Box` button (see Figure 7.2). This will shrink the tabular data display vertically and insert a box containing the plot above the tabular data display. Pressing the button again will remove the data plot box.

The plot box simply contains a *GtkSocket*, the contents of which is the output of *GNUplot*[‡], an open-source plotting utility. *GNUplot* sends its output to and receives its input from the *GtkSocket* in the same way the photometry programme does with its corresponding *GtkSocket* in the main controller (see §4.2.3). In this way, a powerful and diverse plotting utility is employed by the photometry programme in a seamless manner.

GNUplot has built-in auto-scaling, zooming, ruler and panning. More information on these functions are available in the *GNUplot* online help facility accessible from the *GNuplot* shell and on the *GNUplot* website[§].

The data for the plot are stored in a temporary file somewhere on the hard disk - the photometry programme requests a random filename (within a directory of temporary files)

[‡] `http://www.gnuplot.info`

from the operating system. The operating system may delete this file at any time after it is no longer open in any programme. The photometric data read from the photometry-and-time driver is binned (with a bin-width equal to the period with which the photometry programme reads photometry from the driver - currently 1 second) and saved in the temporary file where they can be accessed by *GNUplot*.

7.4 Data Storage

When the photometry programme starts up, it reads the full path of the directory on the hard-disk which contains all the photometry from the global configuration file (see §5.4) and generates an initial filename based on the current date (e.g. "20110221.dat" for the 21st of February, 2011). If the file already exists, such as when a power failure occurs and the system starts up a second time on a particular night after the power returns, the contents of that file will be preserved and any new data will be appended to the file.

The user may change the file to which new data will be saved by pressing either the `Save to File` or the `Load from File` buttons (see Figure 7.2). If the user saves to a new file, the file will be created, the current contents of the tabular data display will be written to the file and any new data will be appended to the file. If the user saves to an existing file, the contents of the file will be erased, the current contents of the tabular data display will be written to the file and any subsequent data will be appended to it. If the user loads an existing file, the contents of the tabular data display will be replaced with the contents of the file and any new data will be appended to the file. The user cannot load a file that does not exist. In this manner, it is ensured that all data collected is saved to some file on the hard disk.

Usually, when working with files on the hard-disk, the operating system buffers all write-operations to the file in the computer's primary memory. This can cause loss of data, as there is no guarantee when the operating system will update the file on disk. For this reason, the photometry programme always *flushes* after writing to file, thus forcing the operating system to write the contents of the buffer to disk immediately.

7.5 Forward Look

Future versions of the photometry programme should implement a database interface for permanent data storage as opposed to a text-file based implementation. A database system would be more robust, as the photometry programme could simply send a data packet to the database server and (as long as the connection to the server is active and no error is reported) assume that the data is securely stored. With most database management systems, databases can be duplicated and exported fairly easily and multiple, duplicate databases can be synchronised regularly and effortlessly. The use of a database system would also improve data security, as database systems can store all data in a compressed and encrypted format and the owner of a database can specify which users can access which parts of which tables

of data. With the advent of the *Structured Query Language* (*SQL* - which most database management systems use to allow access to the databases), working with databases has become significantly easier than working with text files.

Chapter 8

DTI programme

The Dome, Telescope and Instrument programme (*DTI*) is intended to interface with the Programmable Logic Converter (PLC) and the telescope motor controller connected to the computer. Refer to Appendix D for details concerning the tasks of the PLC and how the software interfaces with it and to Appendix C for details concerning the telescope motor controller and the driver developed to simplify controlling the motors.

In many respects, the DTI is the most important of all programmes in the software suite, since it manages all aspects of the dome, telescope and instrument and, in so doing, fulfils the functional requirements of the telescope control software described in §3.1.2 and §3.1.3. The DTI not only needs to control the dome, telescope and instrument, but also protect them - be it protection from damage as a result of mechanical failures, protection from the environment or protection from the user. The DTI therefore needs to be "aware" of the limitations of the hardware and be able to recognise a potentially hazardous situation. The DTI also needs to shield the user from the quirks of the dome, telescope and instrument.

8.1 Level of Control

One of the biggest issues experienced during the development of this programme is the level of control that should be allowed within the user interface. An example would be filter and aperture control. The user would conceivably only need to set these parameters before an integration starts - they are therefore parameters of an observation and should be set within the scheduler. However, the user interface for the DTI should still indicate the status of the filter and aperture wheels (because it would need to respond appropriately to errors anyway). The question is then whether the user should be able to set the filter and aperture or whether merely the feedback from these devices should be presented to the user. It was decided, as a general rule, to allow the user to control everything possible from the DTI and to implement some means of preventing the user from doing things that should not be allowed or does not make sense (such as changing the filter during an integration) or some means of warning the user before such an action is performed.

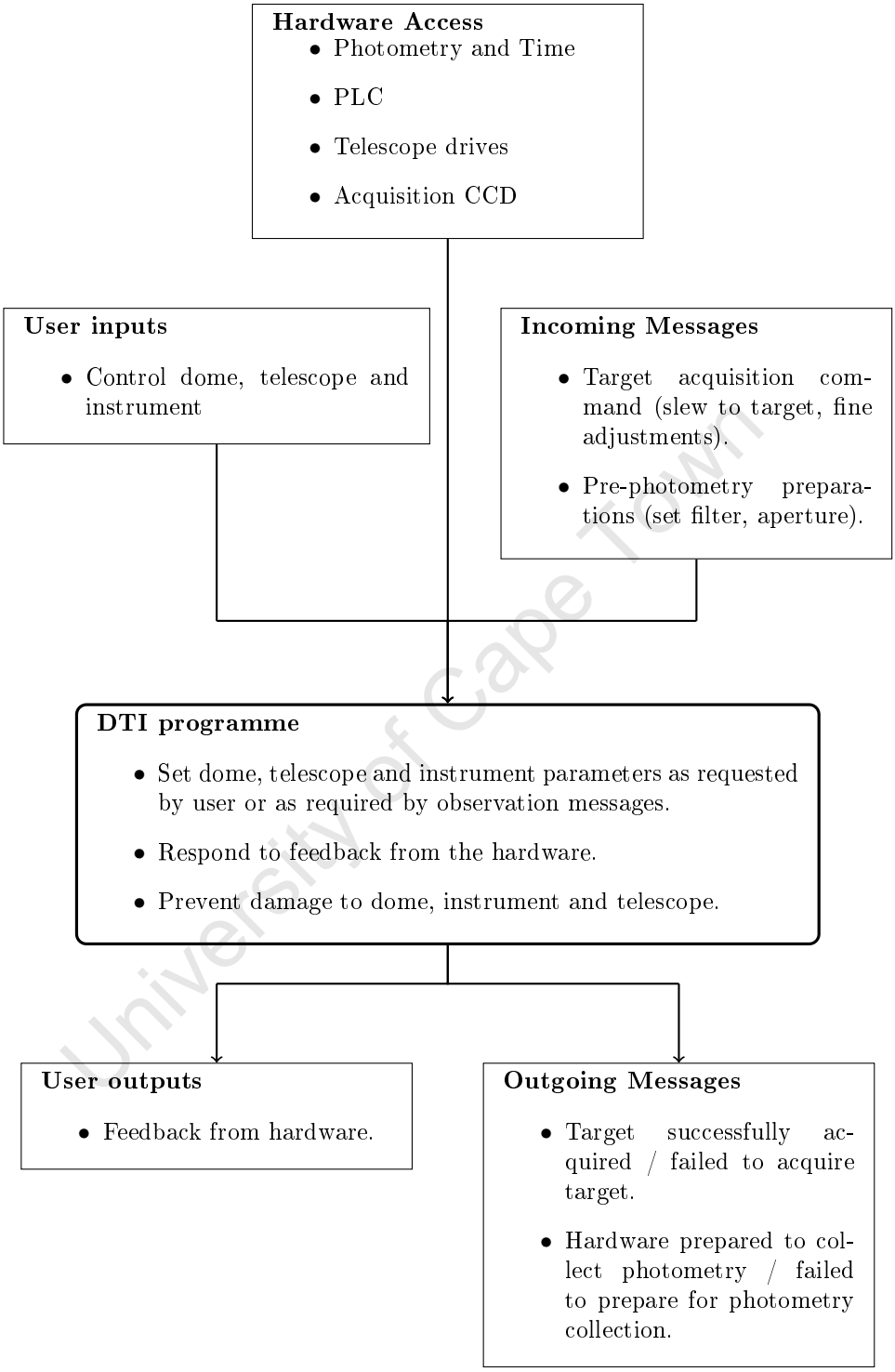

Figure 8.1: Diagram of DTI inputs, tasks and outputs.

8.1.1 Automatic and Manual Control

When the system is set to manual control, the DTI assumes that the user will set all parameters as necessary - with the exception of slewing to the target coordinates. When the DTI receives an observation message requesting the target be set, it will slew to the designated coordinates and return the message to the controller. The DTI assumes that the user will activate tracking (unless it is already active) and dome guiding and will set the acquisition mirror position as necessary. The DTI ignores observation messages requesting all parameters be set as necessary for photometry to be collected if the system is set to manual control.

The DTI sets all necessary parameters when the system is set to automatic control. When the DTI receives an observation message requesting the target be set and the system is set to automatic control, it will ensure that the dome is open, the dropout is open (if necessary), dome guiding is active, the acquisition mirror is set to the acquire position, the instrument shutter is closed and that the telescope is pointing to the specified coordinates. When the DTI receives an observation message requesting the preparations for photometry be done and the system is set to automatic control, it will set the requested filter and aperture, set the acquisition mirror to the measure position and open the instrument shutter - the programme will also ensure that the telescope is pointing at the specified coordinates, that dome guiding is activated and that the dome shutter is open although these should already have been established.

Table 8.1 summarises the actions taken by the DTI in case an observation message is received - both under manual and automatic control.

8.2 DTI Protection

8.2.1 Telescope Limits

The telescope is equipped with both electronic and mechanical limits (see §2.2.1). In addition to these limits, the software is also equipped with a set of limits which is more restrictive than both the mechanical and electronic limits. The software limits define the limits (in hour angle and declination) of the recommended operational range of the telescope. A further complication is introduced by the fact that the user must be able to operate outside the recommended operational area of the telescope, but the DTI should refuse a command to move beyond that area if the command was not issued by a human observer. How the DTI distinguishes between these two scenarios and handles attempts to move beyond the software limits in each scenario is discussed in §8.3.

Hard-coded Software Limits

Although the mechanical, electronic and software limits provide quite comprehensive protection for the telescope, they do not necessarily prevent the telescope from being pointed at a target below the horizon. For this reason, an additional measure was introduced which

Control	Obsn stage	Item	Action
Manual	Target Acquisition	Dome shutter	No change
Auto	Target Acquisition	Dome shutter	Open
Manual	Photometer Setup	Dome shutter	No change
Auto	Photometer Setup	Dome shutter	Open*
Manual	Target Acquisition	Dome dropout	No change
Auto	Target Acquisition	Dome dropout	Open if necessary, close if not
Manual	Photometer Setup	Dome dropout	No change
Auto	Photometer Setup	Dome dropout	Open if necessary, close if not*
Manual	Target Acquisition	Dome guiding	No change
Auto	Target Acquisition	Dome guiding	Activate
Manual	Photometer Setup	Dome guiding	No change
Auto	Photometer Setup	Dome guiding	Activate *
Manual	Target Acquisition	Move telescope	Slew to target
Auto	Target Acquisition	Move telescope	Slew to target
Manual	Photometer Setup	Move telescope	Slew to target *
Auto	Photometer Setup	Move telescope	Slew to target *
Manual	Target Acquisition	Acquisition mirror	No change
Auto	Target Acquisition	Acquisition mirror	Move to "Acquire"
Manual	Photometer Setup	Acquisition mirror	No change
Auto	Photometer Setup	Acquisition mirror	Move to "Measure"
Manual	Target Acquisition	Filter	No change
Auto	Target Acquisition	Filter	No change
Manual	Photometer Setup	Filter	No change
Auto	Photometer Setup	Filter	Set to requested
Manual	Target Acquisition	Aperture	No change
Auto	Target Acquisition	Aperture	No change
Manual	Photometer Setup	Aperture	No change
Auto	Photometer Setup	Aperture	Set to requested
Manual	Target Acquisition	Instrument shutter	No change
Auto	Target Acquisition	Instrument shutter	Close
Manual	Photometer Setup	Instrument shutter	No change
Auto	Photometer Setup	Instrument shutter	Open

Table 8.1: Summary of actions taken by DTI when observation message is received. The items marked with * indicates parameters that should already have been set - if not, they will be set.

calculates the zenith distance of the telescope and stops the telescope immediately should it move beyond a certain hard-coded maximum zenith distance.

Furthermore, none of the telescope limits discussed so far takes the dome into account. With stationary obstacles, preventing a collision with the telescope is as simple as defining the software limits such that the telescope cannot collide with the obstacle. However, because the dome is a moving structure, preventing a collision between the telescope and the dome requires special attention. Yet another protection measure was introduced which considers the azimuth and zenith distance of the telescope as well as the azimuth of the dome and determines whether a collision is possible according to certain hard-coded parameters. The DTI stops the telescope and dome immediately if a collision is imminent.

8.2.2 Photomultiplier Tube

The DTI must be able to close the instrument shutter within a few milliseconds after an over-illumination warning is issued by the photometry-and-time driver. For this reason, a separate programme loop was created which runs more frequently (ca. 100 s^{-1}) than the regular programme loop and checks for such a warning. It was decided not to read the photometry-and-time driver status in order to detect an over-illumination (see §B.3.2), but rather read the count rate from the driver and allow the user to specify the maximum allowable count rate in a configuration file. This allows the user to observe targets which are only slightly brighter than the recommended maximum without the instrument shutter closing immediately after the observation starts.

Although additional measures to protect the PMT (such as those described in §3.1.2) are not implemented at the moment, they will be implemented before the telescope becomes fully operational.

8.2.3 Environmental Conditions

The DTI receives regular messages from the environment programme that relay the current weather conditions. The environment programme determines whether it is safe for the telescope to be operational (see §10.3) through the `Active/Idle` field of the environment message. If the `Active/Idle` field is false, it means that it is not safe for the telescope to be operational, in which case the DTI closes the dome and parks the dome and telescope.

If an observation is in progress when an environment message is received that indicates unsafe observing conditions, the DTI issues an observation command with its error flag set. The observation message will then follow the normal paths (see Figure 4.13) so that all observational tasks are stopped, especially the collection of photometry (to ensure data integrity).

Since the environment programme forms part of the observation cycle, it will prevent any observation from starting if the conditions are not safe for observing. The DTI programme therefore only needs to monitor the environment during observations. Nevertheless, the DTI will not perform its normal operations (see Table 8.1) if it is unsafe to do so.

8.3 Moving the Telescope

The telescope can be moved either by issuing a "go to" command, which requests that the DTI move the telescope to the specified coordinates, or by the user pressing one of the cardinal direction buttons, in which case the telescope is moved in the specified direction and at the specified speed until the button is released. The former command is referred to as an automatic move command, since the command may be issued by the user or an automation routine within the software and entails the DTI starting to move the telescope slowly, speeding it up gradually, slowing it down again when the telescope nears the specified coordinates and then stopping it at those coordinates. The latter command can (by design) only be issued by a human observer and is therefore referred to as a manual command.

As mentioned in §8.2.1, the DTI should grant various levels of freedom (in terms of telescope motion) depending on whether the command to move is automatic or manual.

In the case of a manual command to move, it can be assumed that the observer knows what he/she is doing and the software hour angle and declination limits do not need to be strictly adhered to. However, if the user chooses to move the telescope manually at maximum speed and moves beyond the software hour angle and declination limits, the DTI automatically slows the telescope down to a safe speed*.

In the case of an automatic move command, the telescope cannot be moved beyond the software hour angle and declination limits, whether the command was issued by the user or an automation routine. If the telescope is under manual control, the user could still use the "go to" function to move the telescope near one of the software limits and then go beyond the limits manually. Furthermore, in terms of hour angle, sidereal targets are moving targets. The DTI should therefore be able to differentiate between an automatic command where the hour angle is specified and an automatic command where the right ascension is specified. It was decided that an observation command message would cause an automatic move command specifying the target right ascension and the "Go To" button in the DTI's *graphical user interface* (*GUI*) may be used by the user to issue an automatic move command specifying the target hour angle.

In order to treat both automatic move commands and manual move commands correctly within the DTI (especially in terms of the regular checks done to ensure telescope safety and whether the target has been reached) - along with the various parameters of the motion (such as rate of motion, stopping conditions and the level of freedom of motion) - a C **struct** was defined that would contain all information pertaining to a command to move the telescope (see Figure 8.2).

Besides the automatic and manual move modes, there is the telescope nudge mode and several modes related to the calibration of the pointing encoders. The telescope nudge mode was implemented so that the telescope can be moved through a small number of motor steps instead of moving the telescope until it reaches the specified coordinates. The

*The danger in this case is not so much that the telescope might go beyond the electronic limits (which seems impossible), but rather that the telescope is stopped dead immediately when one of the electronic limits is activated and any sudden changes in rate of motion are strongly discouraged for mechanical reasons.

Telescope Motion Structure	
Move Mode	integer (see Table 8.2)
Direction of Motion	integer, binary (see Table C.3)
Current Rate of Motion	integer, arb. unit
Desired Rate of Motion	integer, arb. unit
Correct for sidereal motion	integer, true/false
Current hour angle	float, decimal hours
Current declination	float, decimal degrees
Target hour angle	float, decimal hours
Target right ascension	float, decimal hours
Target declination	float, decimal degrees
Northern software limit	float, decimal degrees
Southern software limit	float, decimal degrees
Eastern software limit	float, decimal hours
Western software limit	float, decimal hours

Figure 8.2: Data structure containing parameters of telescope motion.

Nr.	Name	Description
1	`MOTOR_MODE_NUDGE`	Nudge Telescope
2	`MOTOR_MODE_MANUAL`	Manual
3	`MOTOR_MODE_AUTO`	Automatic
4	`MOTOR_MODE_PREINIT_PARK`	Initialisation - park telescope
5	`MOTOR_MODE_PREINIT_DOMEPARK`	Initialisation - park dome
6	`MOTOR_MODE_CALIB_EW`	Initialisation - hour angle calibration
7	`MOTOR_MODE_MIDINIT_PARK`	Initialisation - park telescope
8	`MOTOR_MODE_CALIB_NS`	Initialisation - declination calibration

Table 8.2: Summary of modes of telescope motion. The integer given in the first column is stored in the `Move Mode` field of the `Telescope Motion Structure` (see Figure 8.2)

telescope initialisation modes were implemented because calibrating the encoders requires that a particular procedure be followed (see §8.4). Each of the initialisation modes represents a particular step of the calibration procedure. Once a particular initialisation step is complete, the next step is done by incrementing the mode flag and setting the motion parameters as necessary.

The various modes that have been implemented are summarised in Table 8.2. The regular operations of the DTI includes calling a function which takes a `Telescope Motion Structure` as a parameter and performs all regular operations related to telescope motion according to the specified mode and its accompanying parameters. This implementation proved to be very easy to expand in order to accommodate a wider range of modes, most notably because each mode is treated individually and there is therefore no chance of breaking code related to a particular mode by changing the code related to a different mode.

8.4 Initialisation and Encoder Calibration

Several steps are taken when the DTI starts up in order to ensure that all hardware components of the photometer are functioning normally. The following checks are performed:

- Cycle through all filters.
- Cycle through all apertures.
- Move the acquisition mirror to the `measure` and `acquire` positions.
- Open and close the instrument shutter.

Once these checks have been successfully completed, the instrument shutter is closed and the acquisition mirror is moved to the measure position.

After the instrument functionality tests, the DTI calibrates the encoders, which is done by moving the telescope to the 4 electronic telescope limits. The calibration must be done every time the DTI starts up, since the telescope uses relative encoders and at least the zero-points must be determined at least once per night. The hour angle encoder is zeroed at the Western electronic limit and the declination encoder is zeroed at the Northern electronic limit. The telescope also needs to be moved to the Eastern and Southern limits in order to calculate the parameters of the linear functions that translate the encoder positions to the raw hour angle and declination. The steps to follow while doing the calibration are as follows:

1. Park the telescope at the zenith.
2. Park the dome at 0 degrees in azimuth.
3. Move the telescope Westward until the electronic limit is reached.
4. Move the telescope Eastward until the electronic limit is reached.
5. Park the telescope at the zenith.
6. Move the telescope Northward until the electronic limit is reached.
7. Move the telescope Southward until the electronic limit is reached.
8. Park the telescope at the zenith.

Step 2 of the above procedure is necessary to ensure that the dome is in a safe position for the telescope to be moved to all hour angles and declinations accessible by the telescope - without this step there is a chance that a situation will be created where a collision is imminent, in which case the DTI will immediately stop the telescope and cancel the initialisation procedure. This problem is easily solved by the presence of an observer, but in automatic mode the result will be that the telescope will not observe anything until the dome no longer obstructs the telescope when it is near the electronic limits. Step 1 of the

above procedure is performed to ensure that step 2 can be performed safely, since a collision can also be caused by the telescope being near the Northern electronic limit (which is where the telescope is usually parked when the DTI shuts down) and the dome being moved. Also in this case the hardware protection routines would prevent a collision by stopping the dome, cancelling the initialisation and effectively preventing any automated observations.

Once the telescope pointing encoders have been calibrated, the dome may be opened and the first target acquired.

8.5 Testing

The DTI underwent extensive testing in the laboratory. Although the PLC itself was in the laboratory during these tests, it was not connected to any of the telescope hardware. Instead, two panels containing lights and switches was used to indicate when commands are sent to the hardware and to simulate responses from the hardware, respectively (see Figures 8.3 and 8.4).

For example, in order to test whether the DTI was controlling the instrument shutter correctly, the command would be issued using the DTI user interface. If the command was sent correctly, the "SHUTTER" light (see Figure 8.4) would activate. The operator could then flick the "Shutter Open" switch to simulate the instrument shutter opening - this would produce feedback to the PLC, which would be relayed to the DTI. The DTI should indicate on the user interface that the instrument shutter has been opened successfully.

All aspects of the dome, telescope and instrument that are controlled by the DTI, was tested in this manner. When the PLC was installed at the telescope, it was a trivial matter to verify that the DTI was interfacing with the PLC correctly and, by implication, managing the hardware the PLC controls correctly.

The component of the DTI controlling the telescope drives was also tested in the laboratory. Two spare motors and a set of 4 switches (with which to simulate the electronic limit switches being triggered) were used during these tests. The first series of tests were fairly simple and merely served to demonstrate that the DTI is controlling the motors correctly - i.e. checking whether the correct motor is turned in the correct direction and at the correct rate, depending on the command that was issued. Further tests demonstrated that the software limits are correctly adhered to under various conditions (especially different modes of motion - see §8.3). Lastly, several tests were performed to verify that the telescope pointing encoders were being read out correctly and that the encoder readouts were being translated to raw hour angle and declination correctly. These last tests involved initiating an encoder calibration run (from the DTI), turning the encoders while the motors are turning and then triggering the appropriate electronic limit switch. The feedback from the encoders and the limit switches were simulated in this manner in order to verify that the encoder calibration procedure works correctly.

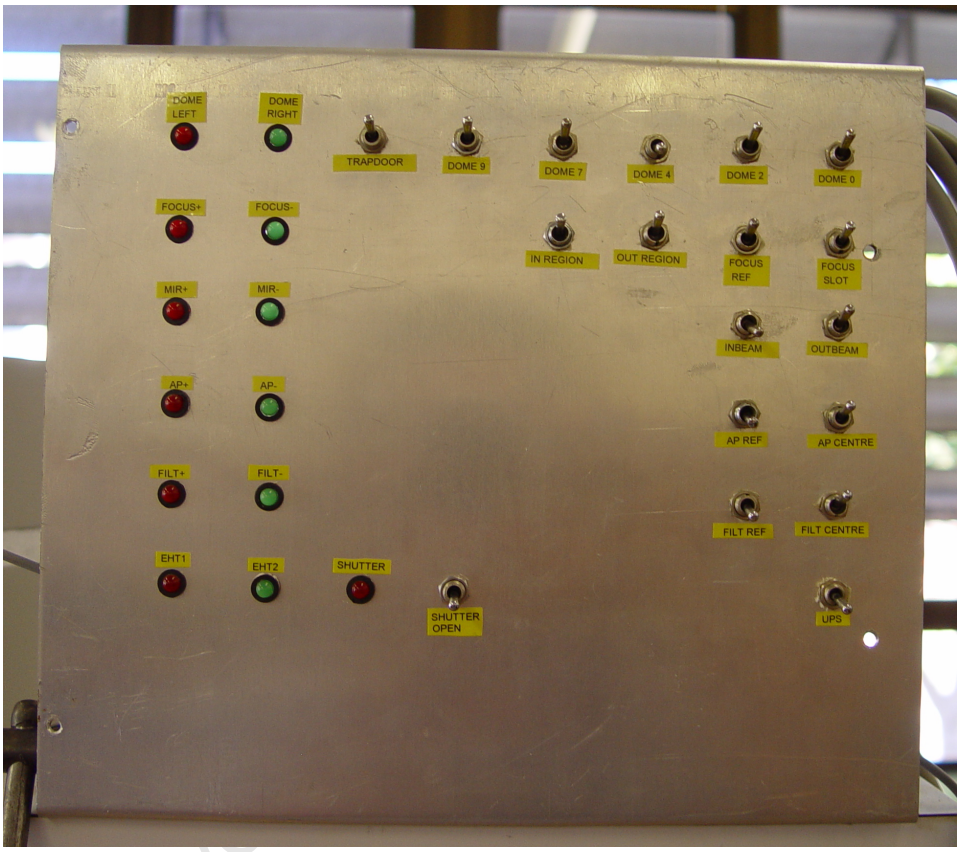

Figure 8.3: Panel of lights and switches used to simulate interactions between the PLC and the hardware it controls. The lights activate when commands are sent to the PLC by the DTI and the switches are used to simulate feedback from the hardware. The top row of lights and switches relate to dome rotation - except for the leftmost switch which simulates feedback from the trapdoor between the dome and the control room. The second row is related to telescope focus. The third row relates to the acquisition mirror. The fourth row relates to the filter wheel. The fifth row relates to the aperture wheel. The sixth row relates to the instrument shutter and the high-voltage PMT power supply - except for the far-right switch, which is used to simulate a main power failure.

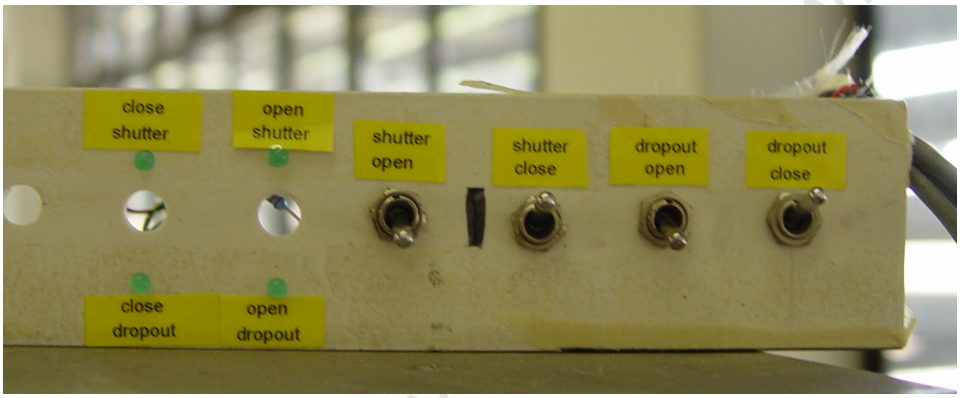

Figure 8.4: Panel of lights and switches used to simulate interactions between the PLC and the hardware it controls. The lights to the left activate when commands to open and close the dome shutter and dome dropout are sent to the PLC by the DTI programme and the switches to the right are used to simulate feedback from the dome shutter and dropout.

Chapter 9

Target Acquisition

The acquisition programme is charged with all tasks related to the acquisition CCD. This includes capturing images from the acquisition CCD (via the CCD driver, see Appendix A), doing elementary image post-processing for the user's convenience and matching the pattern of stars in a given field to the corresponding recorded template.

Section 9.1 describes the functional requirements of the software, followed by §9.2, which describes several aspects of how the acquisition image is displayed on the screen and display parameters implemented for user convenience (including lookup tables, image brightness, image contrast and grid overlays). The pattern matching procedure, one of the core facilities of the acquisition programme, is described in §9.3. The chapter ends with §9.4, which describes potential future improvements of the acquisition programme.

9.1 Programme Requirements

The programme needs to be able issue commands to the CCD driver to initiate an exposure of the acquisition CCD, read the resulting pixel data from the CCD driver and display the image on the screen. The programme must enable the user to adjust the parameters related to how the image is displayed on the screen - such as adjusting image brightness and contrast, selecting a convenient colour table, overlaying a grid over the image and allowing the user to mark points on the image (such as stars). The programme must also be able to match the pattern of stars in the field against a prerecorded pattern template - this function is of particular interest, as it is used to verify that the telescope is in fact pointing at the correct target and, in so doing, contributes toward efforts to ensure the integrity of the collected data during automatic operation of the telescope (see §3.1.1).

9.2 Image Display and Processing

Transferring the acquisition image from the CCD driver to the acquisition programme and from the programme to the graphics display hardware of the computer are the most resource-

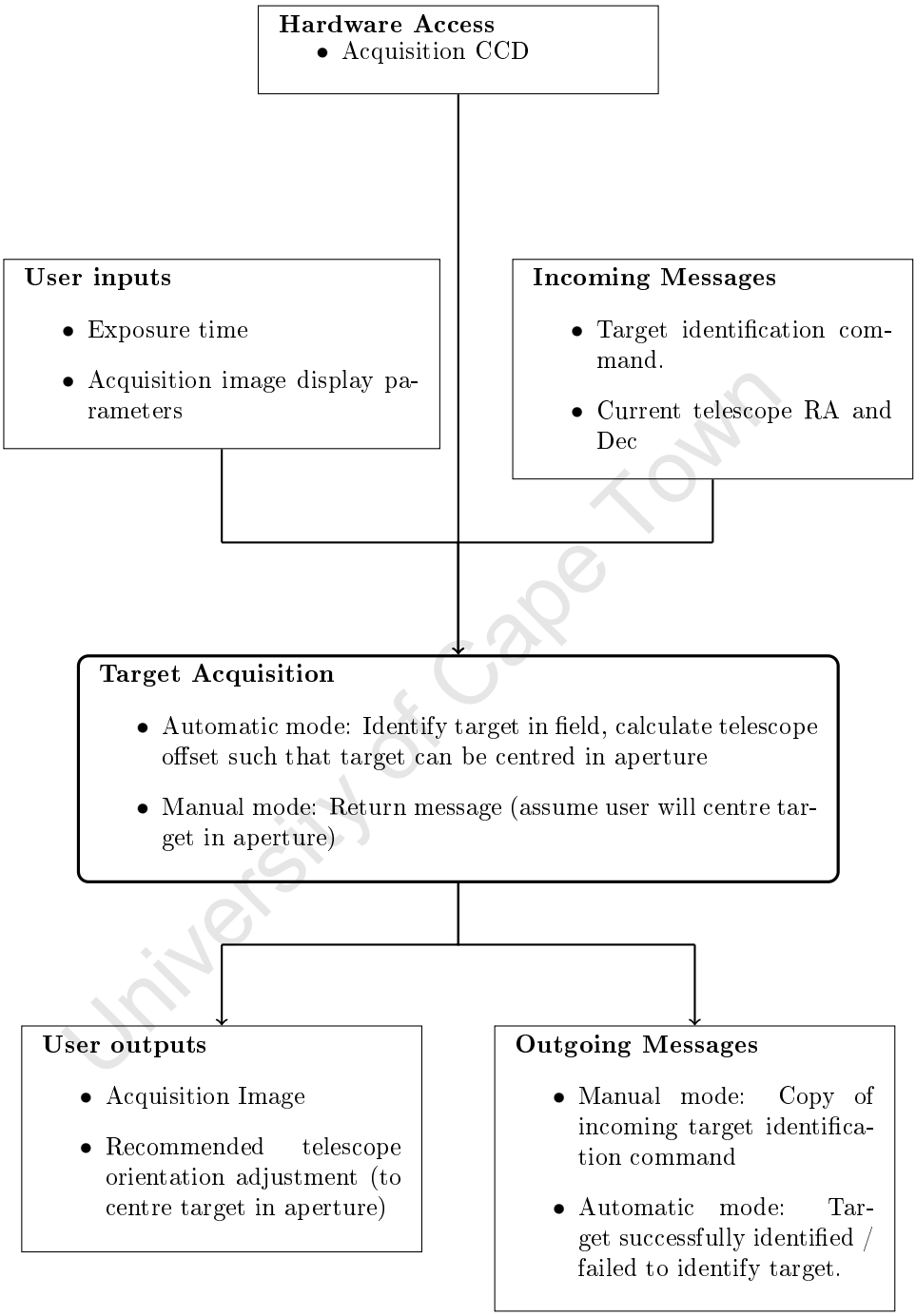

Figure 9.1: Diagram of the acquisition programme inputs, tasks and outputs.

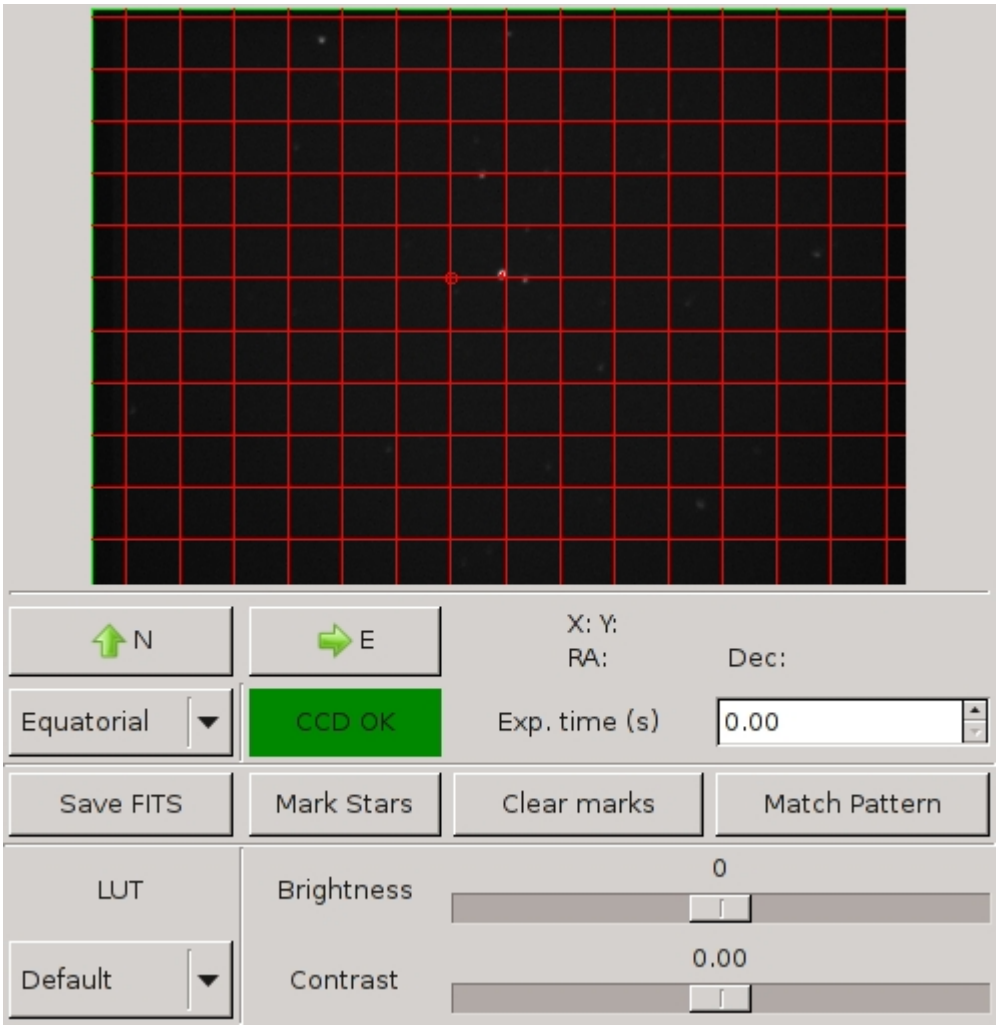

Figure 9.2: Screenshot of the acquisition programme. The top section of the screen shows the newest acquisition image, followed by a section containing controls and displays relating to the acquisition image and how it is displayed, then by a section containing file interaction controls and controls relating to the pattern matching facility and finally a section containing controls to change minor parameters of how the image is displayed.

intensive operations of the acquisition programme. The method and parameters by which the image is transferred from the CCD driver to the acquisition programme is set by the CCD driver and there is no way to reduce the resources consumed during this operation without also modifying the CCD driver. However, once the image is in the domain of the acquisition programme, the image can be processed and transferred to the graphics display hardware by many different means.

The possibility of using *OpenGL** was investigated in an attempt to reduce the resources consumed when the image is transferred to the graphics display hardware. This method reduces the load on the CPU by (effectively) shifting the burden to the graphics processing unit (GPU).

OpenGL is used to draw a textured plane in front of the virtual "camera"†, where the texture is the acquisition image. Anything that needs to be displayed over the image can then simply be drawn closer to the "camera".

Several advantages of using *OpenGL* for graphics display were discovered soon after this interface was implemented in the acquisition programme. Most notably, *OpenGL* enables the programme to do extensive and complex geometric calculations quickly and simply, also by shifting the computational burden to the GPU (which is designed to perform such calculations). This can be applied to transform back and forth between the planar coordinates of pixels on the screen and the coordinates within the virtual three-dimensional drawing volume of items displayed on the screen. In practice, this is used to determine the right ascension and declination of a particular pixel on the screen and to overlay a drawing element at a particular right ascension and declination over the acquisition image.

Although other open-standard protocols exist to transfer images to and from the graphics display hardware, *OpenGL* was chosen because it is very efficient in terms of graphics operations, is widely supported and will likely be well-supported by graphics hardware manufacturers for many years to come and allows for a high level of image processing to be done using the graphics display hardware.

9.2.1 Colour Lookup Tables, Brightness and Contrast

A colour lookup table (LUT) specifies a particular colour for each value a pixel can have. The CCD in use on the ACT has 8-bit pixels (i.e. each pixel has a value that is an integer between 0 and 255). Each LUT must therefore have 256 entries, where each entry specifies the colour (as a red-green-blue or *RGB* triplet) with which that pixel value should be displayed on the screen. All LUTs available on the system are defined in the LUT configuration file. By default, this file contains the `White`, `Red`, `Green` and `Blue` LUTs, where the `White` LUT defines 256 shades of gray, `Red` defines 256 shades of red and so forth. As the pixel values 0 and 255 have special meaning (under-exposed and over-exposed, respectively), these values

*OpenGL is an abbreviation of "Open Graphics Library", an open-standard graphics programming interface, initially developed by Silicon Graphics Inc. in 1992, and is still in use today. [18]

†The notion of a "camera" does not apply to *OpenGL*, but it is a useful and mostly accurate simplification in this case.

are usually given special definitions in the LUTs. For example, the `White` LUT defines the display colour of 0 as bright red and that of 255 as bright green.

The (user-selected) brightness and contrast modify the mapping between pixel values and defined colours in the LUT. Given a LUT defining 256 colours and a pixel value v (between 0 and 255) the element i within the LUT (which defines a particular RGB triplet) is calculated from the brightness B and contrast C as follows:

$$i = v * 255^C + B$$

If i has a negative value, it is set to 0. If i has a value greater than 255 it is set to 255. The equation and the parameters of the equation were chosen to maximise user convenience.

For each pixel in the acquisition image, the LUT element i is calculated and the i^{th} RGB triplet in the LUT is substituted into the image.

9.2.2 Grid Overlay

It was deemed necessary to implement three grid types: `Viewport`, `Pixel` and `Equatorial`. In each case, the grid is displayed in the under-exposed colour for roughly 1 second after a new image is received from the CCD driver and in the over-exposed colour otherwise.

Viewport

In the context of the acquisition programme and how it uses *OpenGL*, the viewport can be considered a two-dimensional plane onto which all visible elements within the three-dimensional volume is projected. The acquisition programme defines the viewport such that the coordinate $(0, 0)$ on the viewport corresponds to the bottom-right (i.e. South-Eastern) corner of the acquisition image and the coordinate $(1, 1)$ on the viewport corresponds to the top-left (i.e. North-Western) corner of the acquisition image. With the `Viewport` grid type, lines are drawn at intervals of 0.1 in the reference frame of the defined viewport. Because the acquisition image is rectangular (not square), the `Viewport` grid type produces a rectangular grid. See Fig. 9.3.

Pixel

The `Pixel` grid type, as the name implies, draws a grid in the reference frame of the acquisition image. Grid lines are drawn with a regular spacing of 10 pixels. Although the grid may appear square, it is usually slightly rectangular (this effect depends on the ratio of the width to the height of the pixels on the screen on which the programme is being displayed). The origin of this reference frame is the top-left (i.e. North-Western) corner of the acquisition image. See Fig. 9.4.

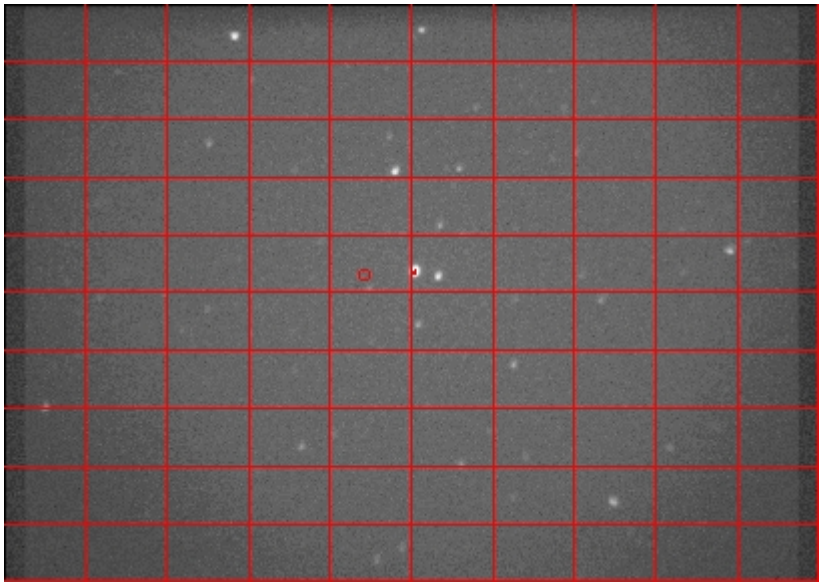

Figure 9.3: `Viewport` grid type overlay over acquisition image.

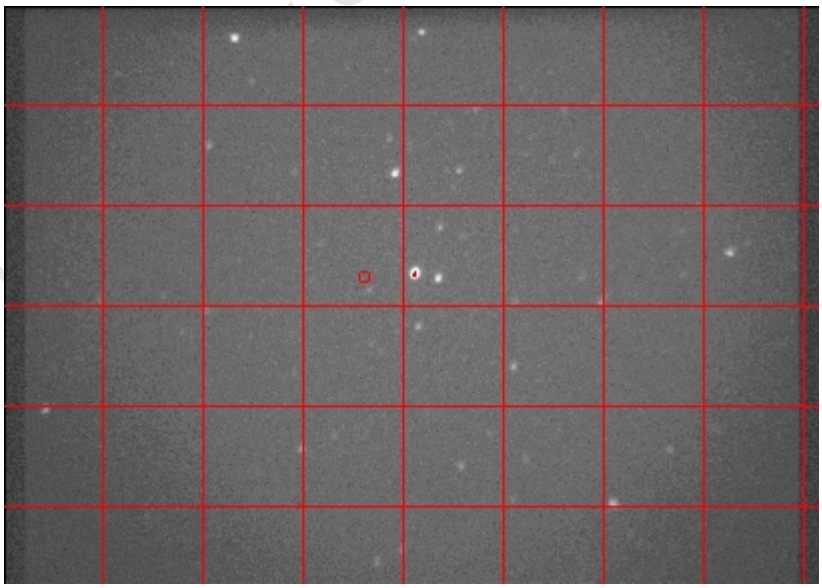

Figure 9.4: `Pixel` grid type overlay over acquisition image.

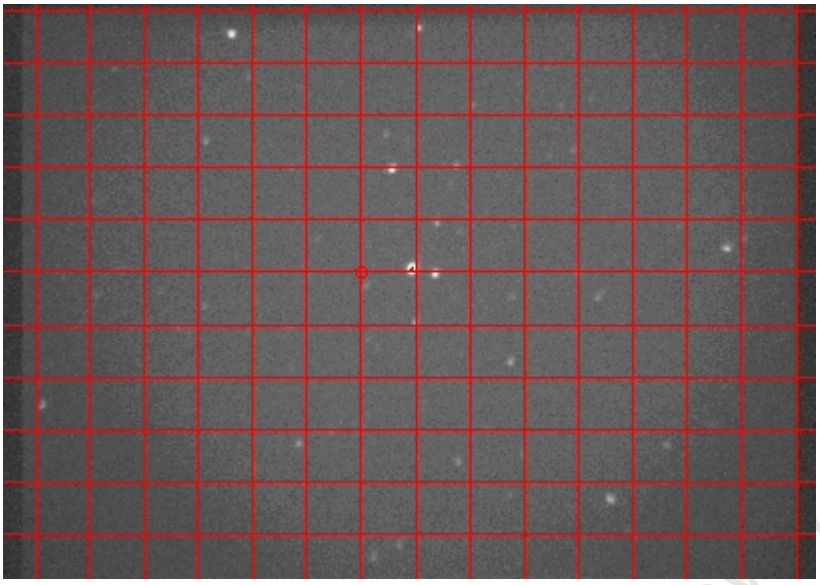

Figure 9.5: `Equatorial` grid type overlay over acquisition image (at 0° in declination)

Equatorial

The `Equatorial` grid type displays a grid according to the current telescope right ascension and declination. The declination grid-lines are separated by $1'$. The right ascension grid-lines are also separated by $1'$ (i.e. 4 seconds in right ascension) at the equator, but the spacing increases near the Southern and Northern poles - the spacing is always an integer number of arcminutes and has a maximum value of 1 hour at the poles.

The *OpenGL* matrix (specifically, the "modelview" matrix) is manipulated such that the virtual "camera" in fact points toward that point on a virtual unit sphere corresponding to the current telescope right ascension and declination. The acquisition programme then draws the grid as if it was a section of a mesh sphere with unit radius. The modelview matrix is stored for later use (see §9.2.4) - in fact, even if the equatorial grid display is disabled, the matrix is still manipulated accordingly and stored for later use.

Figures 9.5 and 9.6 show the equatorial grid at the equator and near the Southern pole, respectively.

9.2.3 On-screen Markers

A facility by which points on the acquisition image may be marked, was implemented. This facility is useful if, for instance, the user wishes to highlight a particular star in the field. The programme supports an arbitrary number of markers and the user may add and remove markers and move individual markers at will. The same facility is used to label stars that are identified as part of the pattern-matching facility - although this function is only for the user's convenience and plays no role in the pattern matching process. The pattern matching facility is described in §9.3.

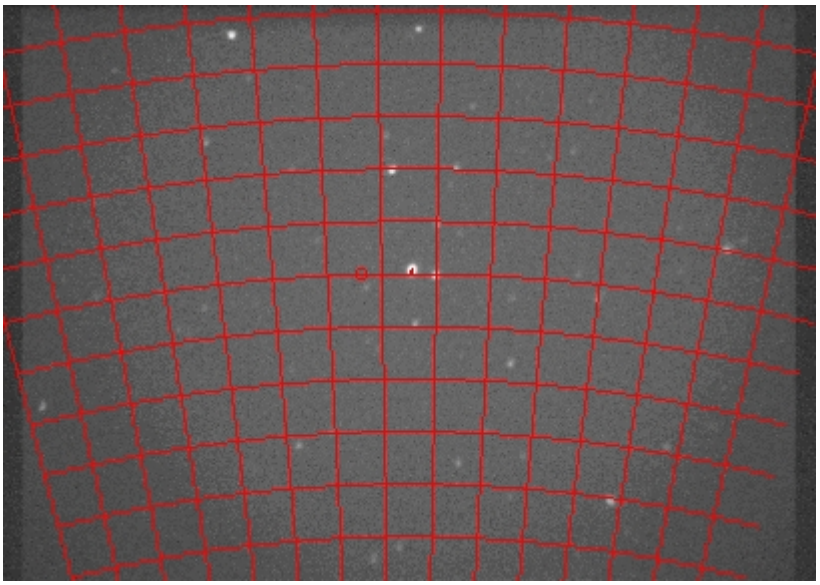

Figure 9.6: `Equatorial` grid type overlay over acquisition image (at $-89.5°$ in declination)

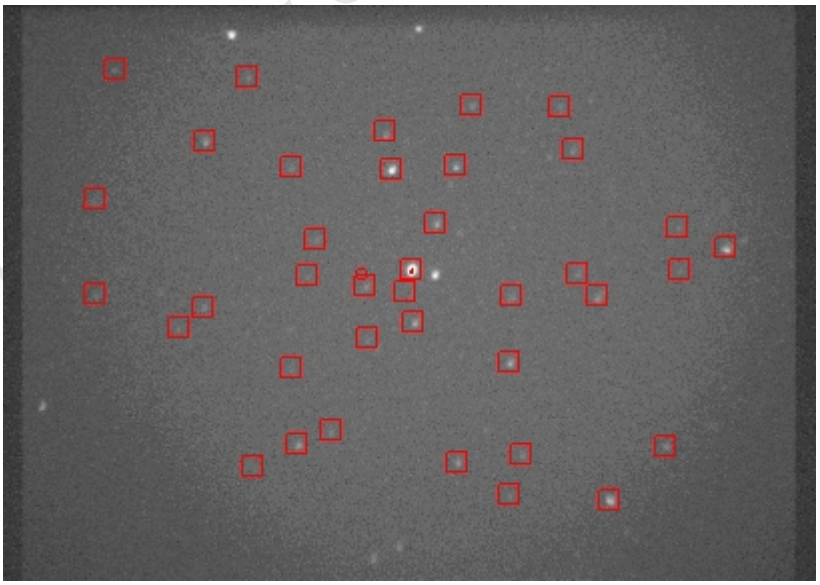

Figure 9.7: Acquisition image with identifiable stars highlighted using on-screen markers.

Aperture Centre

The acquisition programme also provides a special on-screen marker which indicates the centre of the photometric aperture. This was implemented so the user can easily align the target with the aperture, which may not necessarily co-incide with the centre of the acquisition image. At the moment, the position of the centre of the aperture on the acquisition image is hard-coded in the acquisition programme at the acquisition image pixel coordinate $(180, 135)$, as was emperically determined during the original commissioning phase of the ACT. This function was implemented in a generic way in the source code so the centre(s) of the apertures can be specified after it is determined during commissioning. Note that the centre-of-aperture circle is offset from the target in Fig. 9.2-9.7 because those acquisition images were captured for demonstration purposes only and the target was not aligned with the aperture.

9.2.4 Coordinate Display

The transformation matrix used to project an equatorial grid onto the screen (see §9.2.2) is stored during the stage where all graphical elements are drawn onto the screen mostly so that the reverse action can be performed - i.e. to specify a pixel coordinate on the screen (specifically, the acquisition image) and calculate the position in the virtual, three-dimensional *OpenGL* drawing volume that corresponds to that point on the screen. A function in the *OpenGL* Utility Library ("GLU" - an extension of *OpenGL*) provides this functionality.

In order to do these calculations, the last-used set of *OpenGL* matrices is retrieved and passed as parameters to the GLU de-projection function along with the desired (x, y) co-ordinate on the screen and another parameter which represents the depth (z) within the three-dimensional drawing volume. It is assumed $z = 1$ and, as it turns out, the value of z is trivial if the only information sought is the right ascension (α) and declination (δ) corresponding to the specified (x, y) coordinate. The de-projection function returns the coordinate (x', y', z') corresponding to the specified (x, y, z). The right ascension and declination can then be calculated by simply transforming the Cartesian coordinate (x', y', z') to spherical coordinates (α, δ, r) - although r can be omitted in this case, as it depends only on the choice of z. The right ascension and declination are then calculated from:

$$\alpha = \tan^{-1} \frac{x'}{z'}$$

$$\delta = \tan^{-1} \frac{z'}{y' \cos \alpha}$$

This facility is useful for calculating the right ascension and declination of a point or star in the acquisition image based on its position on the acquisition image and for displaying the right ascension and declination of the mouse pointer (when the pointer is over the acquisition image) to the user. The latter function (coordinate display) is used to give a readout of the

current right ascension and declination of the mouse pointer to the user.

9.3 Pattern Matching

The pattern matching routines are used to ensure that the correct target is being observed and therefore forms a crucial component of the robotic operations of the telescope. The algorithm in use in the acquisition programme was developed by Greg Cox, underwent significant testing and has been successfully used for several years on a previous telescope control system of the ACT. [4]

Although a full and in-depth description of the algorithm and its theoretical basis falls beyond the scope of this document, a brief summary of the algorithm is given here. For further information, the reader is referred to Cox [4].

The first step of the algorithm is to identify all stars within the acquisition image, which is done by first finding all the regions of the acquisition image with enhanced brightness (these are referred to as *blobs* in the algorithm). The list of stars is then constructed from the list of *blobs*, by discarding those *blobs* that are probably not stars - in this context, it is better to discard too many *blobs* than too few. Discarding too many of the *blobs* could, at worst, cause the pattern matching routines to fail outright. However, not discarding enough of the *blobs* could lead to a false positive, which could, in turn, cause the pattern matching routines to identify the field as the correct field when it is not.

Once the list of stars has been extracted from the acquisition image, it must be compared to a pattern template. These templates contain the (x, y) pixel coordinates of the stars in the field when the target is centred on the aperture. Therefore, by moving the telescope such that the same stars are at the same (x, y) coordinates on the acquisition image as they are in the template file, the target will be centred on the aperture.

Of course there is no guarantee that the exact same set of stars from the template file will also be present in the list of stars from the acquisition image, nor that the two lists will be in the same order. It is therefore necessary to find a mapping between the stars in the two lists, which is done by considering the distances between the points in the list of template stars and the list of acquisition image stars.

The offset (in pixels) of the target from the centre of the aperture can then be calculated by taking the average of the offsets of the stars in the list of template stars from the stars in the list of acquisition image stars. The de-projection procedure used to determine the right ascension and declination of pixels of the acquisition image (see §9.2.4) is then used to determine the right ascension and declination to which the telescope must be moved in order to centre the target in the aperture.

Of course the procedure described above depends on the availability of a template. Therefore, templates should be created for those targets that will be regularly observed either during commissioning or before the target is observed for the first time. A template can be created by clicking the `Mark Stars` button on the *graphical user interface*. The stars will be extracted from the acquisition image and their location on the image shown using

on-screen markers (see §9.2.3). At this time the user is given an opportunity to store the pattern template on disk. Alternatively, it should be possible to construct a template file artificially from electronic catalogues of stars, although this has never been tested.

9.3.1 Automated Observations

When the acquisition programme receives an observation message that indicates that the telescope is operating automatically (see §4.2.4), it requests a new acquisition image from the CCD driver with an exposure time that is estimated from the integration time specified in the observation message. This approximation is not strictly accurate and is insufficient for regular operation, since the specified integration time is as much dependant on the required time resolution of the data as it is on the apparent magnitude of the target, not to mention the apparent magnitudes of the other stars in the acquisition image field. However, the approximation should be sufficient for testing and commissioning under controlled conditions.

Once a new acquisition image has been received, the acquisition programme extracts the target identifier from the message structure and searches for a file name "<identifier>.pat" (where "<identifier>" is the target identifier) in the directory containing all the pattern template files, which is specified in the global configuration file. If the acquisition programme cannot find the pattern template file, it will return the observation message to the main controller with the error flag set, otherwise it performs the pattern matching procedure.

If less than a particular percentage (currently set to 25%) of the stars in the acquisition image and the pattern template match up or some other error related to the pattern matching procedure occurs, the acquisition programme will return the observation message to the main controller with the error flag set. Otherwise, it calculates the new right ascension and declination to which the telescope must be moved in order to have the target centred on the aperture, stores the new coordinates in the observation message structure and returns the message to the main controller.

According to the structure of the target acquisition cycle, the observation message should then be sent to the Dome, Telescope and Instrument programme (DTI) to have the telescope moved to the new coordinates, after which the message is again sent to the acquisition programme in order to repeat the pattern matching procedure. The process is repeated in this fashion until the acquisition programme makes no modifications to the right ascension and declination of the target (as specified in the observation message) to justify moving the telescope again.

9.3.2 Manual Observations

When the acquisition programme receives an observation message that indicates manual control, it only extracts the target name from the message structure, then returns the message to the main controller. For the sake of user convenience, the pattern matching system is also accessible when the telescope is under manual control. When the user clicks

the `Match Pattern` button, the acquisition programme prompts the user to select the pattern template file corresponding to the target. Once the pattern template file is selected, the acquisition programme performs the usual pattern-matching procedure. However, instead of sending the new coordinates using an observation message, the acquisition programme simply displays the new right ascension and declination on the screen, which can then be entered into the DTI manually in order to centre the target on the aperture.

9.3.3 Testing

The components of the acquisition programme related to the pattern matching facility underwent rigorous testing. For this purpose, a simple simulator programme (acqsim) was written. Acqsim enables the user to load a *FITS* file containing a previously recorded acquisition image and to insert a right ascension and declination offset in the image. Acqsim then shifts the pixels of the loaded acquisition image by a number of pixels (horizontally and vertically) corresponding to the specified right ascension and declination offset, padding with 0-valued pixels where necessary, and uploads the simulated image to the CCD driver (provided that the CCD driver was compiled with the `ACQSIM` flag enabled - see §A.2.2). Acqsim also allows the user to specify the right ascension and declination of the field centre (i.e. the simulated telescope coordinates) so that the acquisition programme can be tested at all right ascensions and declinations and not necessarily just the right ascension and declination of the target that was observed when the acquisition image was recorded.

During such tests, the acquisition programme must be able to reproduce the offsets entered into acqsim by the user, to within an error that is as large as the pixel resolution of the acquisition image. Naturally, the right ascension pixel resolution would be larger near the celestial North and South poles (in fact, larger by a factor of $\sec\delta$) than near the celestial equator, whereas the declination pixel resolution is the same everywhere on the sky.

Several dozen tests were performed using several distinct recorded acquisition images, various offset right ascensions and declinations and various telescope right ascensions and declinations. At offsets of several arcseconds, with the telescope coordinates set to a point near the equator, the acquisition programme always calculated the correct offset within an error no greater than $1''$, which is less than the pixel resolution near the equator. The test was repeated at various declinations and the result was always that the error in the offset calculated by the acquisition programme is less than the pixel resolution of the acquisition image (after applying a $\sec\delta$ correction to the right ascension pixel resolution). Many more tests were conducted (at low and high declination) where the offset in the simulated image was gradually increased in order to find the largest offset for which the acquisition programme can still find the target.

From the tests, it was concluded that the acquisition programme either identifies the pattern successfully and calculates the offset within an error no greater than the pixel resolution, or fails to find the correct mapping of stars between the template list of stars and the acquisition image list of stars due to the fact that too few of the stars are within the acquisition image (due to the large offset). The maximum offset for which pattern matching

can successfully be done therefore depends on the number of stars in the acquisition image and the distribution of stars around the target. For instance, a field with 20 recognisable stars is more likely to be matched correctly than a field with 10 stars and a field with most stars clustered toward the South-West of the target will have less chances of being matched accurately when the telescope is to the North-East of the target than the South-West.

In order to ensure that the target is correctly identified, it was decided to implement a minimum cut-off for the proportion of stars from the pattern file that must be matched against the stars in the acquisition image in order for the acquisition programme to accept the match as correct. The results of the tests described above suggest that cut-off of 25% is a good starting-point, although this figure would need to be refined during the commissioning of the pattern-matching facility of the acquisition programme.

9.4 Forward Look

The acquisition programme has undergone significant testing both off-line in the laboratory and on-line at the telescope and has been proven to fulfil its functional requirements effectively and efficiently. However, the pattern matching facility has not yet been tested under operational conditions and doing so should be a high priority.

Furthermore, there are several features that would enhance the operation of the acquisition programme and the software system as a whole, but have not yet been implemented due to time constraints, such as:

- A facility which selects an exposure time for the acquisition CCD automatically, based on the cumulative size of under-exposed and over-exposed regions of the acquisition image and the number of recognisable stars within the image, would be a useful addition to the acquisition programme.

- In the absence of a human operator, it may be useful to have the acquisition programme perform automatic telescope focus, however doing so will require an iterative and potentially lengthy process be done, which may not be viable in an operational context.

- If the acquisition programme fails to find the pattern template file for the specified target, rather than abandoning the target and reporting an error, it should construct a template file using information from electronic stellar catalogues available on the internet.

- Furthermore, if the pattern matching procedure fails because too few stars from the pattern file can be matched to stars in the acquisition image, the acquisition programme should either increase the exposure time of the acquisition images or construct a template file that covers a larger area surrounding the target using information from electronic stellar catalogues available on the internet.

Chapter 10

Miscellaneous Applications

10.1 ACT Common Library

The ACT common library (ACT common) is a development library that contains implementations of various algorithms that are frequently used by a number of programmes in the ACT software suite, such as:

- Conversion algorithms for angles in radians, degrees and hours.
- Conversion algorithms for angles in hours, minutes, seconds, milliseconds; degrees, arcminutes, arcseconds; fractional hours and fractional degrees.
- Conversion algorithms for equatorial and horizontal coordinates.
- Algorithms to calculate right ascension from hour angle (at a particular sidereal time) and vice-versa.
- Algorithms to calculate universal time (UT) from local time (LT) in an arbitrary time zone.
- An algorithm to calculate the Julian date at a particular calendar date and time.
- An algorithm to determine parameters related to the Sun (Heliocentric correction to the Julian date, apparent right ascension and declination and Earth-Sun distance).
- An algorithm to calculate parameters related to the Moon (namely the right ascension and declination of the Moon).
- An algorithm to precess target coordinates from one epoch to another.

The ACT common methods also perform checks to ensure the validity of the input and output and that the input and output are within particular ranges, where applicable (for instance, right ascension should always be between 0 h and 24 h and the ACT common methods always ensure that the input and/or output right ascension is within this range).

The ranges of the output of these methods may not necessarily be the range required by the calling application. However, even in this case, it is easier for the calling application to put the output from the ACT common method into the required range if the output from the ACT common method is within a definite range.

A further advantage of grouping such methods in a single library is that, should any of the methods be updated for whatever reason, any programmes that use those methods of the library automatically use the newer version of the method (although, if the programme is statically linked against the ACT common library, it would have to be recompiled for the changes to the library to take effect in that programme).

Most of these methods have fairly trivial implementations. Some of the methods that required additional attention and/or research (conversion between hour angle and right ascension, conversion between horizontal and equatorial coordinates, Julian date calculation, Solar parameters, Lunar parameters and precession of coordinates) are described below.

10.1.1 Hour Angle and Right Ascension

At a specified sidereal time, the hour angle is calculated from the right ascension using equation (10.1).

$$HA = SIDT - \alpha \tag{10.1}$$

Where: α is the right ascension
$SIDT$ is the sidereal time
HA is the hour angle

In both conversion methods in ACT common, the output is always shifted so the hour angle is between -12 h and 12 h and the right ascension is between 0 h and 24 h.

10.1.2 Horizontal and Equatorial Coordinates

Vincent [24] gives a set of 3 equations that can be used to calculate the horizontal coordinates (altitude and azimuth) of a target or the telescope from its equatorial coordinates (hour angle and declination).

$$\sin(Alt) = \sin(\delta)\sin(\phi) + \cos(\delta)\cos(\phi)\cos(HA) \tag{10.2}$$

$$\sin(Azm) = -\frac{\sin(HA)\cos(\delta)}{\cos(Alt)} \tag{10.3}$$

$$\cos(Azm) = \frac{\sin(\delta) - sin(\phi)sin(Alt)}{\cos(\phi)\cos(Alt)} \tag{10.4}$$

Where: Alt is the altitude
Azm is the azimuth
δ is the declination
HA is the hour angle
ϕ is the geographic latitude of the telescope

The telescope altitude can then be calculated as follows:

$$Alt = \sin^{-1}\left[\sin(\delta)\sin(\phi) + \cos(\delta)\cos(\phi)\cos(HA)\right] \tag{10.5}$$

The telescope azimuth can be calculated by dividing equation (10.3) by equation (10.4):

$$\begin{aligned}\tan(Azm) &= -\frac{\left[\frac{\sin(HA)\cos(\delta)}{\cos(Alt)}\right]}{\left[\frac{\sin(\delta)-sin(\phi)sin(Alt)}{\cos(\phi)cos(Alt)}\right]} \\ Azm &= \tan^{-1}\left[-\frac{\left[\frac{\sin(HA)\cos(\delta)}{\cos(Alt)}\right]}{\left[\frac{\sin(\delta)-sin(\phi)sin(Alt)}{\cos(\phi)cos(Alt)}\right]}\right]\end{aligned} \tag{10.6}$$

Vincent [24] also gives a set of 3 equations that can be used to calculate the equatorial coordinates (hour angle and declination) of a target or the telescope from its horizontal coordinates (altitude and azimuth).

$$\sin(\delta) = \sin(Alt)\sin(\phi) + \cos(Alt)\cos(\phi)\cos(Azm) \tag{10.7}$$

$$\sin(HA) = -\frac{\sin(Azm)\cos(Alt)}{\cos(\delta)} \tag{10.8}$$

$$\cos(HA) = \frac{\sin(Alt) - sin(\phi)sin(\delta)}{\cos(\phi)\cos(\delta)} \tag{10.9}$$

The telescope declination can then be calculated as follows:

$$\delta = \sin^{-1}\left[\sin(Alt)\sin(\phi) + \cos(Alt)\cos(\phi)\cos(Azm)\right] \tag{10.10}$$

The telescope hour angle can be calculated by dividing equation (10.8) by equation (10.9):

$$\begin{aligned}\tan(HA) &= -\frac{\left[\frac{\sin(Azm)\cos(Alt)}{\cos(\delta)}\right]}{\left[\frac{\sin(Alt)-sin(\phi)sin(\delta)}{\cos(\phi)}\right]} \\ Azm &= \tan^{-1}\left[-\frac{\left[\frac{\sin(Azm)\cos(Alt)}{\cos(\delta)}\right]}{\left[\frac{\sin(Alt)-sin(\phi)sin(\delta)}{\cos(\phi)}\right]}\right]\end{aligned} \tag{10.11}$$

The corresponding conversion methods in ACT common always convert input angles from

fractional degrees or fractional hours (as appropriate for altitude, azimuth, declination and hour angle) to radians, which is the unit used internally, and shift the angles so they are between 0 and 2π. The output of these methods are also shifted to the proper range, i.e. $\delta \in [-90°, 90°]$, $HA \in [-12^{\mathrm{h}}, 12^{\mathrm{h}}]$, $Alt \in [-90°, 90°]$ and $Azm \in [0°, 360°]$, if necessary.

10.1.3 Julian Date

The Julian date (JD) is calculated using the algorithm given by Meeus [17] (pg. 337-347). The results of the algorithm, as implemented in the ACT software, were checked against reliable sources - such as the United States Naval Observatory's "Calendar Date to Julian date Converter", which is available on their website*, by comparing the Julian dates calculated by the ACT JD converter and the USNO online converter on 10 randomly selected calendar dates and times between the years 1800 and 2100. The disadvantage of the USNO online converter is that it specifies the JD with only 10^{-5} d precision, which is why alternative sources were sought. One such source was found at the website of the Physics and Astronomy Department† of the Stephen F. Austin State University (SFA) in Texas, USA, which specifies the JD to 10^{-10} d precision. Seven randomly selected calendar dates and times were selected and the JDs given by the ACT converter compared to those given by the USNO online converter and the SFA online converter. The ACT converter and the SFA online converter gave the same JDs as the USNO online converter to a precision of 10^{-5} d and the ACT converter gave the exact same JDs as the SFA online converter to a precision of 10^{-10} d (barring rounding errors, which are at most 1×10^{-10} d). It can therefore be concluded that the ACT JD calculator is accurate to within 1×10^{-10} d.

10.1.4 Solar Parameters

Meeus [17] (pg. 337-347) gives an algorithm to calculate the Heliocentric correction (HC), which is added to the current Julian date (JD) in order to calculate the Heliocentric Julian date (HJD), as well as the right ascension and declination of the Sun and the distance between the Earth and the Sun.

The right ascension and declination of the Sun (as calculated by the ACT common method) was checked against the tabulated values given in the Astronomical Almanac [23] (pg. C6-C20) on 24 randomly selected days of the year 2011 at a time of 00 : 00 : 00 UT and against the values from the *XEphem* programme‡ on 10 randomly selected days between the years 2000 and 2020 at various times. It was concluded that the ACT common method for calculating Solar parameters calculates the right ascension and declination of the Sun within accuracies of 3^{m} and $4'$, respectively.

The HJDs calculated by the ACT common method were also checked against the HJDs calculated by *XEphem* at 26 randomly selected dates and times, from which it was concluded that the ACT common method calculates HJDs to within an accuracy of 3×10^{-5} d.

*http://www.usno.navy.mil/USNO/astronomical-applications/data-services/cal-to-jd-conv
†http://www.physics.sfasu.edu/astro/javascript/hjd.html
‡http://www.clearskyinstitute.com/xephem/

10.1.5 Lunar Parameters

The "low-precision formulae for topocentric coordinates of the Moon" (which incorporates the "low-precision formulae for geocentric coordinates of the Moon"), as given in the Astronomical Almanac [23] (pg. D22), are used to calculate the right ascension and declination of the Moon. These formulae should be accurate to $1^{\mathrm{m}}.2$ in right ascension and $12'$ in declination over the period 1900-2100 [23] (pg. D22). The accuracy of the results of the ACT common method was demonstrated by comparing the right ascension and declination of the Moon (as calculated by the ACT common method) against those given by *XEphem* on 10 randomly selected dates between 2010 and 2020, which showed that the ACT common method agrees with *XEphem* to within $1^{\mathrm{m}}.1$ in right ascension and $11'$ in declination.

10.1.6 Coordinate Precession

The method that precesses the coordinates of astronomical objects from one equinox to another was copied from the Linus160 programme previously used on the ACT. The output of the method was checked against the ideal answers given in Meeus [16] (pg. 66-67), which showed that the ACT common method is accurate to within 0.3^{s} in right ascension and $1''$ in declination, when the difference in equinox is less than ~ 50 years.

A higher precision formula may be implemented in future, although the errors in the algorithm used by the method to precess the coordinates is less than the proper motion of many stars, so a more accurate overall result would be produced by accounting for the proper motion of stars as well.

10.2 Time and Telescope Coordinates

This programme (referred to as the time-and-coordinates programme) was designed to ensure that the current time and telescope coordinates are available in a variety of forms to other clients that might need them and that this information is displayed to the user.

The input to and output from the time-and-coordinates programme are depicted schematically in Figure 10.1. Figure 10.2 shows the *graphical user interface* (*GUI*) of the current operational version of the programme.

The time-and-coordinates programme reads both the local time and sidereal time from the photometry-and-time driver via the appropriate ***Input-Output Controls*** (***IOCTLs***). The programme also reads the current system date and time.

The programme infers the current local date and time from the external time signal and the system date and time, which, in turn, is used to calculate the universal date and time - functions from the ACT common library (see §10.1) are used to perform these calculations. The geocentric Julian date and heliocentric Julian date are also calculated using ACT common methods. The apparent sidereal time requires no further processing and is used as-is.

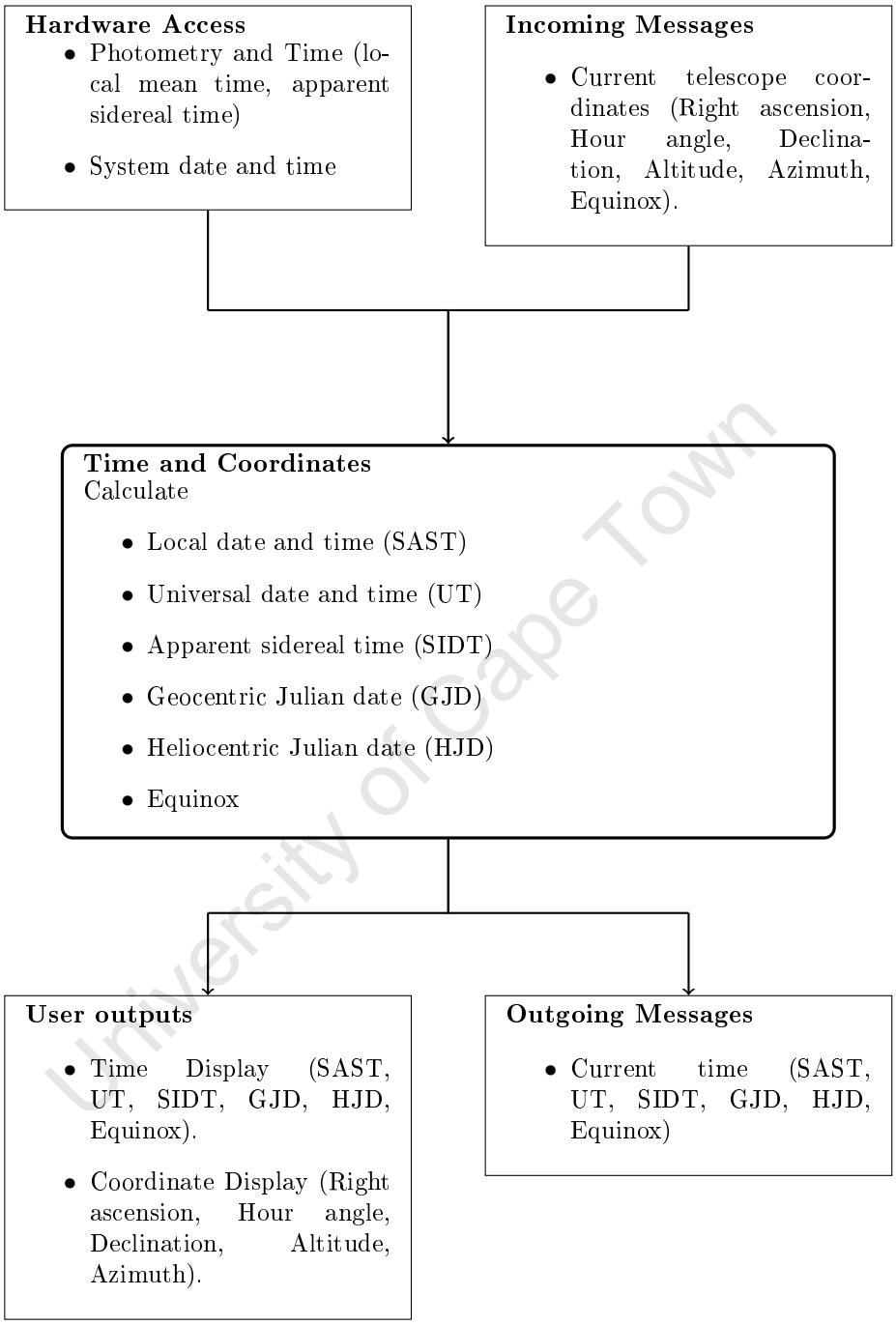

Figure 10.1: Diagram of time-and-coordinate programme inputs, tasks and outputs. Note that, although the equinox at which the telescope coordinates are given is received, the current equinox is calculated (from the current date and time) and displayed.

(a)

(b)

(c)

Figure 10.2: Screenshots of the time-and-coordinates programme's GUI and accompanying GUI elements. Panel 10.2a shows the GUI of the current operational version of the time-and-coordinates programme. Panel 10.2b shows the enlarged right ascension that is displayed in a movable popup window when the button displaying the right ascension on the programme GUI is pressed and (similarly) Panel 10.2c shows the enlarged declination. Each quantity displayed by the programme can be displayed in this way, by clicking the corresponding button on the programme GUI. The `Clear` button provides a quick way of removing all the enlarged popup windows - the alternative is to close each one individually. All images are shown to the same scale.

The times are calculated and updated on the screen once per iteration of the main loop of the time-and-coordinates programme (the period of the main loop is currently set to 1 second). The IPC resources are also checked once per iteration and the on-screen telescope coordinate display updated if new coordinates are available.

The time-and-coordinates programme had little potential for novelties and even less for future developments. The one novelty which will likely come in quite handy in future is the way the programme displays certain quantities (such as right ascension and declination) in large, bold lettering so they may easily be read from a distance and in the dark. The boxes displaying the current times and coordinates are actually buttons and not mere display objects. When a button is clicked, the quantity displayed on the button will be displayed with large lettering in a dialog box. This way, any number and any combination of quantities may be prominently displayed. See Figure 10.2.

10.3 Situational Awareness

The situational awareness programme, also known as the environment programme, is responsible for sensing all necessary parameters of the environment to ensure safe and stable operation of the telescope. This section outlines the core aspects of this programme (the sources of environmental data used, factors pertaining to operational safety of the telescope system and the programme's role during observations).

Figure 10.3 gives a schematic representation of the operation of the environment programme and Figure 10.4 shows the GUI of the programme in its current operational version.

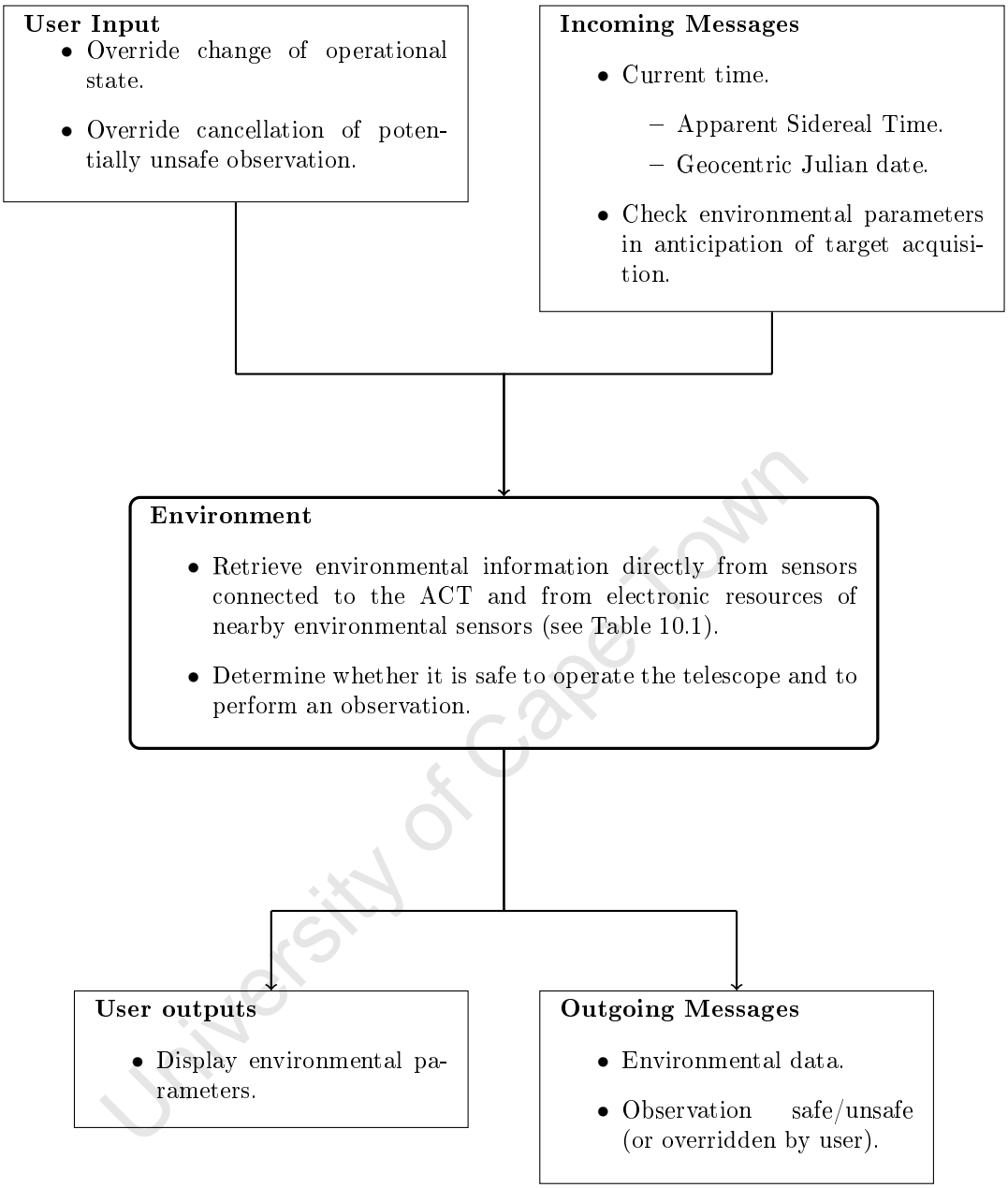

Figure 10.3: Diagram of environment programme inputs, tasks and outputs.

Figure 10.4: Screenshot of the environment programme. The SALT and SuperWASP indicators show the status of the two sources of environmental data that are currently used - the indicators are green when the data have been successfully retrieved from the corresponding sources and red otherwise. The quantities related to the safe operation of the telescope also have coloured indicators, where green indicates that the corresponding environmental parameter is within safe limits.

Parameter	Source
Exterior Temperature	SALT, SuperWASP
Relative Humidity	SALT, SuperWASP
Cloud Coverage	SuperWASP
Rain	SALT, SuperWASP
Wind Velocity	SALT, SuperWASP
Wind Direction	SALT, SuperWASP
Altitude of the Sun	Internally calculated (see §10.1.4)
Position of the Moon	Internally calculated (see §10.1.5)

Table 10.1: Environmental parameters and their sources.

10.3.1 Environmental Input

The environment programme senses environmental conditions by retrieving the tabular data provided by the Southern African Large Telescope weather system[§] (SALT weather) and *scraping* the locally available, general access website of the Super Wide Angle Search for Planets[¶] (SuperWASP weather) - see §2.6. The data from these sources are combined with each other and with environmental information and derived quantities produced by the ACT's hardware.

The values of the critical environmental parameters (those that are used to determine whether it is safe to operate the telescope - see Table 10.2) are always multiplexed from the various sources by choosing the quantity that comes closest to producing an unsafe operational environment (e.g. choosing the highest relative humidity and the highest wind speed) - barring cases where particular sources of environmental data or particular data are missing from any of the input data sets. All non-critical quantities are averaged between all available sources.

Table 10.1 lists the parameters required by the environment programme and the sources for data relating to these parameters.

10.3.2 Safety of Operations

The environment programme uses the environmental data to determine whether or not it is safe and/or advisable for the telescope to be operating and collecting data. The environment is said to be safe for operation if the Sun is below the horizon, the relative humidity is low, wind velocity is low and it is not raining. Furthermore, it is not advised that the telescope be collecting data during periods of high cloud coverage, even if the environmental conditions are safe.

A set of criteria were devised by which the environment programme can determine whether or not it is safe for the telescope to be operating and whether or not data should be collected - see Table 10.2.

If all the criteria in Table 10.2 are satisfied, the environmental conditions are deemed to be safe and advisable for operating and collecting data.

[§] `http://www.salt.ac.za/~saltmet/weather.txt`
[¶] `http://swaspgateway.suth/`

Parameter	Condition
Altitude of the Sun	$< -12°$
Relative Humidity	$< 90\%$
Rain	No rain
Wind Velocity	< 60 km/h
Cloud coverage	$< 50\%$

Table 10.2: Criteria for safe operation based on environmental conditions. These criteria will be refined during commissioning based on the accuracy of the sources of environmental data. The `Cloud coverage` criterion is not necessary for the safe operation of the telescope, however it is imposed to prevent the system from collecting photometric data under cloudy sky conditions.

The limiting values of the criteria listed in Table 10.2 are hard-coded into the environment programme, but they can be easily modified by changing the global definitions for these criteria appropriately and recompiling and restarting the environment programme.

10.3.3 Disseminating Environmental Information

The data relating to environmental parameters are communicated to the other programmes in the ACT software suite via the `Environmental Conditions` IPC message (see §4.2.4). The environment programme also indicates whether or not it is safe and/or advisable for the telescope to operate via the `Active/Idle` flag of the `Environmental Conditions` message, where a safe environment corresponds to the `Active` state. It is up to the receivers of the `Environmental Conditions` messages to determine what action is to be taken depending on the state of the flag.

The user is prompted for confirmation before the environment programme switches between the `Active` and `Idle` states, at which point the user has 1 minute to postpone the switch. The same is true during automatic operation, although in the absence of a user, the operational state will be changed 1 minute after the notification is displayed. The option to postpone the operational state change was implemented to prevent the software system simply shutting down the operational components of the telescope during a manual observation. Should the user opt to postpone changing the `Active/Idle` state, all `Environmental Conditions` messages sent by the environment programme will show the old `Active/Idle` state, but the environmental parameters causing the unsafe operational conditions will not be modified such as to seem safe to the other programmes. If the state change is postponed by the user, he/she will be granted 10 minutes grace time before being prompted again. If the environmental conditions should change for the worse quickly, the onus is on the user to manually secure the system to prevent damage.

10.3.4 Role During Observation

The environment programme should be one of the first programmes to be polled in the observational cycle - as it is in the current implementation of the software architecture, see

§4.2.4 - so that no action is taken that might put the telescope system at risk during unsafe environmental conditions (such as opening the dome while it is raining).

When the environment programme receives an observation message with the `Automatic Mode` flag activated (see §4.2.4), it determines whether or not it is safe for that particular observation to be done based on the criteria listed in Table 10.2 as well as the position of the Moon (if the specified target is either behind the Moon or within a circular region with a user-defined radius around the Moon, that particular observation is deemed to be unsafe). If a particular observation is unsafe, this fact is indicated by activating the `Error` flag of the observation message and returning the message to the main controller.

Observation messages that don't have the `Automatic Mode` flag activated (i.e. a manual observation) are sent back to the main controller immediately if the environmental conditions are safe for that particular observation. Should the environmental conditions be unsafe, the message is temporarily stored internally and the user warned and prompted to choose whether or not to proceed with the observation. If the user chooses to proceed, a message is entered into the environment programme log and the observational message returned to the main controller unaltered. Alternatively, if the user cancels the observation, the `Error` flag of the observation message is set and the message is returned to the main controller.

10.3.5 Forward Look

A future version of the environment programme should read environmental data from more of the available sources, once the accuracy of these sources can be confirmed empirically. Furthermore, as all the available sources are fallible, the programme should be implemented with a level of "intelligence" (in terms of identifying and ignoring incorrect data). However, such an implementation would require significant testing in order to verify that it is effective.

Chapter 11

In-Situ Validation of Software

This chapter outlines the tests that were performed at the telescope in order to ensure that all components of the hardware and software are performing correctly. Section 11.1 describes the checks that were performed to check that all hardware components function correctly and that the software interfaces with them correctly. Section 11.2 describes the tests performed in order to validate the software components related to target acquisition. Section 11.3 describes the tests performed in order to validate the the software components related to photometric data acquisition. Lastly, some issues related to the optical alignment and hardware of the telescope were exposed during the course of the validation tests - these are outlined in §11.4

11.1 Preliminary Checks

All components of the telescope that must be controlled by the telescope control software had to be tested. This was done by going through the list of components of the telescope system and their functions and testing each function of each component. These components and their functions are listed in Table 11.1.

Once these basic tests were passed, several more advanced tests were performed, such as:

- Establishing the telescope's operational limits in hour angle and declination.
- Confirming that the telescope software limits are properly observed.
- Finding all combinations of telescope orientation and dome orientation that may lead to a collision (or otherwise cause damage to the hardware) and accounting for them by adjusting the software limits appropriately.
- Confirming that the dome is aligned with the telescope at all accessible parts of the sky.

Component	A/F	Function
Dome shutter	A	Open dome shutter
	A	Close dome shutter
	F	Dome shutter is open
	F	Dome shutter is closed
Dome dropout	A	Open dome dropout
	A	Close dome dropout
	F	Dome dropout is open
	F	Dome dropout is closed
Dome rotation	A	Rotate dome
	F	Dome azimuth encoder
Trapdoor	F	Notification when trapdoor to dome open and dome rotation stopped
UPS	F	Uninterrupted Power Supply activates in case of main power failure and notification issued
Watchdog	F	Notification issued and dome shutter closed when no commands received by PLC in 5 minutes
Acquisition mirror	A	Mirror moved to measure position
	A	Mirror moved to view position
	F	Mirror is in measure position
	F	Mirror is in view position
EHT	A	Change photomultiplier tube high-voltage power supply mode
	F	Detect photomultiplier tube high-voltage power supply mode
Instrument shutter	A	Open instrument shutter
	A	Close instrument shutter
	F	Instrument shutter is closed
	F	Instrument shutter is open
Filter wheel	A	Cycle through available filters
	F	Detect current filter
Aperture wheel	A	Cycle through available apertures
	F	Detect current aperture
Telescope focus	A	Move focus (secondary mirror) in and out
	F	Detect focus position
Handset	F	Detect handset focus buttons pressed
	F	Detect telescope move buttons (cardinal directions)
	F	Detect telescope speed switch (slew, set, guide)
Telescope Motion	A	Move telescope North, South, East and West at various rates
	F	Detect telescope pointing encoder readouts
	F	Detect telescope electronic limits triggered; telescope is stopped by drive electronics and encoders zeroed at Northern and Western limits.

Table 11.1: List of computer-controlled hardware components and their functions. The hardware components are listed in the first column. The second column specifies whether the function being tested is an action (A) or sensor feedback (F) function. The third column lists the actions and feedbacks.

Table 11.1 cont.

Acquisition CCD	A	Record acquisition images with various exposure times
	F	Acquisition images recorded correctly and correctly oriented
Photometry	F	Detect pulses from photomultiplier tube
Time	F	Keep accurate time

Limit	**Position**
North	declination $43°7'31''$
South	declination $-104°36'17''$
East	hour angle $-5^h1^m13^s5$
West	hour angle $5^h17^m45^s0$

Table 11.2: Hour angle and declination of telescope electronic limits.

- Confirming that the telescope initialisation procedure (as outlined in §8.4 - in particular calibrating the pointing encoders) is done effectively and safely. This must hold even if the telescope and dome were not properly parked and if the the pointing encoders were erroneously zeroed (such as during a main power failure).

These tests were all completed and a positive result obtained in each case.

11.2 Target Acquisition

Communications with the acquisition CCD electronics had already been confirmed during the preliminary hardware checks. The next step was to use the acquisition CCD to observe a star field. This required that the telescope focus be moved such as to produce clear and sharp images. It was found that a focus position of ~ 260 units in the focus "out"-region produced the best results (see Figure 11.1).

Dome autoguiding was checked by moving the telescope to different parts of the sky and verifying that the dome is automatically rotated so as to not obstruct the telescope.

The finder telescope was aligned with the telescope by pointing the telescope to an easily recognised bright star, finding and centering it in the acquisition CCD field and adjusting the finder telescope as necessary. During these early stages, the telescope pointing relied on the previously established hour angle and declination of the optical limit switches (electronic limits). After the finder telescope was aligned, the hour angle and declination of the limit switches were refined using the tabulated coordinates of bright constellation stars compared to the telescope hour angle and declination read out given by the software and extrapolating the coordinates of the electronic limits. The empirically determined coordinates of the telescope limits are given in Table 11.2.

As the pattern matching facility of the software had not yet been implemented at the time of validation tests described here, no in-situ validation of the pattern matching facility has yet been done. However, given the extent of the simulated tests that were performed on the pattern matching algorithm, it is highly unlikely that any errors in the algorithm will be exposed when (in the near future) the in-situ validation of the algorithm is done.

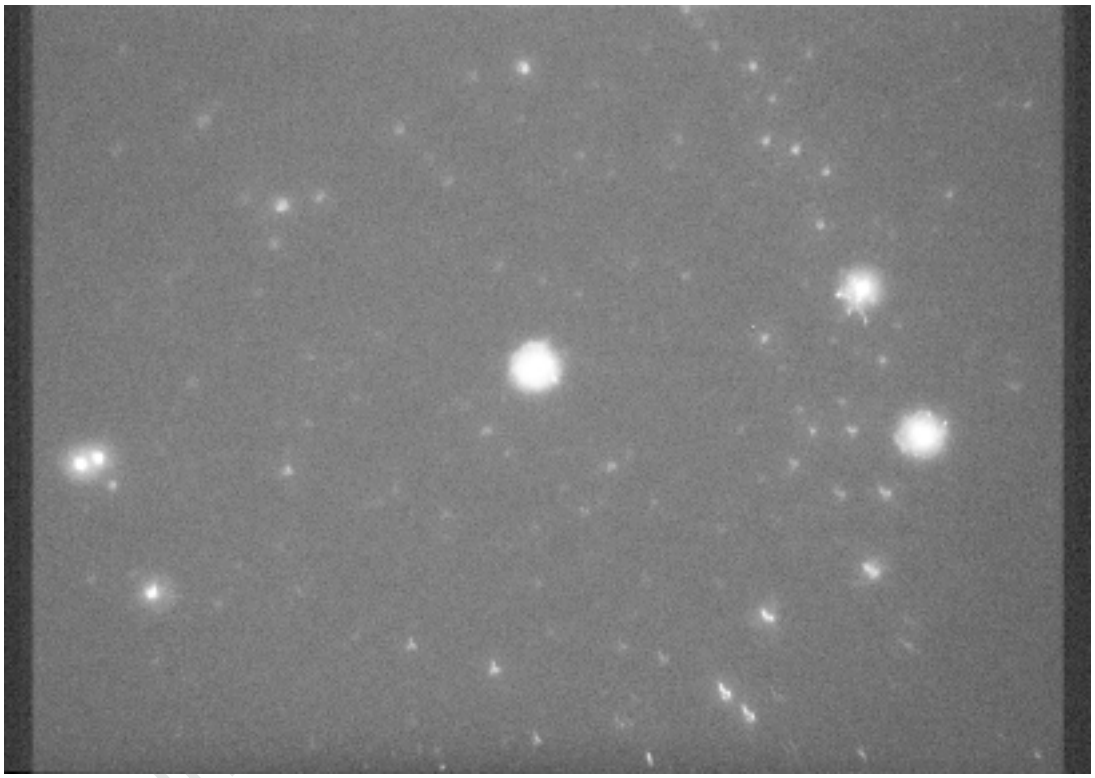

Figure 11.1: Arbitrary star image taken with the ACT acquisition system while validating the software.

11.3 Data Acquisition

Communications with the photometry hardware had already been confirmed during the preliminary hardware checks. However, the dark counts displayed by the photometry programme were significantly higher than expected (several hundred counts per millisecond rather than several tens of counts per millisecond). This condition persisted even after the thermoelectric cooling system was left running for several days, which is more than enough for the cooling system to cool the PMT to the required $\sim -10^{\circ}$ C. The source of the problem was traced back to the "coldbox" of the cooling system, which seemed unable to cool the PMT to temperatures less than $2-3^{\circ}$ C below the ambient temperature.

The coldbox was sent to Cape Town to be fixed and was only returned shortly before the software validation process came to an end. As such, there was insufficient time to perform further validation tests of the data acquisition system using astronomical sources.

11.4 Hardware Issues Exposed

During the course of the validation tests, some technical issues related to the telescope hardware were exposed, namely slipping of the pointing encoders and the telescope collimation. A detailed discussion of these issues falls outside the scope of this project. Here we give only a brief description of the problems and note that they were identified with the aid of the ACT control software during its validation tests.

11.4.1 Slipping Encoders

While establishing the coordinates of the telescope electronic limits (as outlined in §11.2), it was noted that there was a discrepancy between the initial and final coordinates of a star (as calculated by the software from the pointing encoders) when pointing at the star, pointing several degrees away from the star and then pointing at the star again. Extensive tests were performed that confirmed this effect, but attempts to quantify the nature of the effect were only partially successful.

The source of the problem was traced back to the mechanism for mounting the telescope pointing encoders. Attempts to find a quick fix to this problem were unsuccessful. The encoder mounts are in the process of being redesigned and the new mounts should be installed at the telescope toward the end of 2011.

11.4.2 Collimation of Primary and Secondary Mirrors

The out-of-focus images suggested that the telescope mirrors were not very well collimated. It was later discovered that there is a declination dependence in the collimation, which indicated that one (or more) or the optical components are shifted or disoriented when the telescope is moved. The primary and secondary mirrors will be reseated on their supports as a first attempt to solve this problem.

Chapter 12

Conclusion

12.1 Assessment of Control System Performance

The aim of the project was to develop the control software for the Alan Cousins Telescope. The software is still a work in progress and it will take several years of operational experience to improve and refine the control system.

Through the course of this thesis, this singular goal was analysed in several layers of increasing detail and a set of concepts for solving the larger problem were developed. These concepts, in turn, were developed to produce a series of tasks - i.e. the design, implementation and testing of software components - which were completed with much overall success.

The success of the outcome of the project (i.e. the telescope control software) may be gauged by answering the questions listed in §3.4.

12.1.1 Performance in terms of Data Collection

The software components involved with collecting and recording photometric data underwent extensive testing under controlled conditions, which showed that these components performed as expected. Unfortunately, due to technical difficulties encountered during the software validation phase (see §11.4), no photometric data of astronomical targets could be collected in order to confirm the correct performance of the data collection components of the software in practice.

12.1.2 Reliability of Collected Data

Extensive testing under controlled conditions showed that the recorded data are very reliable. In fact, the data are absolutely reliable with the exception of the mysterious time skews observed during the testing phase of the photometry acquisition component of the software (see §B.4). However, the time skews are logged so their effect can be removed from the photometric data when the data is reduced. The only other factor that can affect the quality of the collected data is atmospheric effects. These effects are logged by other instruments at

the SAAO in Sutherland - the logs can be used to gauge the photometric quality of the night when the data are reduced and analysed and structures have been put in place to enable future revisions of the data collection components of the software to do so automatically when the data are recorded.

12.1.3 Software Control of the Telescope Hardware

The software components that control various aspects of the telescope hardware underwent significant testing both under controlled conditions in the laboratory and in practice at the telescope (with the exception of the software involved with the collection of photometry, which has only been tested in the laboratory). It was found that the software controls all aspects of the telescope hardware effectively, comprehensively and safely. However, there may yet be scenarios that could damage components of the telescope system that (despite every effort) have not yet been identified and have therefore not been catered for specifically within the control software. Luckily, the DTI in particular was designed and implemented in such a way that measures to prevent such scenarios (should any be identified in the future) can be implemented relatively easily.

12.1.4 Software Stability

The software has been designed and implemented with much redundancy so that any incorrect behaviour of any of the software components would not likely lead to damage to any of the components of the telescope system. Furthermore, the modular design approach employed and, specifically, the implementation of the main controller means that any component of the software that exits abnormally due to a programming error is immediately restarted.

12.1.5 Maintainability and Upgradability

The software should be fairly easy to maintain and upgrade, due in large part to the modular design of the software as well as the structure with which individual components were designed and the ideal of consistency which guided the development of the software at all levels. Furthermore, in most cases structures have already been put in place in the source code which would simplify the implementation of the potential future improvements of individual components (as outlined in the corresponding chapters of this thesis).

In terms of user and technical documentation for the telescope software and the telescope system in general - this document is currently the most comprehensive documentation of the ACT control software and several aspects of the telescope system as a whole. As such, it will be the foundation for a planned electronic repository of all matters related to the ACT.

12.1.6 Ease of Use and Safety

The ease of use of the system depends greatly upon how experienced at observing and computer literate the user is. The ACT software should be fairly simple to use for someone with marginal observing experience who is fairly computer literate. However, the system has been developed in such a way that even if the user is too inexperienced to use the telescope system properly, he/she would still not be able to damage any component of the telescope system without trying and/or ignoring the warnings of the telescope control software.

Furthermore, all aspects of the control of the telescope system can be accessed fairly easily, which is one of the criteria that directed the design of the *graphical user interfaces* (*GUIs*). However, there are still some comparatively rarely used features of the software that are easily accessed and frequently used features that are less easily accessed. A future version of the software may have redesigned GUIs in order to improve this situation.

12.1.7 Remote Access and Robotic Operation

All software components have been developed in such a way as to allow controlling the telescope robotically and remotely.

The telescope was frequently controlled remotely during the software validation phase of the project, although the user was always in or near the dome. In future tests, the distance between the user and the telescope must be gradually increased until the telescope system may be confidently controlled from any node on the SAAO network in Cape Town or Sutherland and (preferably) anywhere around the world through the internet.

Various tests that were performed under controlled conditions have shown that the robotic control features of the telescope software work, however much more testing must be done in practice before a regular, robotic observing programme can be initiated on the telescope.

12.1.8 Summary

In summary, all components and features of the software have been shown to work either in the laboratory alone or both in the laboratory and in practice. Those components and features that have not yet been shown to work in practice will be tested under operational conditions in the near future.

Figure 12.1 shows the *graphical user interface* (*GUI*) of the ACT control software in its current form.

12.2 Future Work

Although the ACT software is functional, as always, there is room for improvement. The potential future developments of individual components of the ACT software system have already been discussed in the corresponding chapters, therefore this section serves only to

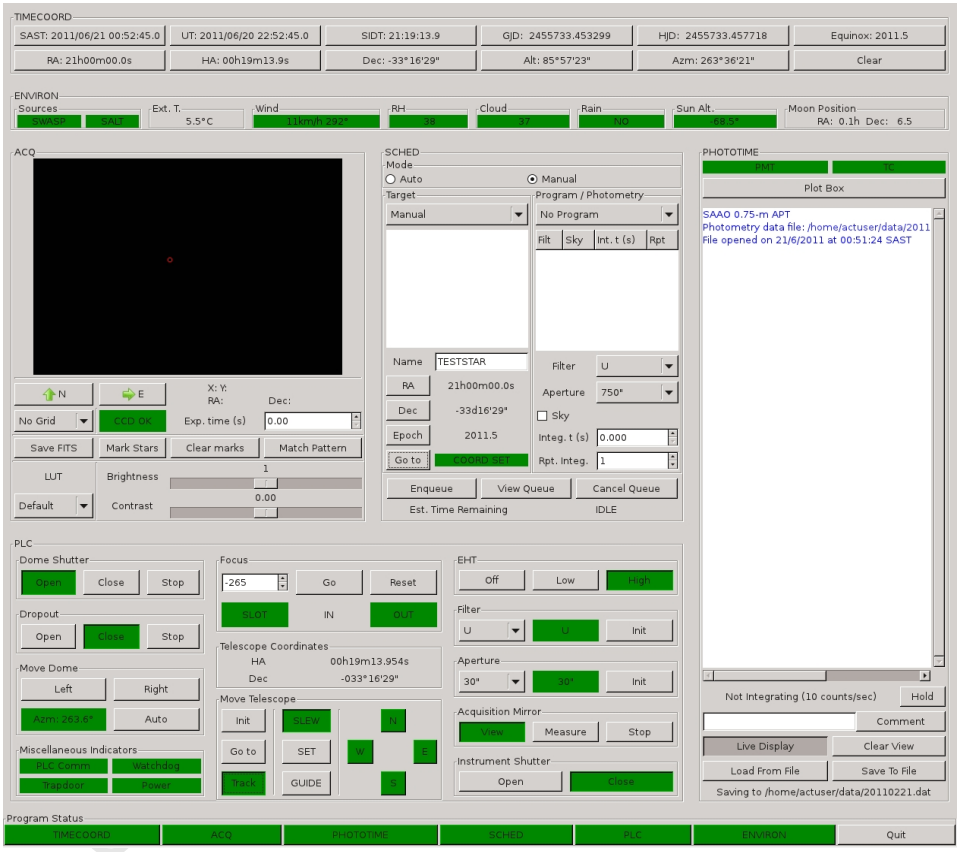

Figure 12.1: The ACT control software's *GUI*.

comment on general improvements to the ACT software system that should be considered in the future.

12.2.1 Implementation of Database System

Several telescope software systems already use sophisticated databases (such as *SQL* databases) at some level (such as storing astronomical data). However, there is no reason why databases cannot be used in a more general way - particularly storing configuration options which are currently stored in files on disk. In future, a database system should be employed not only to store photometric data (as outlined in §7.5), but to store all data that must be stored permanently on the computer (even all aspects of the software configuration).

Databases allow for symbolic links between tables and for data from separate tables to be joined together in a fraction of the time it would take to cross-reference data between data files. This can simultaneously be used to reduce duplication of data and to simplify relatively complex and/or tedious tasks related to working with the data. In practice, this means that things such as basic information about targets can be contained within one table and all recorded photometry of all targets contained within another table and the two tables joined so that each datum of photometry is paired with some or all of the stored basic stellar information of the corresponding target. In this way, tasks that are relatively tedious and/or complex to perform using scripts or programmes, such as finding out when last a standard star was observed, how many times a particular target was observed or even finding the maximum and minimum photometric datum for a particular target, can be as simple as constructing a single-line database query.

The fact that a database is given particular, definite structure when it is created means that all data contained within the database must be stored in a way that follows that structure. For instance, different users may have different naming conventions for files containing photometric data and may choose to store their data files in directories on the hard disk that are not regularly backed up, which could lead to data loss. By enforcing a particular structure, the scope of things the user can do that the developers of the system did not consider and may lead to an undesirable outcome, are reduced. Databases can also be structured to prevent duplicate entries within individual fields of the tables of the database. This can be very useful in the context of storing configuration options in a database, as configuration options can be stored as key-value pairs and the database structured so each key must be unique, which will prevent the user from inadvertently creating a second key-value pair with the same key but a different value where only one key-value pair is used by the programme requiring that configuration option.

Lastly, databases do not need to be hosted on the same computer as the control software, meaning that all configuration options can be stored in a central repository thus eliminating the need to synchronise configuration files between computers hosting components of the control software.

12.2.2 Generalisation of Components

Although generality was one of the principles guiding the development of the ACT software suite, certain aspects of the control software had to be designed and implemented specifically for the ACT. In future, these specialised components of the software could possibly be redesigned in a more general way so the software could more easily be modified to work on different telescope systems.

12.2.3 Internet Access

At the moment the telescope system can be controlled and/or monitored remotely via SSH and *NX* (see §5.8). However, a preferable alternative to enable remote monitoring only is to have each software component regularly generate HTML code that mimics the status information given by its GUI and have the main controller or some other programme multiplex the segments of HTML code produced by the various software components into a single HTML file that can be exported to a server for public access.

Appendix A

MERLIN CCD driver

This appendix outlines the kernel driver that was created to interface with the ACT's acquisition CCD. Given the sparse documentation available for the MERLIN CCD controller in use on the ACT, many of the functions implemented had to be reverse-engineered.

The CCD controller tends to lock up when input and output operations are not done in the exact way the controller was designed to do input and output. Once in this state, the controller does not respond to any input and produces no output. A hard reboot of the controller - effected by pressing a switch on the CCD-and-controller unit - is required to resolve the error. In this case, all data still buffered by the controller or still on the CCD chip are lost.

For the above reason, it was decided to develop a software interface layer for the CCD. This layer had to be at the *Kernel*-level due to *I/O* timing requirements. An *RTAI* (Real Time Application Interface) module was considered, but the notion was rejected in favour of a *Kernel* driver in order to reduce the dependence of the software system (as a whole) on *RTAI*.

The driver must be robust, must be able to communicate with the CCD controller in exactly the prescribed manner and must be easy to use (from the perspective of a developer of software that needs to communicate with the CCD through the driver).

The first section of this appendix explains how the driver should communicate with the CCD controller (henceforth referred to as "the controller") - the input the controller accepts and the output it produces. The second section describes the internal workings of the driver (how it requests CCD exposures from the controller), how user-space applications should communicate with the driver and the testing that was done to ensure that the driver works. The final section outlines potential future expansions of the driver.

A.1 Commands

A.1.1 'Z'-test

The most basic command the controller accepts is the *ASCII* 'Z' character. The controller responds to this command by returning a 'Z' character. The purpose of this is to ensure that communications with the controller has been established and probably is unnecessary in a programme not intended to diagnose communication with the controller.

A.1.2 CCD Information

The CCD unit's ID and controller software version number can be accessed by sending the 'Y' character. The controller will return the ID and version number as a string of characters. The first byte returned by the controller represents the length of the string, followed by that number of *ASCII* characters and then a byte representing the status of the CCD and controller. There is very little documentation available on the status byte; however 0 means that the operation was completed successfully.

A.1.3 Reading out the CCD

A frame can be ordered by sending the 'R' character to the controller, immediately followed by two bytes representing the exposure time. The first byte represents the low order integration time, or rather an integer (between 0 and 255) indicating how many 40 ms intervals to expose for. Similarly, the second byte represents the high order integration time as an integer (between 0 and 255) multiple of 10.260 seconds. Both the high and low order integration time bytes are sent to the CCD in their binary form (i.e. not as `ASCII` characters) and are equivalent to the data type **unsigned char** in C.

Once the exposure command and the major- and minor exposure times are received, the controller will purge the frame currently on the exposed half of the CCD and start a new exposure. After exposing for the specified time, the exposed frame will be transferred to the dark side of the CCD and the CCD will be read out. The controller reads the CCD out at a rate of 15 μs per pixel and into the controller's internal buffer. As the buffer isn't large enough to contain an entire frame, the computer must be ready to read the pixels from the CCD (via the buffer) from the moment the read-out starts and must continue reading until all pixels are read out. If the buffer is full and unable to receive new pixels, the buffered pixels will be over-written. In this case, a smaller number of pixels than expected will be read out by the computer, however there is no way of testing for such an occurrence. After all pixels have been read by the computer, the controller sends a status byte, which should always be zero (unless an error has occurred).

A.2 Kernel Driver

A.2.1 CCD Exposures

The CCD driver accepts expose commands from any user-space application connected to it. Once the driver receives an expose command, it formats the command such that the controller would accept it and forwards it to the controller.

The CCD driver *sleeps* for the same length of time as the exposure and then awaits output from the controller. As the controller needs to transfer the exposed frame to the dark side of the CCD after the exposure, the CCD driver will always be ready to accept output from the controller before the controller produces any output.

Once the controller starts to send output to the CCD driver, the driver enters a loop which reads all available CCD pixels (if any) and sleeps for several microseconds if no pixels are available. The driver stops the loop once all the pixels and the status byte have been read out. All the pixels are stored in a buffer internal to the driver. No pixel in the driver's buffer can be accessed by an application until all pixels have been read from the controller.

A.2.2 Application Interface

The CCD driver uses a combination of the character device (***chardev***) and ***Input-Output Control*** (***IOCTL***) interfaces to communicate with user-space applications. The driver status byte (which incorporates status information of the CCD-and-controller unit) is accessed by reading from the ***chardev***. CCD exposures and the pixel data of the resulting image are accessed through the ***IOCTL*** interface. An additional ***IOCTL*** was implemented by which simulated images can be loaded to the driver, which is useful for off-line testing.

Status Information

The status of the CCD driver and CCD-and-controller unit are accessed by reading from the ***chardev*** interface as if one is reading a single character from a text file. The status character is interpreted by considering its binary representation. The meaning of the bits constituting the status character (also known as the status byte) is tabulated in Table A.1.

Exposure Requests

An exposure of the CCD is requested by writing to the corresponding ***IOCTL*** of the CCD driver (designated `IOCTL_WRITE_EXP` in the CCD driver source header file), where the ***IOCTL*** parameter is a pointer to the memory location (owned by the user-space application requesting the exposure) where the desired exposure time is stored.

Once an exposure has been requested, the CCD driver activates the `CCD_INTEGRATING` flag of the status byte. After the exposure, while the CCD is being read out by the CCD driver, the driver deactivates the `CCD_INTEGRATING` flag and activates the `CCD_READING_OUT` flag. After the CCD has been successfully read out, the `CCD_READING_OUT` flag is deactivated

Bit	Name	Meaning
1	CCD_ERROR	An error has occurred on the CCD. The controller should be reset and the driver restarted.
2	CCD_INTEGRATING	A CCD exposure is underway
3	CCD_READING_OUT	The pixel data is being read from the controller
4	IMG_READY	A new frame is ready in the driver's buffer and can be read from the driver's buffer by an application. Once the application has read the entire buffer, this bit is set to 0.

Table A.1: Summary of the CCD driver status bits, which are returned to the accessing application through the driver's character device. A 0 status means the driver is idle and no error has been reported.

and the IMG_READY flag is activated, at which point the image data may be retrieved by the user-space application.

Image Retrieval

Through this interface, a user-space application can instruct the CCD driver to copy the current image to a memory location owned by the user-space application. When accessing this IOCTL (designated IOCTL_GET_IMG in the CCD driver source header file), the user-space application must provide a pointer to a memory location that is owned by the user-space application and is large enough to contain an entire image from the CCD.

This ***IOCTL*** is only accessible if the IMG_READY flag of the driver status byte is active. Any attempts by a user-space application to read an image when none is available will return a descriptive error code to that application. When an image is available, the CCD driver deactivates the IMG_READY flag once the image is completely read by the user-space application.

It should be noted that the user-space application that requested the exposure need not be the same application that reads the image from the CCD driver. Although there are ways of preventing such a situation within the driver itself, these solutions are too complex given how unlikely such a scenario is in the controlled operational environment within which the driver is to be used.

Image Simulation

During the development of the user-space acquisition system software, it became necessary to devise a means of doing off-line (i.e. without the CCD attached to the computer) testing of the software. In order to simplify such tests, another ***IOCTL*** was implemented, by which a user-space application could upload an acquisition image that had been recorded previously to the driver, thus creating a controlled, simulated environment within which the acquisition software could interface with the CCD driver, exactly as it would in practice.

A user-space application capable of uploading a simulated acquisition image to the driver would first prepare the image (which must have the same dimensions and data type as an

image originating from the CCD) in a memory location owned by it and then access the `IOCTL_SET_IMAGE` ***IOCTL*** with a pointer to the memory location containing the prepared image. The CCD driver copies the image data to its buffer that normally contains the image from the CCD.

This feature is disabled by default, but can be activated by activating the `ACQSIM` flag and compiling the driver. However, activating the `ACQSIM` flag also disables the driver's interface with the CCD controller and therefore the `ACQSIM` feature is not suitable for operational use.

A.2.3 Testing of CCD Driver

A small test application was written to test whether the CCD driver was performing as expected. In the most basic case, the test application simply requested exposures through the CCD driver and read out the frames. In this way, the CCD driver was demonstrated to work with any exposure time the CCD unit is capable of.

Further testing was done to demonstrate that the CCD driver performed as expected while the computer's CPU is under heavy load. This included running several computationally-intensive programmes in user-space and running computationally-intensive kernel-level software (such as the photometry-and-time driver) while the CCD is being read out by the driver (which is the most time-critical part of the driver's operation). In all cases, the CCD driver performed as required.

A.3 Forward Look

This driver is currently operating successfully in the ACT. Several improvements could potentially be made (in a reasonably short length of time) to enhance its performance.

A separate interface (through the /proc pseudo-filesystem) can be developed through which information about the CCD is made available to any external programs. Information such as the CCD identifier and version number, the width and height of the CCD in pixels, the on-sky size and resolution of the CCD and the minimum and maximum allowable exposure times can be included in this file. This will circumvent the need to compile these details into external programs or to save them in normal files on disk and provide a more portable way of determining the parameters of the CCD.

Given the widespread use at the SAAO of the make and model of CCD for which this driver was developed, it may be worthwhile investigating the possibility of using this driver on some of the other telescopes.

Appendix B

Photometry-and-Time driver

B.1 Introduction

The functional requirements of the telescope control software require that the software be able to collect accurate photometry with integration times as long as several minutes and as short as 1 ms. However, most modern computers lack the ability to keep accurate time at the millisecond level - a computer would require additional hardware in order to keep accurate and precise time, which is why the control computer was equipped with a specialised expansion card (time card) that connects to the time service available at the SAAO. The time service provides both the local mean time and the local sidereal time with 1 s precision as well as a train of electronic pulses with a period of 1 ms that can be used to determine the time with a precision of 1 ms.

The control computer is also equipped with an expansion card (the photometry card) that counts pulses from the photomultiplier tube (PMT). Since the collection of accurate photometry is central to the existence of the telescope and all photometric data requires an accurate and precise time stamp, it was decided to combine the software interface with the photometry card and the software interface with the time card. The collection of photometry is the most time-critical task of the telescope control software, however it is not the only task that requires correct and precise time. Therefore, this combined software interface would need to provide the local mean time and local sidereal time as well as photometric data to other programmes on the control computer that require access to this information.

Because the collection of photometry is a time-critical task of the control software, it was decided to implement the photometry-and-time interface as a Linux kernel module, which would enable the interface to operate in real time.

B.2 Design Considerations

The time card can be used to convert the 1 ms pulse train from the time service to a series of *interrupt requests* (*IRQs*) that the driver can catch in order to perform a regular

operation at the turn of every millisecond. Such regular operations would include reading the counts from the photometry card and incrementing a counter (in order to keep time with millisecond precision). The counts could then either be read from the photometry card once per millisecond or once per integration (where an integration could span 1 ms or several minutes). However, in the latter case the PMT could sustain damage during longer integrations because there would be no way of detecting an over-illumination of the PMT until the counts are read, which may be too late. Therefore, it was decided to read the counts from the photometry card once per millisecond, regardless of the desired integration time, so that an over-illumination of the PMT can be detected in time to prevent damage.

Even though the photometry card is read out once per millisecond, these measurements can be combined or binned to form an integration lasting the desired length of time. The notion of having a user-space application specify an integration time and having the driver bin the data appropriately was considered, but rejected because of complications related to how repeated observations should be handled. It was therefore decided to provide the raw data to any programmes that require them, although this would require a buffer to store the collected photometry until such time that the user-space application that handles the collection of photometry at a higher level is able to read it from the driver.

The local and sidereal times are read from the time card as 2 sets of 2 integers which are processed by a series of binary operations to yield the times in hours, minutes and seconds - these operations (along with the initialisation of the time card at the software level) were copied from Stephen Potter's photometry-and-time *RTAI* interface [20]. It was decided to export the processed times to the other programmes that require them (rather than the raw times) since the raw time is not useful in the context of a high-level application.

B.3 Application Interface

Information related to photometry and time can be accessed by user-space applications through a combination of the photometry-and-time driver's character device (***chardev***) and a number of ***Input-Output Controls*** (***IOCTL***).

B.3.1 Time

The local mean time can be read from the driver through the `IOCTL_GET_MT` ***IOCTL*** and the local sidereal time can be read from the driver through the `IOCTL_GET_ST` ***IOCTL***. In each case, the ***IOCTL*** parameter must be a pointer to a memory location owned by the calling user-space application that contains an **unsigned long int** variable. Calling one of these ***IOCTLs*** makes the driver write the mean time or sidereal time (respectively) to that variable, in milliseconds since 0h0m0s0ms. This format was chosen because it allows time data to be transferred between kernel-space and user-space simply and efficiently and is relatively simple to work with.

B.3.2 Photometry

An application can read from the driver's ***chardev*** interface as if reading a character from a file. This status character (or status byte) is interpreted in its binary form to give information concerning the status of the PMT, such as whether the the PMT is over-illuminated (i.e. the count rate from the PMT exceeds the recommended maximum) or no counts are being received from the PMT (which suggests a communication failure between the PMT and the photometry card).

A second means of detecting an over-illumination of the PMT has been implemented, which is accessed through the `IOCTL_GET_CUR_COUNTRATE` ***IOCTL***. When calling this ***IOCTL***, the user-space application must provide a pointer to a memory location owned by it, which stores a variable of type **unsigned long int** and to which the driver can write the total counts recorded during the previous millisecond. This is useful if the recommended maximum count rate of the PMT should not be absolutely adhered to, but the maximum allowable count rate is considered flexible. A user-space application intended to take steps to prevent an over-illumination of the PMT can therefore monitor either the driver's ***chardev*** or the `IOCTL_GET_CUR_COUNTRATE` ***IOCTL*** or both. However, every time the count rate exceeds the recommended maximum, a descriptive line is entered into the control computer's log file - this is useful if (either by some failure or by tampering with the system) the PMT protection measures are not successfully applied.

The `IOCTL_GET_CUR_COUNTRATE` ***IOCTL*** can, in theory, be used by a user-space application to read photometric data from the driver. However, this would require the user-space application that does higher-level processing of the photometry and records measurements to poll the driver at least once per millisecond, which is a fairly complex matter, unless that application is allowed to consume vast amounts of system resources. A suitable alternative was implemented in the form of an array that can buffer photometry for several seconds (5 seconds is the recommended default) and a pair of pointers to elements within that array. The one pointer points to the buffer element that contains the counts of the previous millisecond and the other pointer points to a buffer element that corresponds to the start of an integration. The latter pointer needs to be set when an integration is to be started, which is done by calling the `IOCTL_ZERO_OFFSET` ***IOCTL***, which makes the driver set the pointer to the element in the buffer that corresponds to the next turn of a second - this pointer then serves as a zero-point and whenever data are read from the buffer, it is done relative to this zero-point. When a user-space application calls the `IOCTL_ZERO_OFFSET` ***IOCTL***, it must specify a pointer to a memory location to which the driver will write the local mean time corresponding to the zero-point pointer (in the form of milliseconds since 0h0m0s0ms). Since the zero-point is set in the future, there may be a delay of up to 999 ms between the zero-point being set (i.e. the user-space application is starting a new integration) and data that form part of that integration becoming available. Once data that form part of a newly-started integration are available, they may be read by the user-space application through the `IOCTL_GET_SPEC_COUNTRATE` ***IOCTL***. When calling this ***IOCTL***, the user-space application must specify a pointer to a memory location containing a variable of type

unsigned long int - which is used both by the user-space application to specify which element in the driver's buffer it requires and for the driver to store the value of the element for the user-space application's use. The desired element is specified in milli-seconds since the start of the integration - the driver will internally translate that to an element number in its buffer. In order to take repeated measurements, the user-space application may either set the zero-point for each repetition (in which case there may be delays of up to 999 ms between repetitions) or set the zero-point only at the start of the first integration and determine the start and end points of the integrations in order to bin the data correctly.

B.4 Testing

Since the photometry-and-time driver is so central to the operation of the telescope, it was subjected to particularly rigorous testing, which was done off-line - i.e. with a signal generator (instead of the PMT) connected to the photometry card.

Preliminary tests involved simply checking that the photometry-and-time driver was reading counts from the photometry hardware and that the time was being correctly read from the time service hardware. In both cases, the results seemed to indicate that both tasks were being performed correctly and no large or critical errors were present in the data, although closer inspection revealed some minor anomalies that required further investigation.

Numerous and lengthy tests were done to investigate the anomalies in photometric counts observed during the preliminary tests - which were done by comparing the input frequency of the signal generator with the count rate read by the driver. Although the measurements were (for the most part) correct, it was clearly not perfect (there seemed to be counts missing every 10 s). After consulting with the SAAO technical staff, the source of this anomaly was traced to the signal generator, although there was no way to remove the effect and no alternative means of directly proving the accuracy of the data was available. The only available alternative was to attempt to reproduce the results using an alternative to the photometry-and-time driver described here. The test was repeated using Stephen Potter's photometry-and-time RTAI interface [20] - which has been in use for several years - and the result was exactly reproduced.

An extension of these tests served to prove that the driver was responding within a negligible length of time to the 1 KHz *IRQ* produced by the time card. This was also done by comparing the input frequency of the signal generator with the counts read from the photometry card. Although it has been proven that the signal generator is not a reliable comparison for these tests, the errors in the signal are periodic and can therefore be accounted for. If there is a lag between a 1 KHz *IRQ* being emitted and the driver responding to it, it would be a result of the CPU being overtaxed, which would introduce a stochastic component in the counts read by the driver. The stochastic component would be more pronounced the more taxed the CPU is. The time taken by the driver to respond to an *IRQ* can therefore be qualitatively tested by comparing the data collected when the CPU is relatively idle with the data collected when the CPU is highly taxed and noting any stochastic effects. During

the tests, several CPU-intensive operations were carried out, but no stochastic component was observed in the data. It can therefore be concluded that the driver responds to the 1 KHz *IRQs* within a negligible length of time.

Further tests involved switching off the signal generator and setting the signal generator to a frequency higher than the recommended maximum count rate in order to check whether the driver responds correctly when the PMT is (probably) disconnected and the PMT is over-illuminated, respectively. In both cases, the driver issued the correct warning through the ***chardev*** interface.

The timing anomaly observed during the preliminary tests, after several additional tests, were confirmed and characterised. For some reason, the time read from the time card did not always increment by 1 second at the 1000^{th} milliseconds after the time had previously been incremented by 1 second. After much more testing, it became clear that there was a particular pattern to this effect: At some point, the driver counts 1005 1 KHz pulses from one turn of the second to the next, followed by exactly 5 instances where 999 1 KHz pulses are counted from one turn of the second to the next interspersed between $5 - 10$ "normal" seconds (i.e. where exactly 1000 1 KHz pulses are counted from one turn of the second until the next) - after which the cycle restarts. This cycle lasts approximately $10 - 15$ s and has been repeating without fail for several months. All efforts to track down the source of this effect within the software failed. Although the SAAO technical staff are confident in the accuracy of the SAAO time service, it would be the best place to start looking during future efforts to find the source of the effect. Fortunately, for most practical purposes this error of up to 5 ms is of no consequence. Furthermore, the driver logs each "abnormal" second (i.e. where greater or fewer than 1000 1 KHz pulses are counted from one turn of the second until the next), so that this effect can be accounted for in future applications where millisecond-level precision is critical.

B.5 Forward Look

In the near future, further tests should be done to verify the correctness of the data collected by the photometry-and-time driver using astronomical targets.

In the more distant future, a different means of reporting an over-illumination of the PMT should be considered. The current interface is fast enough to prevent damage to the PMT, although it requires that the programme responsible for taking measures to protect the PMT be run on the same computer as the photometry-and-time driver.

Appendix C

Motor driver

This appendix outlines several aspects of the motor driver that was developed to simplify the matter of controlling the telescope drives from user-space applications. A description of the electronics used to control the motors may be found in Boyd and Genet[2].

Because the telescope drives are controlled by an expansion card installed in the telescope control computer, only privileged programmes can send commands to the drive controller and receive feedback from it. Such a privileged programme would then have unlimited control over all of the control computer's hardware and software, which is obviously unwise and can even be dangerous.

It was decided to implement a separate software interface with the telescope drive electronics that is completely removed from the higher-level operations of the telescope control software. By the introduction of this additional, intermediate layer between the user-space applications requiring access to the telescope drives and the telescope drive electronics, the telescope drives may be controlled more simply by those user-space applications (to an extent that depends on the complexity of the intermediate layer) and access to the telescope drives may be controlled more finely.

The current version of the motor driver merely acts as a wrapper around the direct interface with the telescope drive electronics and so does little more complex operations than forwarding commands and feedback between telescope drive electronics and the user-space applications that need access to it. However, now that the motor driver has been developed, tested and used at the telescope with no failures for several months, it may be expanded such as to shift the burden of some of the less complex operations related to telescope drive control from the user-space applications to the motor driver.

The motor driver uses several ***Input-Output Control*** (***IOCTL***) interfaces to communicate with user-space applications. Those ***IOCTLs*** that are used send data to the telescope drive electronics are tabulated in Table C.1 and those ***IOCTLs*** that are used to receive data from the telescope drive electronics are tabulated in Table C.2.

In order to move the telescope, three parameters need to be written to the telescope drive electronics, they are (in order): rate of motion, distance to move (in motor steps) and

IOCTL Name	Meaning
IOCTL_SET_SLEW_RATE	Write new rate of motion
IOCTL_SET_DIRECTION	Write desired direction of motion
IOCTL_SET_STEPS	Write distance (in motor steps) by which to move
IOCTL_SET_SID_RATE	Write sidereal rate (rate of motion while telescope is tracking)

Table C.1: Summary of motor driver ***IOCTLs*** used for sending data to the telescope drive electronics. In each case, the user-space application must provide a pointer to a variable of type **unsigned long int** , which has the desired value for the corresponding parameter.

Ioctl Name	Meaning
IOCTL_GET_STEPS	0 if telescope is stationary, otherwise an integer indicating the distance (in motor steps) remaining before the specified target is reached.
IOCTL_GET_LIMITS	0 if none of the telescope's electronic limits (see §2.2.1) have been reached, otherwise an integer where the binary representation indicates which electronic limit has been reach (see Table C.3).

Table C.2: Summary of motor driver ***IOCTLs*** used for reading data from the telescope drive electronics. In each case, the user-space application must provide a pointer to a variable of type **unsigned long int** to which the data can be written.

the direction in which to move. Once this set of three parameters is correctly specified, the motion will commence.

The direction ***IOCTL*** is also used to activate telescope tracking and to de-energise the telescope drives. This is in fact how the telescope drive electronics receives these commands, i.e. the integer passed to the telescope drive electronics that specifies the direction of motion is interpreted in its binary form and has extra used bits that indicate whether the telescope should be tracking and whether the telescope drives should be energised. Table C.3 gives a breakdown of the meaning of the various bits in this parameter.

Bit	Symbolic Name	Meaning
1	DIR_WEST_MASK	Move telescope West or Western electronic limit reached
2	DIR_EAST_MASK	Move telescope East or Eastern electronic limit reached
3	DIR_NORTH_MASK	Move telescope North or Northern electronic limit reached
4	DIR_SOUTH_MASK	Move telescope South or Southern electronic limit reached
5	TRK_OFF_MASK	Disable telescope tracking (active by default)
6	POWER_OFF_MASK	De-energise telescope drives (energised by default)

Table C.3: Summary of motor driver direction parameter bits.

Appendix D

PLC interface

This appendix outlines the low-level software interface with the PLC. For a higher-level description of the software interfacing with the Programmable Logic Converter (PLC), refer to chapter 8. For a more in-depth description of the PLC, refer to Fourie[5; 6].

The PLC connects to the control computer via a standard RS-232 ("serial") connector and manages the following pieces of hardware on behalf of the control computer:

- Dome shutter
- Dome dropout
- Dome rotation (both manual and automatic)
- Telescope pointing encoders
- Telescope focus
- Hand paddle (to move telescope in cardinal directions and adjust the telescope focus manually - status only)
- Dark slide
- Acquisition mirror
- Filter
- Aperture
- PMT high-voltage power-supply

D.1 Rationale of User-space Implementation

The software interface with the PLC was implemented in the form of a user-space application (as opposed to a Linux kernel driver) for the simple reason that there was no need to develop

it as a kernel driver. The other software interfaces were implemented as kernel drivers either so user-space applications could access the interface simply and securely or to guarantee accurate timing within the interface.

The PLC connects to the serial port of the computer, so unprivileged (user-space) applications have full access to it by default, so no wrapper driver is necessary (as with the motor driver).

Input and output over a serial connection is inherently slow and hardware that use serial connections are usually designed to allow long delays in the data transmission process. The software interface with the PLC could therefore be designed without any considerations for accurate timing and synchronous data transmission (unlike the photometry-and-time driver and the CCD driver).

If the software interface with the PLC were implemented as a kernel driver, it would necessarily be subject to a privileged scheduling priority (in terms of CPU time as well as other system resources). As such, there would have been an increased likelihood that the software interface with the PLC could interfere with some of the timing-critical interfaces had the PLC interface been implemented as a kernel driver.

D.2 Reading Status and Sending Commands

Both the hardware status read from the PLC and the commands sent to the PLC are transmitted in the form of an array of C **chars** exactly like a text string - hence the terms "status string" and "control string", respectively. Furthermore, on UNIX-like systems (such as all flavours of Linux) the computer's serial port is represented by a pseudo-file which can be worked with like any normal binary or text file. Therefore, in terms of writing the software interface, accessing the status is identical to reading a line of text from a file on the hard-disk and sending a command is identical to writing a line of text to a file on the hard-disk.

However, where text strings normally contain human-readable text, the control strings and status strings (mostly) only have significance when interpreted in their binary forms. In this case, each **char** can contain 4 bits of information, where each bit of information is either true or false. A C **char** consists of 8 binary digits, but only the lower 4 digits are used. A good example is the dome control character in the control string (see Table D.1).

Some parts of the control and status strings are not interpreted in their binary forms, instead they contain decimal numbers formatted as human-readable text strings. This representation is only used in cases where some number needs to be communicated (such as the desired/current dome azimuth, filter position, aperture position and pointing encoder readouts). A breakdown of the control and status strings are given in Tables D.2 and D.3.

Binary Digit	4	3	2	1
Binary Value	16	8	4	2
Meaning	Load Dome Offset	Dome Manual Move	Dome Direction	Dome Auto Move

Table D.1: Example of PLC control/status character. If bit 1 is true, the PLC should rotate the dome to the ideal azimuth (specified elsewhere in the control string) and the PLC will only do so if bit 1 of this character in the control string is true. Bits 2 and 3 work in tandem - if bit 3 is true the PLC will move the dome until bit 3 is set to false and if 2 is true the dome will be rotated left, otherwise the dome will be rotated right. If bit 4 is true, the PLC will read the new offset of the dome azimuth zero point from the dome fiducial (specified elsewhere in the control string).

Character(s)	**Bit**	**Interpretation if value is true**
0-6	(text)	Status header `@00RD00`
7-10	(decimal)	Current dome position in decidegrees
12	1	Dome dropout is open
	2	Dome dropout is closed
	3	Dome dropout is moving
13	1	Dome shutter is open
	2	Dome shutter is closed
	3	Dome shutter is moving
14	1	Trapdoor to dome is open
	2	Dome is moving
15	1	Aperture wheel is at the initialisation position
	2	Aperture wheel is at a slot
	3	Aperture wheel is moving
16	1	Filter wheel is at the initialisation position
	2	Filter wheel is at a slot
	3	Filter wheel is moving
17	1	Acquisition mirror is in-beam
	2	Acquisition mirror is out-of-beam
	3	Acquisition mirror is moving
18	1	Instrument shutter is open
19	(decimal)	Current focus region: `0` is in-region, `8` is out-region
20-22	(decimal)	Current focus position in region specified by character 19
25	1	Focus is moving
	2	Focus at initialisation position
	3	Focus motor has stalled
26	1	Focus at slot position
	2	Focus at reference position
	3	Focus in-region sensor is active
	4	Focus out-region sensor is active
27	3	Mains power failure
	4	PLC watchdog has tripped (probably computer has stopped responding)

Table D.2: Breakdown of PLC status string. Unspecified bits and characters are zero. `(text)` means that part of the string is interpreted as human-readable text. `(decimal)` means that part of the string is a human-readable decimal number. `hex` means that part of the string is a human-readable hexadecimal number.

Table D.2 cont.

28	1 2 3	High-voltage power source is manually switched off High-voltage power source 1 is on High-voltage power source 2 is on
29	1 2 3 4	Handset South button pressed Handset North button pressed Handset East button pressed Handset West button pressed
30	1 2 3 4	Handset Focus In button pressed Handset Focus Out button pressed Handset Slew speed selected* Handset Guide speed selected*
32	(decimal)	Current aperture position
34	(decimal)	Current filter position
35-38	(decimal)	Current dome offset in decidegrees
39-40	(decimal)	Current dome maximum flop in decidegrees
31-42	(decimal)	Current dome minimum flop in decidegrees
43-46	(decimal)	Current RA encoder low word †
47-50	(decimal)	Current RA encoder high word †
51-54	(decimal)	Current Dec encoder low word †
55-58	(decimal)	Current Dec encoder high word †
75-76	(hex)	Frame check sequence - calculate checksum of status string and compare with this number to verify accurate data reception
77-78	(text)	Status footer `*\r` (where `\r` is the carriage return character)

The control computer can request the current status string by sending the command `@00RD0150001754*\r` to the PLC. The PLC should (practically instantaneously) start transmitting the status string.

The control computer can send a command to the PLC at any time by sending the relevant control string. After the control string is received by the PLC, it should respond with the string `@00WD0053*\r`, which indicates that the control string was successfully received. If the number represented by the 6^{th} and 7^{th} characters is non-zero, it means the PLC received an invalid control string or some other error occurred. Some commands in the control string only need to be specified once, some should be sustained until the desired outcome is reached, some should be sustained for as long as the result is desired and some should be specified depending whether or not other options are specified. For this reason, it is important to not only send appropriate commands to the PLC but also keep track of the desired outcome(s) a particular command was intended to effect.

Each string sent by the PLC or sent to the PLC must contain a frame check sequence (FCS). The FCS is the 2-digit hexadecimal number immediately before the terminating `*\r` sequence in each string, which serves as a checksum to ensure accurate data transmission.

*If both Slew and Guide bits are false, then Set speed is selected.

†Both the RA and Dec encoder readouts are read by concatenating the low word string and high word string and interpreting it as a decimal number.

‡If `Manual dome move` is true and `Rotate dome left` is false, dome will move right.

Character(s)	Bit	Interpretation if value is true
0-8	(text)	Control header `@00WD0100`
9-12	(decimal)	Required dome azimuth for autoguiding (in decidegrees)
13-14	(decimal)	Dome azimuth maximum flop (in decidegrees)
15-16	(decimal)	Dome azimuth minimum flop (in decidegrees)
17	1 3 4	Open instrument shutter Reset acquisition CCD controller Reset watchdog timer
18	1 2 3 4	Open dome dropout Close dome dropout Zero RA encoder Zero Dec encoder
19	1 2	Open dome shutter Close dome shutter
20	1 2 3 4	Dome auto-guide Rotate dome left (`Manual dome move` must be true)‡ Manual dome move ‡ Set dome offset (specified by characters 33-36)
21	1 2 3 4	Move focus to position specified by characters 29-32 Move focus out (away from primary mirror) Move focus in (toward primary mirror) Reset focus (move to next slot position and stop)
22	1 2 3 4	Move aperture to initialisation position (slot 0) Move aperture to position specified by character 26 Reset aperture (move to next slot position and stop) Move focus to initialisation position (slot 0)
23	1 2 3	Move filter to initialisation position (slot 0) Move filter to position specified by character 26 Reset filter (move to next slot position and stop)
24	1 2 3 4	Activate high-voltage power source 1 Activate high-voltage power source 2 Halt acquisition mirror (default: false) Set acquisition mirror position - true is in-beam, false is out-of-beam
26	(decimal)	Requested filter position (only applicable if character 23, bit 2 is true)
28	(decimal)	Requested aperture position (only applicable if character 22, bit 2 is true)
29 30-32	(decimal) (decimal)	Region of requested focus position: `0` is the in-region, `8` is the out-region Requested focus position (region specified by character 29)
33-36	(decimal)	New dome offset in decidegrees (only applicable if character 20 bit 4 is true)
53-54	(hex)	Frame check sequence - checksum is calculated from control string so PLC can verify accurate data reception
55-56	(text)	Control footer `*\r` (`\r` is the carriage return character)

Table D.3: Breakdown of PLC control string. Unspecified bits and characters are zero. `(text)` means that part of the string is interpreted as human-readable text. `(decimal)` means that part of the string is a human-readable decimal number. `(hex)` means that part of the string is a human-readable hexadecimal number.

The checksum is the result of an *exclusive-or* (*XOR*) operation performed an all data to be transmitted that appears before the FCS in the string.

D.3 Parsing Status Strings

For the sake of convenience, a single function was designed and implemented that would request the current status string, receive the reply, extract all the information and store the information in a C **struct** for easy access. The **struct** designed for this purpose is depicted in Table D.4.

Using the data contained within this structure, the current status of all hardware managed by the PLC can be easily accessed from any function that requires it.

D.4 Constructing the Control String

As with the status string, a function was written to extract the desired outcomes from a C **struct** and use those data to construct a control string. The **struct** designed for this purpose is depicted in Table D.5.

As some commands need to be sustained for a length of time and others only need to be specified once, a function was written that would create and initialise the PLC control **struct** . The initial values of all the fields are derived from the current status of that piece of hardware and whether or not that command needs to be issued only once or sustained. As an example, the command to open the instrument shutter needs to be sustained until the instrument shutter should be closed - this function will therefore initialise the instrument shutter field of the control **struct** to true if the instrument shutter is currently open (according to the status **struct**) and false otherwise so that the user or telescope automation routine may open and close the instrument shutter as necessary. On the other hand, the command to move the filter wheel only needs to be specified once, so the corresponding fields of the control **struct** can be initialised to 0.

Once the control **struct** is initialised, it may be modified to effect a particular result and sent to the PLC.

C Type	Field	Purpose
unsigned short	dome_pos	Current dome azimuth in decidegrees
unsigned char	dome_dropout	Dome dropout (1 Open, 2 Closed, 0 otherwise)
unsigned char	dome_shutter	Dome shutter (1 Open, 2 Closed, 0 otherwise)
unsigned char	dome_moving	True if the dome is rotating, false otherwise
unsigned char	trapdoor_open	True if trapdoor to dome is open, false otherwise
unsigned char	aperture_stat	Aperture wheel (bitwise interpretation: bit 1 Initialisation position, bit 2 Slot position, bit 3 Aperture wheel turning)
unsigned char	filter_stat	Filter wheel (bitwise interpretation: bit 1 Initialisation position, bit 2 Slot position, bit 3 Filter wheel turning)
unsigned char	acq_mir	Acquisition mirror (bitwise interpretation: bit 1 Mirror is in-beam, bit 2 Mirror is out-of-beam, bit 3 Mirror is moving)
unsigned char	pmt_shutter	Instrument shutter (true if open, false otherwise)
short	focus_pos	Current focus position
unsigned char	focus_stat	Focus status (bitwise interpretation: bit 1 Focus at slot, bit 2 Focus at reference position, bit 3 Focus in out-region, bit 4 focus in in-region, bit 5 Focus moving, bit 6 Focus initialised, bit 7 Focus stalled)
unsigned char	power_fail	True if mains power has failed, false otherwise
unsigned char	watchdog_tripped	True if PLC watchdog has been tripped, false otherwise
unsigned char	eht	PMT high-voltage power source (bitwise interpretation: bit 1 EHT manual off, bit 2 EHT set to low, bit 3 EHT set to high)
unsigned char	handset	Status of the hand paddle representing depressed buttons (bitwise interpretation: bit 1 focus in, bit 2 focus out, bit 3 slew speed, bit 4 guide speed, bit 5 South, bit 6 North, bit 7 East, bit 8 West)
unsigned char	aperture_num	Aperture number
unsigned char	filter_num	Filter number
unsigned short	dome_offset	Dome offset in decidegrees
unsigned short	dome_min_flop	Dome coasting angle in decidegrees
unsigned short	dome_max_flop	Maximum dome flop in decidegrees
long	ha_pulses	Hour angle encoder pulses
long	dec_pulses	Declination encoder pulses

Table D.4: PLC status structure.

C Type	Field	Purpose
unsigned short	dome_pos	Required dome azimuth in decidegrees
unsigned short	dome_max_flop	Required maximum flop in decidegrees
unsigned short	dome_min_flop	Required minimum flop in decidegrees
unsigned char	acq_reset	Reset acquisition CCD controller (true or false)
unsigned char	pmt_shutter	Open instrument shutter (true or false)
unsigned char	tel_lim	Zero RA and/or Dec encoder (bitwise interpretation: bit 4 Zero Dec encoder, bit 8 Zero RA encoder
unsigned char	dome_dropout	Open dome dropout (true or false)
unsigned char	dome_dropout	Open dome shutter (true or false)
unsigned char	dome_cmd	Dome rotation command (bitwise interpretation: bit 1 Dome auto-guide, bit 2 Move dome left (if `Move dome` bit true), bit 3 Move dome, bit 4 Load dome encoder offset.
unsigned short	dome_offset	Required dome offset in decidegrees
unsigned char	focus_cmd	Focus move command (bitwise interpretation: bit 1 Move focus to requested position, bit 2 Move focus in, bit 3 Move focus out, bit 4 Reset focus)
unsigned short	focus_pos	Required focus position (bit 1 of `focus_cmd` must be set)
unsigned char	aperture_cmd	Aperture wheel command (bitwise interpretation: bit 1 Initialise aperture wheel, bit 2 Move to requested aperture, bit 3 Reset aperture wheel)
unsigned char	aperture_num	Required aperture number
unsigned char	filter_cmd	Filter wheel command (bitwise interpretation: bit 1 Initialise filter wheel, bit 2 Move to requested filter, bit 3 Reset filter wheel)
unsigned char	filter_num	Required filter number
unsigned char	acq_mir	Acquisition mirror command (bitwise interpretation: bit 3 Halt acquisition mirror, bit 4 Move acquisition mirror in-beam)
unsigned char	eht	PMT high-voltage power source (bitwise interpretation: bit 1 EHT low, bit 2 EHT high - if neither bit is set, EHT is switched off)

Table D.5: PLC control structure.

Appendix E

Dictionary of Technical Terms

character device, chardev - a communication mechanism that enables the exchange of data between a Linux kernel driver and some user-space or kernel-space process running on the same computer. Data is sent to the driver by writing text or binary data to a pseudo-file (usually under /dev on most Linux distributions) as if writing to a file on disk.

Input-Output Control, IOCTL - a communication mechanism that enables the exchange of data between a Linux kernel driver and some user-space or kernel-space process running on the same computer. A kernel driver may have several IOCTLs, each related to a particular read- and/or write-operation that a programme may perform with the driver.

shared memory block - an interprocess communication mechanism by which two processes can exchange data using a block of memory that is accessible by both processes. The type of data that can be exchanged depends on the data type of the memory block.

kernel message queue - an interprocess communication mechanism by which data is exchanged between multiple processes using message structures containing binary data that is managed by the Linux kernel. Message structures can be given an identifier number when they are uploaded to the kernel message queue and, when retrieving messages from the queue, only those messages with an identifier matching that requested will be returned to the programme.

pipe, first-in-first-out (FIFO) pipe - a pseudo-file intended for temporary storage of data that are sent from one programme to another. The FIFO has a read end and a write end, thus defining the direction of flow of data. The FIFO may be accessed from both programmes exactly as with binary or text files on disk.

signals, SIGCHLD - a means of interrupting the normal execution of a process in order for that programme to respond to some event occurring. SIGCHLD is the particular signal a process receives when a process it spawned (using the *fork* function) exits.

network socket, UNIX socket - the implementation of network communication on Linux and UNIX-like operating systems.

tunnel - encapsulating one communications protocol within another to enable features not supported by the carrier protocol.

X Window System, X11, X - a set of applications and communications protocol specifications that enable graphical user interfaces on Linux and UNIX-like systems. X employs a client-server architecture, where the client is a programme that has a graphical user interface and the server is the X server programme. The X server programme manages input and output hardware connected to the computer and allows its clients to display output on output devices (such as a monitor) and receive feedback from input devices (such as a mouse and keyboard) in a generalised manner. The X communication protocol is network-transpartent, meaning that the client and the X server can be run on separate computers (as long as the computers are connected through a network).

XLib - a library containing functions for communicating with the X server. All forms of interactions with the user (such as displaying items on a monitor or receiving feedback from a mouse or keyboard) makes use of XLib. Graphical applications rarely use XLib directly, but rather use libraries of convenience functions, such as GTK+ or Qt.

XLib socket, XLib plug, XEmbed - XLib plugs and sockets are used to embed the output from one graphical programme (the embedded programme) within another (the embedding programme) and XEmbed is the communication protocol used to connect a plug with a socket. Similarly all feedback from the user that occurs within the section of the embedding programme's output that corresponds to the embedded programme are forwarded to the embedded programme. In this way, a feature provided by a particular programme seems to the user to be enabled in another programme, which lacks this feature. For instance, XEmbed is in popular use by web browsers to display multimedia content (such as videos and animations) as part of a webpage.

Bibliography

[1] D. P. Bovet and M. Cesati. Understanding the Linux Kernel - Process Scheduling. O'Reilly, `http://oreilly.com/catalog/linuxkernel/chapter/ch10.html`, 2000. Retrieved 15 July 2011.

[2] Louis J. Boyd and Russell M. Genet. *Telescope Control Board - Technical Manual.* Fairborn Observatory and AutoScope Corporation, 1988.

[3] D. Carter. *The SAAO Automatic Photoelectric Telescope - Technical Manual.* South African Astronomical Observatory, Cape Town, 1999.

[4] Gregory Sean Cox. *Designing Hypothesis Tests for Digital Image Matching.* PhD thesis, University of Cape Town, August 2000.

[5] P. Fourie. *ACS-APT PLC Command Structure.* South African Astronomical Observatory, Cape Town, 2007.

[6] P. Fourie. *Communication with the OMRON CJ1M PLC.* South African Astronomical Observatory, Cape Town, 2007.

[7] L. Fu and R. Schwebel. RT PREEMPT HOWTO. `https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO`, 2010. Retrieved 15 July 2011.

[8] G. W. Henry. ATIS Dispatch Scheduling of Robotic Telescopes. In T. Boroson, J. Davies, & I. Robson, editor, *New Observing Modes for the Next Century*, volume 87 of *Astronomical Society of the Pacific Conference Series*, pages 145–+, 1996.

[9] F.V. Hessman. Robotic Telescope Projects. `http://www.astro.physik.uni-goettingen.de/~hessman/MONET/links.html`, September 2010. Retrieved 15 July 2011.

[10] F.V. Hessman. Robotic Telescopes of the World. `http://www.astro.physik.uni-goettingen.de/~hessman/MONET/map.gif`, September 2010. Retrieved 15 July 2011.

[11] D. Kilkenny, P. Martinez, and D. Carter. *The SAAO Automatic Photometric Telescope - User Manual.* South African Astronomical Observatory, Cape Town, 2010.

[12] P. Kubánek, M. Jelínek, S. Vítek, A. de Ugarte Postigo, M. Nekola, and J. French. RTS2: a powerful robotic observatory manager. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 6274 of *Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*, July 2006.

[13] P. Mantegazza. DIAPM RTAI for Linux: WHYs, WHATs and HOWs. `https://www.rtai.org/index.php?module=documents&JAS_DocumentManager_op=downloadFile&JAS_File_id=31`, 1999. Retrieved 15 July 2011.

[14] P. Martinez, D. M. Kilkenny, G. Cox, D. B. Carter, D. T. Ellis, G. P. Evans, M. Fouche, W. P. Koorts, D. Metcalfe, W. Pearson, A. Riddick, E. Sommeregger, J. Stoffels, D. Weir, and G. Woodhouse. The automatic photometric telescope at the South African Astronomical Observatory. *Monthly Notes of the Astronomical Society of South Africa*, 61:102–111, 2002.

[15] J. Medkeff. The ASCOM Revolution. *Sky & Telescope*, 99(5):66–68, May 2000.

[16] J. Meeus. *Astronomical formulae for calculators.* Willmann-Bell, 1982.

[17] J. Meeus. *Astronomical algorithms (2nd ed.).* Willmann-Bell, 1998.

[18] J. Neider, T. Davis, M. Woo, and OpenGL Architecture Review Board. *OpenGL programming guide: the official guide to learning OpenGL, release 1.* Addison-Wesley, 1993.

[19] C. Pennypacker, M. Boer, R. Denny, F. V. Hessman, J. Aymon, N. Duric, S. Gordon, D. Barnaby, G. Spear, and V. Hoette. RTML - a standard for use of remote telescopes. Enabling ubiquitous use of remote telescopes. A&A, 395:727–731, November 2002.

[20] Stephen B. Potter. SAAO polarimeter Reference manual. `http://www.saao.ac.za/~sbp/HIPPO/reference.html`, 2011. Retrieved 22 July 2011.

[21] S. Regis. Introduction to NX Technology. `http://www.nomachine.com/documents/pdf/intr-technology.pdf`, 2009. Retrieved 15 July 2011.

[22] David A. Rusling. The Linux Kernel - Interprocess Communication Mechanisms. `http://tldp.org/LDP/tlk/ipc/ipc.html`, 1999. Retrieved 15 July 2011.

[23] Steadly, R. Scott and Robinson, Michael S., editors. *The Astronomical Almanac (2011).* United States Naval Observatory, The United Kingdom Hydrographic Office, 2010.

[24] Fiona Vincent. Positional astronomy: Conversion between horizontal and equatorial systems. `http://star-www.st-and.ac.uk/~fv/webnotes/chapter7.htm`, November 2003. Retrieved 15 July 2011.