# THE IRMA III CONTROL AND COMMUNICATION SYSTEM

**IAN SEAN SCHOFIELD**

**B. Sc. Computer Science, University of Lethbridge, 2000**

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfilment of the
Requirements of the Degree

**MASTER OF SCIENCE**

Department of Physics
LETHBRIDGE, ALBERTA, CANADA

**THE IRMA III CONTROL AND COMMUNICATION SYSTEM**

**IAN SEAN SCHOFIELD**

Approved:

---

Dr. David A. Naylor, Supervisor, Department of Physics — Date

---

Dr. Shahadat Hossain, Department of Computer Science — Date

---

Dr. Adriana Predoi-Cross, Department of Physics — Date

---

Dr. James Di Francesco, External Examiner — Date

---

Dr. Brian Dobing, Examination Committee Chair — Date

## Abstract

The IRMA III infrared radiometer is a passive atmospheric water vapor detector designed for use with interferometric submillimeter arrays as a method of phase correction. The IRMA III instrument employs a distributed, multi-tasking software control system permitting precise fine-grained control at remote locations over a low-bandwidth network connection. IRMA's software is divided among three processors tasked with performing three primary functions: command interpretation, data collection and motor control of IRMA's Alt-Az mount. IRMA's hardware control and communication functionality is based on compact, low cost, energy efficient Rabbit 2000 microcontroller modules, selected to meet IRMA's limited space and power requirements. IRMA accepts scripts defined in a custom, high level control language as its method of control, which the operator can write or dynamically generated by a separate GUI front-end program.

## Acknowledgements

This thesis builds upon the work of many people, starting with Dr. David Naylor and Graeme Smith, who envisioned, built and tested the initial IRMA prototype. Much of the theoretical base of this thesis stems from Graeme's work, to whom I am very grateful. Thank you David for allowing me to be involved in the IRMA project from its infancy up to the present, as IRMA is on verge of becomming a commercial product. The additional following people have provided support in the development of the IRMA III software system:

Dr. Robin Phillips: for his efforts in reviewing my initial draft of this thesis, as well as spearheading the effort to get IRMA built. Additional thanks for advice to problems relating to Perl and Linux.

Greg Tompkins: for his help in building the hardware IRMA's software is dependent upon. Additional thanks for your electronics related support – from building cabling to helping diagnose problems. Greg's insights on good nutrition have been especially helpful.

Brad Gom: for his efforts in designing and building the initial IRMA III hardware.

Jacob Ellegood and Dan Clossen: for their work laying out the printed circuit boards (PCBs) for IRMA III and Alt-Az. It should also be mentioned that Jacob wrote a portion of the user interface code for IRMA II.

Amy Smith: for writing a graphical user interface for IRMA that works seamlessly with the IRMA command processor.

Dr. Arvid Schultz: for his aid testing the Alt-Az to RA-DEC conversion routines, and his regular visits to the lab.

Frank Klassen: for building custom components and precision machine work for IRMA.

Dr. Gary Davis, director of the Joint Astronomy Centre, Hawaii, for allowing site testing of IRMA at the James Clerk Maxell Telescope (JCMT).

Special thanks must be given to my wife, Terilynn, whose support over the past two years has been extraordinary.

The IRMA III project has received support from the Natural Sciences and Engineering Research Council of Canada (NSERC), National Reserch Council Canada (NRC), the Alberta Science and Research Authority (ASRA).

# Contents

# List of Tables

# List of Figures

# Chapter 1

# The IRMA Concept

## 1.1 An Infrared Radiometer for Millimeter Astronomy

IRMA is an infrared radiometer designed to measure passively 20 micron water vapor rotational absorption lines, which indicate the amount of precipitable water vapor (PWV) in the atmosphere. IRMA has two primary applications: as a solution for phase correction in submillimeter interferometry, and as an sky opacity monitor for use in infrared astronomy.

### 1.1.1 IRMA as a Method of Phase Correction

Long wavelength electromagnetic radiation emitted by celestial objects remains nearly untouched as it travels through space on its journey to the Earth. Only in its final moments, as it passes through the lower regions of the Earth's atmosphere, is the radiation significantly degraded. At submillimeter wavelengths, the principal source of opacity is due to atmospheric water vapor. High-energy, short-wavelength radiation such as gamma rays,

X-rays and ultraviolet light are effectively blocked out, along with significant portions of the infrared and submillimeter wavelengths. Only visible light passes through the atmosphere relatively unhindered. The submillimeter spectral window, a band of wavelengths occupying the region between infrared light and microwaves (0.1 mm to 1 mm), contains regions (or windows) that are only partially filtered out by the presence of water vapor in the Earth's atmosphere. Submillimeter astronomy aims to exploit these transparent and semi-transparent windows that appear at submillimeter wavelengths.

The submillimeter band is of interest to astronomers for two reasons: the relatively long wavelength of submillimeter radiation allows it to penetrate gas and dust, permitting observations to be made of objects inside nebulae such as the Orion nebula, which are believed to be stellar nurseries where stars are born. Second, observations at submillimeter wavelengths can be used to observe distant objects whose light has been red-shifted (or stretched in wavelength) into the submillimeter band, permitting astronomers to view objects that appeared in the earliest epoch of the universe. The wavelength lengthening of light from distant objects is a consequence of the fact, first observed by Edwin Hubble in 1929, that distant objects are receding from the earth at increasing rates, now understood as the expansion of the universe.

The only way to observe objects in the submillimeter spectral window is to get above the bulk of the Earth's atmosphere responsible for rendering these bands opaque. This can be accomplished by placing observatories in orbit, such as the Hubble Space Telescope (HST), in an aircraft, such as NASA's SOFIA (Stratospheric Observatory For Infrared Astronomy), or at high altitude ground locations, such as at Mauna Kea, Hawaii

(4200 m), or the Atacama Desert, Chile (5000 m). Spaceborne observatories enjoy the advantage of being able to observe at all wavelengths, but are limited to mirror diameters no larger than approximately 3.5 m, the maximum diameter of payload that can be carried on board a rocket. Furthermore, at roughly 22,000 dollars per kilogram[17], the cost of launching a large payload into space is very expensive. Given the advances in astronomical technology, ground-based observatories are an attractive alternative, that can approach the performance of its space-based counterparts.

Figure 1.1: IRMA at Gemini South Observatory, September 2004.

Submillimeter ground-based observatories can be configured as interferometric arrays in order to synthesize a massive receiving antenna whose diameter equals the length of the maximum baseline of the array. The maximum baseline is the distance between the two farthest-separated antennas in the array.

The minimum spatial resolving power of a telescope is found in any standard optics text. For a telescope of circular aperture, the diffraction limit, expressed in radians, is:

$$\theta_{min} = \frac{1.22\lambda}{d} \tag{1.1}$$

where $\lambda$ is the wavelength being observed and $d$ is the diameter of the telescope[4].

Increasing the length of the baseline effectively increases the diameter of the antenna, which increases the array's spatial resolution; the minimum angle separating two objects that can be individually resolved.

The Atacama Large Millimeter Array (ALMA) project, an interferometric submillimeter telescope array consisting of 64 antennas, each 12 m in diameter, will allow reconfigurable baselines ranging from 150 m to 18 km. ALMA promises to resolve objects at 10 milliarcsecond resolution; ten times better than the Hubble Space Telescope[41]. Situated on a 5000 m high plateau in the Chilean Andes, the ALMA site is one of the driest regions on Earth. Atmospheric water vapor exists in low enough quantities to make submillimeter wavelength observation feasible, although not low enough to have negligible effect on the incoming celestial signal. In order for an interferometric array to achieve its maximum spatial resolution (approaching its diffraction limit), the distorting effects of the Earth's atmosphere must be overcome.

Water vapor found in the Earth's troposphere (0 - 14 km) is present in sufficient amounts to slow down the incoming wavefront of the celestial signal. The water vapor, measured in millimeters of precipitable water vapor (PWV), contributes a delay factor of 6 to the optical path[29]. The distribution of water vapor is neither spatially nor temporally

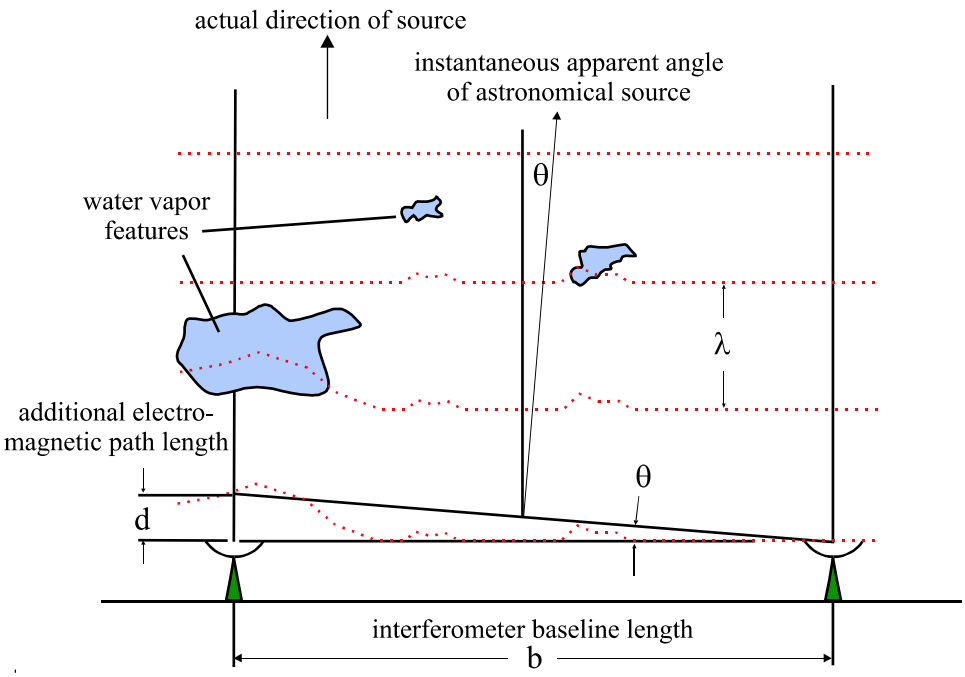


Figure 1.2: Atmospheric phase distortion of celestial signal.[57].

homogeneous inside the column of atmosphere projected from the antenna's receiving dish. Thus, it is probable that each receiving antenna will be subject to a different amount of instantaneous PWV. Since the presence of PWV slows down the incoming signal, each receiving antenna detects the wavefront at different times, rather than simultaneously, as desired.

The effect of atmospheric phase distortion is illustrated in figure 1.2, which shows an interferometric array with two antennas. The antenna pair observes the same object, whose wavefront appears planar in the upper atmosphere. The wavefront above the left hand antenna passes through a region of water vapor, which adds excess optical path length ($d$) to the incoming signal. Interferometry requires the precise measurement of the time the

wavefront was received at each antenna. The apparent direction of the observed object is perpendicular to the planar wavefront. A slight phase error manifests itself as a slight change in the immediate apparent angle of the astronomical source's direction, diminishing the interferometer's ability to spatially resolve astronomical objects.

Spectral emission measurements above Mauna Kea using high resolution Fourier transform spectroscopy show that virtually all of the atmospheric opacity in the 20 micron near-infrared band is caused by the rotational transition of water vapor molecules[41]. Water vapor molecules, which rotate at quantized rates, change their rotational rate absorbing or emitting photons. IRMA observes a number of transitions due soley to water vapor in the 20 micron ( 15 THz) band No other atmospheric molecule exhibits transitions at this wavelength, making it an ideal indicator of water vapor content.

By using a single bandpass filter with a cutoff of 20.5 microns, the 20 micron band can be isolated and measured by a detector, thus determining the column abundance of PWV in the antenna's line-of-sight, and ultimately the amount of excess optical path length at submillimeter wavelengths. By continually measuring PWV levels above each antenna in the interferometric array, and subtracting the amount of excess path length from the antenna's data (sampled at synchronized intervals), the phase error contained in the antenna's data set can be compensated, thus enabling the interferometric array to operate at its full potential. This is the basic operational theory behind the IRMA water vapor detector.

### 1.1.2 IRMA as an Opacity Detector

The infrared spectral window appears in the region between visible light (700 nm) and the submillimeter (100 microns). The infrared window has varying degrees of opaqueness depending upon the amount of atmospheric water vapor content. When used in conjunction with an infrared telescope, IRMA can serve as an effective monitor of atmospheric water vapor abundance.

## 1.2 History of IRMA

IRMA was originally envisioned as an alternative solution to the problem of phase correction in submillimeter interferometry. One of several solutions to phase correction involves measuring the strength of the water vapor molecule's transitions at 183 GHz. The strength of the 183 GHz signal is proportional to the column abundance of water vapor above the receiver antenna. This system, however, requires the use of a high-frequency heterodyne receiver, which besides being costly and complex, is an emitter of RF noise in the telescope receiver cabin.

### 1.2.1 IRMA I

Proof of concept tests were performed in December, 1999 at Mauna Kea, Hawaii using a prototype IRMA device, IRMA I[57]. The first generation IRMA consisted of a wet cryostat containing the infrared detector, a tipper mirror driven by a stepper motor, and an off-axis parabolic mirror. The tipper mirror allowed 180 degrees rotation about the elevation (or altitude) axis, permitting the operator to perform skydips between the

horizon and zenith, as well as point to nadir, where the calibration target (a cold bucket, filled with liquid nitrogen) was located. Control and data collection were performed by laptop computer running a MS-DOS based control application. The cryostat and cold bucket required a liquid nitrogen refill roughly every 4 hours.

When results from the IRMA I experiments showed that the IRMA accurately tracked the 183 GHz data, work began on a second generation IRMA, which would feature improved hardware and software. Hardware improvements included new filters that had a better spectral match to the band of interest, a more sensitive IR detector with lower signal to noise, and an improved ADC with higher dynamic range[6]. The basic mechanical design, however, remained the same, including the need for frequent liquid nitrogen refills. The original MS-DOS control software was rewritten for the GNU/Linux operating system by this author, and was designed as a common gateway interface (CGI) application, allowing the operator to control the instrument over the WWW using a web browser. IRMA II was the University of Lethbridge Astronomical Instrumentation Group's (AIG) first networked instrument; one in the line of many that followed.

### 1.2.2 IRMA II

IRMA II operated from December 2000 to March 2001, collecting PWV abundance data. The goal of IRMA II was to compare atmospheric transmission measurements performed with IRMA with measurements performed by existing water vapor detection systems, namely the James Clerk Maxwell Telescope (JCMT) SCUBA bolometer camera, the Caltech Submillimeter Observatory (CSO) 225 GHz and 350 micron radiometers, and the JCMT 183 GHz water vapor meter radiometer[37].

Tau ($\tau$), or optical depth, is a measure of atmospheric transmission at some spectral band of interest. Conversely, $\tau$ can be described as the fraction of radiation absorbed per unit traveled, which is the definition to opacity. Opacity is an indicator of atmospheric water vapor content as both are directly related; an increase in opacity (or lower transmission) is a result of an increase in atmospheric water vapor.

Analysis comparing SCUBA and IRMA atmospheric transmission (or $\tau$) values showed strong correlation at the 850 and 450 micron bands[5]. Comparisons with the CSO Tau opacity monitors showed a similar, although slightly weaker correlation (particularly with the 350 micron data). The positive results from IRMA II showed that IRMA was a reliable means of measuring PWV. The data collected by IRMA II contributed to the development of the ULTRAM radiative transfer model[6].

### 1.2.3 IRMA III

In the summer of 2001, work began on a third generation IRMA unit, which promised substantial improvements: an autonomous, steerable water vapor radiometer that did not require liquid cryogen refilling. IRMA III would be remotely controllable by an operator over a network link, and be capable of pointing to an altitude-azimuth (Alt-Az) coordinate in the sky. Finally, IRMA III would use a custom command control language, allowing the operator maximum flexibility of control over the instrument.

It was hoped IRMA III could be deployed at the ALMA site in Chile. The demands of operating at a remote site without electrical power or a persistent, high-bandwidth network connection made it necessary that IRMA be a self-contained, compact unit that consumed little power. These restrictions led to the adoption of the Rabbit Semiconductor

Rabbit 2000 embedded microcontroller as IRMA's control computer. IRMA was to be a true embedded system distributed between three processors, both of which were required to be multi-tasking and provide real time performance, meaning the system needed to respond to external interrupts in a known period of time.

A project manager for IRMA was hired in 2003 to solve some of IRMA's outstanding mechanical problems, the most important being a stable vacuum for IRMA's cooled IR detector. Concerted effort was poured into IRMA's development, so by June of 2004, IRMA III was ready for initial field tests. In February 2005, IRMA III was deployed at the Gemini South observatory at Cerro Pachon, Chile for a second round of field testing. At the time of writing, IRMA is still operational at the Gemini site.

IRMA III is being upgraded with a new motherboard and master control computer. The discussion on IRMA III contained in this thesis, however, will consider the original IRMA III model that was tested in Hawaii and Chile. This thesis will discuss the structure of the IRMA III control software, the communication mechanisms binding IRMA's software modules one to another, IRMA's command control language, and IRMA's hardware/software interface. Appendix A describing the IRMAscript language in detail is provided as a reference guide for operating the IRMA III device.

# Chapter 2

# IRMA Hardware

## 2.1 Overview

An overview of IRMA's hardware, starting with the detector box and the Alt-Az mount, will provide a background to understanding the roles and operations of the IRMA master controller (MC) and IRMA altitude-azimuth mount controller (AAC). IRMA consists of a 38 cm x 22 cm x 18.5 cm aluminum box mounted on an Alt-Az fork mount, as depicted in figure 2.1. The Alt-Az mount allows IRMA to rotate approximately 170 degrees of rotation about its azimuth axis, and approximately 185 degrees about its altitude axis. The shoebox-sized IRMA unit contains a 117 mm diameter aperture, behind which is a motorized sliding shutter. The shutter serves as a calibration source as well as waterproof the IRMA unit when it is not observing, as it makes a tight seal when it is in closed position. A 13 micron thick mylar window protects the instrument against dust during observing.

Inside the unit, light reflects off a 10 cm diameter f/1 90 degrees off-axis parabolic

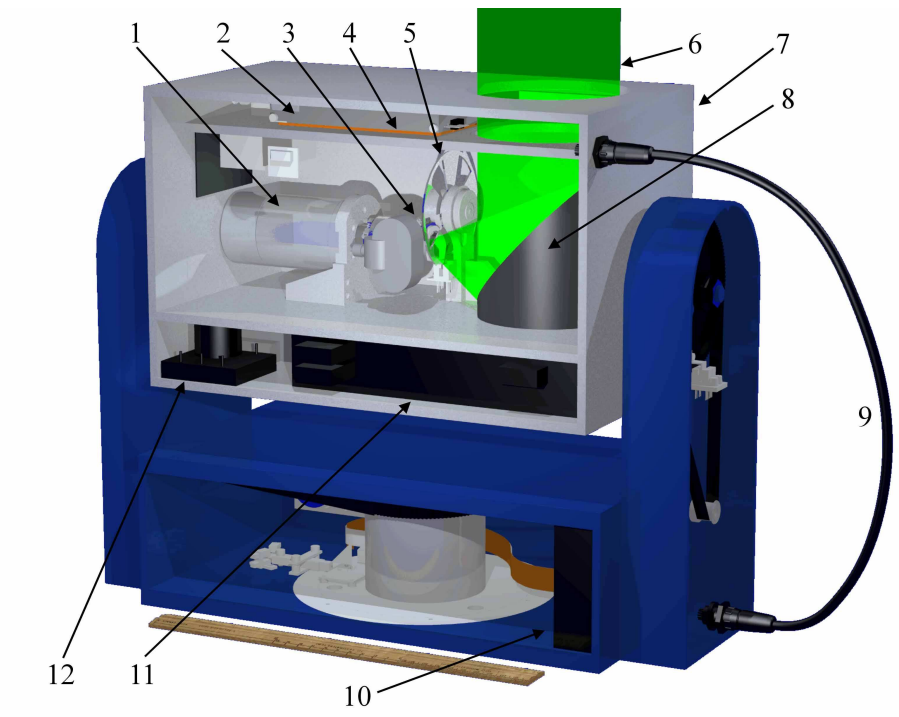


Figure 2.1: Cutaway view of IRMA in its Alt-Az mount. 1) Stirling cycle cooler 2) Shutter 3) MCT detector 4) Black body and heater 5) Reflective chopper 6) Input beam 7) Main board and IRMA master controller (hidden from view - on reverse side of detector box) 8) Parabolic mirror 9) Power/communication umbilical cable 10) Alt-Az controller 11) Cryo cooler controller 12) Power supply [38]

mirror, focusing on a 1 mm square Mercury-Cadmium-Telluride (MCT) infrared (IR) detector. The IR detector is cooled to 70 K by means of a Stirling-cycle cryo-cooler. A stainless steel vacuum vessel ($p \leqq 10^{-4}$ mbar) encloses the cryo-cooler's cold finger and IR detector. The IR detector is attached to the tip of the cold finger with a mechanical clamp. The incoming optical beam passes through a 5-blade reflective chopper wheel before reaching the IR detector. The chopper wheel blades modulate the signal at 450 Hz. This frequency was chosen as a result of spectral analysis tests, which showed that the IRMA system had lowest overall noise at this frequency. A unique notch, located on the chopper

wheel's circumference, ensures samples are triggered on the same blade, thus eliminating uncertainties associated with blade to blade emittance/reflectance variations[38].

Given IRMA's compact size in comparison to the amount of required internal hardware, little space remains for a control computer, which made it necessary to use a miniature microcontroller module in both the MC and the AAC. The MC uses a Rabbit Semiconductor RCM2100 microcontroller module. The RCM2100, pictured in figure 2.2, is an 89 mm x 51 mm printed circuit board containing an 8-bit microprocessor, memory, digital and serial I/O, and an Ethernet-based network interface controller. This microcontroller is responsible for interpreting commands from the command processor (CP), with which it commands and queries IRMA's hardware components.

## 2.2 IRMA Master Controller

### 2.2.1 Rabbit 2000 Microcontroller Module

The MC and AAC control computers are based on the Rabbit 2000 8-bit microprocessor. The Rabbit 2000 processor and its related products are produced by Rabbit Semiconductor, Inc., a fabless semiconductor company which specializes in high-performance, low cost 8-bit microprocessors and development kits. Rabbit 2000 (and its more powerful sister processor, the Rabbit 3000) processors are generally sold as small single board computers known as core modules, and are promoted as rapid development solutions for connecting systems and devices to the Internet. Typical applications include point-of-sale systems, automated utilities meter reading, and traffic monitoring[22]. Internet web searches on Google, however, show that IRMA may be the only publicized application of

Rabbit microcontrollers in astronomical instrumentation control.

Introduced to the market in 1999, the Rabbit 2000 is based on the venerable Zilog Z-80/Z-180 architecture. Consequently, the Rabbit 2000 shares a similar register layout, memory addressing modes and machine instructions with the Zilog processor. The two architectures are so similar, it is possible to execute Zilog assembly code on the Rabbit 2000. The primary difference between the two processors is that the Rabbit 2000's register layout is optimized for 16-bit arithmetic and memory manipulation, unlike the original Z-80 architecture. This feature makes the Rabbit 2000 more compatible with C language compilers, which are typically biased towards 16-bit (or higher) arithmetic and memory access. Ultimately, a processor architecture that is more in step with the target compiler's capabilities will generate more efficient machine language programs.

The RCM2100 core module in its maximum outfitted configuration features a 22 MHz Rabbit 2000 8-bit microprocessor, 512 KB of static random access memory (RAM), 512 KB of non-volatile flash memory, 40 lines of TTL compatible digital I/O (DIO) lines, eight of which serve as 4 serial communication channels, and a 10Base-T 10 Mbit/s Ethernet controller. A comprehensive listing of the RCM2100's capabilities is shown in table 2.1.

**Rabbit 2000 Memory Structure**

Memory is a scarce resource in embedded computers, primarily due to their small data size (8-bit) and consequently small memory address space. Although the Rabbit's software development environment, Dynamic C, largely insulates the programmer from the intricacies of Rabbit memory management, it is worthwhile to examine how memory is

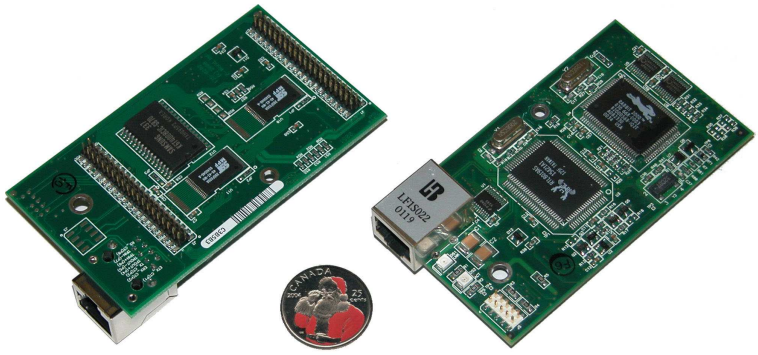

Figure 2.2: Rabbit RCM2100 Core Module (front and reverse view).

organized and handled. At the very least, this knowledge is helpful in understanding and diagnosing runtime memory errors, which are typically difficult to resolve on any platform.

Flash memory is used to permanently store the IRMA MC executable code and related static data, such as constants, tables and files. Volatile SRAM holds the executing program and its associated variables. Rabbit 2000 program size is limited by the amount of available flash memory. The maximum amount of flash RAM supported on Rabbit processors is 512 KB. Although 512 KB (roughly equivalent to 25,000 to 50,000 C-language statements) does not sound like a great deal of memory, it is more than adequate for running serious control and data acquisition programs, because the Rabbit's C language compiler, Dynamic C, produces lean and efficient executable code. SRAM and flash RAM together add up to 1024 KB, and is addressable using 20-bit address space referred to as physical memory[55].

The Rabbit 2000, being an 8-bit microprocessor, operates within a 16-bit address space derived from a larger 20-bit physical memory pool. Addressing space is kept small in order to keep Rabbit executable files small and code execution fast. The Rabbit does

| Feature | RCM2100 |
|---|---|
| Microprocessor | 22 MHz Rabbit 2000 |
| Memory: Flash | 512 KB |
| Memory: SRAM | 512 KB |
| Networking | 10Base-T Ethernet + RJ-45 |
| Serial | 4 channels, max 115 kbps (async) |
| DIO | 40 TTL lines |
| Real Time Clock | yes |
| Timers | Five 8-bit times, one 10-bit timer |
| Connectors | Two 2x20 pin, 2mm IDC headers |
| Power | 5V +/- 0.25V, 140 mA |
| Dimensions | 89mm x 51mm x 22mm |

Table 2.1: Rabbit 2100 Core Module Specifications [50]

not have 32-bit wide registers. As a result, performing 32-bit arithmetic requires more processor cycles than performing internally-supported 16-bit calculations. Since the Rabbit 2000 cannot access 20-bit physical memory addresses, it uses a segmented memory scheme, whereby its built-in memory management unit (MMU) maps the 20-bit physical address space to the smaller 16-bit logical address space. The Rabbit 2000's memory structure, showing the mapping relationship between logical and physical memory, is shown in figure 2.3.

Memory addresses from 0 to $2^{16}$ comprise root memory, while addresses above this boundary up to $2^{20}$ comprises extended memory. Root memory can be manipulated directly using C-language assignment statements, but extended memory can only be accessed using Dynamic C's extended memory routines such as *xalloc*, *xmem2root*, and *root2xmem*.

Within root memory (or logical address space) are four segments: the base segment, data segment, stack segment and extended memory segment. The base segment can

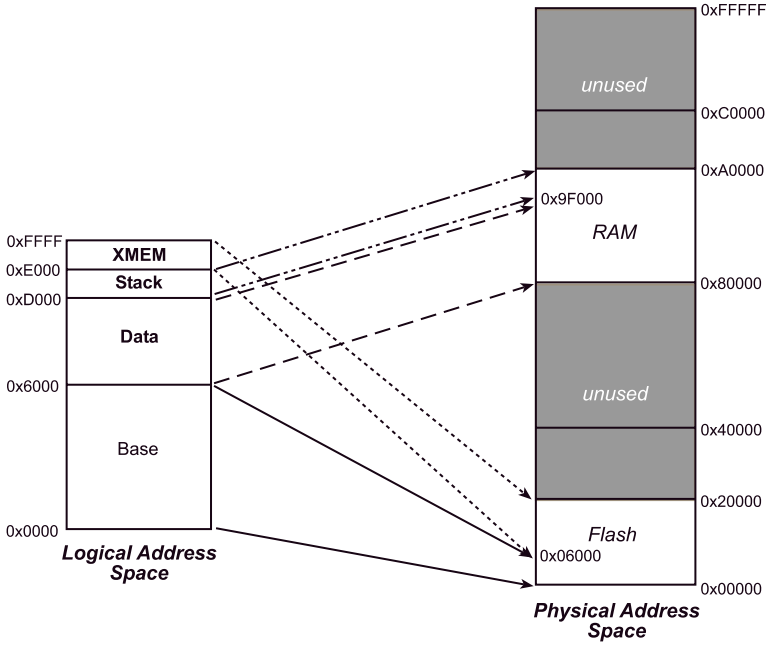


Figure 2.3: Rabbit 2000 memory mapping between logical and physical address space[21].

be used for storing speed-critical or short-length functions, interrupt service routines (if any), and the Rabbit BIOS (Basic Input/Output System). It is typically 24 KB in size, and is mapped to flash memory where executable code is stored when running the Rabbit in non-debug mode.

Above the base segment is the data segment, which is mapped to SRAM. It is used for storing run-time variables, and extends to address 53,238 (D000 in hexadecimal). The size of the root and data segments can be adjusted, but together they cannot exceed 52 KB. Global variables as well as pure assembly language functions are placed in these segments.

Above the data segment is a 4 KB region called the stack segment. Positioned between addresses D000 (hex) and E000 (hex), the stack segment contains the Rabbit

system stack, and is mapped to SRAM. The system stack is used for storing variables local to a function that exist only for the duration of the function call. They are declared using the *auto* directive. Dynamic C by default treats all local variables as auto. One consequence the Rabbit developer should be aware of is that all the local (auto) variables contained in a function cannot exceed 4 KB (4096 bytes) of memory storage.

The extended memory segment sits between address E000 (hex) and 10000 (hex). This 8 KB region is used to execute extended code as well as act as scratch memory space for routines that manipulate extended memory. For the most part, memory management is transparent to the software developer, as the Dynamic C compiler and memory handling libraries take care physical/logical memory mapping. The developer sees only a flat 20-bit address space [21].

### 2.2.2 Rabbit 2000 Input/Output

#### Parallel Ports

The Rabbit 2000, like other embedded microcontrollers, excel at providing copious amounts of I/O, since their primary application is hardware control. Five 8-bit wide parallel ports are featured on all Rabbit 2000 processors, making available a maximum of 40 TTL-compatible (0 to 5 V) DIO lines. Since the Rabbit maps some of these lines for multiple uses, such as Ethernet or the Rabbit slave port, the total available DIO lines will decrease depending on which RCM2100 functions the programmer wishes to use.

Each of the Rabbit 2000's parallel ports have particular characteristics in terms of

| | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| **PA** | PA7 | PA6 | PA5 | PA4 | PA3 | PA2 | PA1 | PA0 |
| | All pins must be set collectively as input or output | | | | | | | |
| **PB** | PB7 | PB6 | PB5 | PB4 | PB3 | PB2 | PB1 | PB0 |
| | out | | in | | | | | |
| **PC** | PC7 | PC6 | PC5 | PC4 | PC3 | PC2 | PC1 | PC0 |
| | in | out | in | out | in | out | in | out |
| | serial A (debug) | | serial B | | serial C | | serial D | |
| **PD** | PD7 | PD6 | PD5 | PD4 | PD3 | PD2 | PD1 | PD0 |
| | in/out | in/out | in/out | in/out | in/out | in/out | in/out | in/out |
| | Ethernet | | | | | | | |
| **PE** | PE7 | PE6 | PE5 | PE4 | PE3 | PE2 | PE1 | PE0 |
| | in/out | in/out | in/out | in/out | in/out | in/out | in/out | in/out |
| | | Ethernet | | int 1B | int 0B | Ethernet | int 1A | int 0A |

Figure 2.4: Rabbit 2000 Parallel Ports.

their flexibility in setting the data direction of their I/O pins, their potential shared usage, and if applicable, their electrical characteristics. As shown in figure 2.4, only ports D and E allow data direction to be set at the pin level, while port B and C have fixed data direction assignments. Four serial channels are mapped to parallel port B, where each serial channel maps to a pair of DIO lines (one for transmit, the other for receive). When Ethernet is enabled, six DIO lines (four in parallel port D and two in port E) are reserved. Serial port A is assigned to carry the Rabbit-PC debug channel. This channel is used to upload software into the Rabbit, or to receive feedback from *printf* statements embedded in the executable when the Rabbit is run in debug or diagnostic mode. Parallel port E contains two lines dedicated to Ethernet, as well as four external interrupt lines. The Rabbit 2000 parallel port data direction registers, PDDDR and PEDDR, control whether a parallel port pin is set in read or write mode.

The Rabbit 2000 has two external interrupt channels, each of which is mapped to two pins, permitting up to four external interrupt lines to be connected. Two unique priority interrupts are assigned to each interrupt channel. Rabbit 2000 processors shipped before January 2002 contain a bug in their interrupt pulse edge detection circuitry, which in certain situations could cause spurious interrupts. The manufacturer's recommended workaround[20] halves the number of usable interrupt lines. Although this was a serious design issue early on in IRMA III's design, all current IRMAs use the newer, bug-free Rabbit 2000 processors. The older Rabbit processors can be identified by the version code *IQ2T*.

**Ethernet**

Rabbit 2000 processors do not support networking internally, as they do not contain Ethernet control circuitry. Selected Rabbit 2000 based controller modules, however, do support networking through an external network interface controller chip. Controller modules, such as the RCM2100 used in the IRMA MC, use the RealTek 8019 network interface controller (NIC). All network-capable processor modules have an RJ-45 socket allowing connection to a local area network using a standard Cat-5 (EIA/TIA-568) network cable. Rabbit networking is powered by the Rabbit processor, causing it to be considerably slower than networking performed on a desktop computer. This is due to the fact that the rate at which the microprocessor can process network packets is limited by its clock speed and data width. The Rabbit can at best transmit 270 KB/s on a traffic-free network, one quarter the rate at which PC hardware can process network traffic[54]. Rabbit networking is a major component of Dynamic C, supporting high-level server protocols such as HTTP, Telnet and FTP in addition to TCP and UDP sockets. Being a software matter, Rabbit

networking using Dynamic C is beyond the scope of this discussion. Fortunately, Rabbit Semiconductor has provided extensive tutorial[45] and reference documentation[65][47][48], as well as program examples[46] relating to Rabbit network programming.

**Rabbit 2000 Peculiarities**

One cannot expect modern PC performance from 8-bit microcontroller modules such as the Rabbit RCM2100, nor does it have features deemed standard in conventional 32-bit computers. Many of these features, such as protected memory, file systems or preemptive multitasking using priority round-robin scheduling, are features of the host operating system, not the hardware. The Rabbit software development kit, Dynamic C, provides libraries which provide rudimentary network services, disk-less file system and multitasking. A real-time multitasking kernel, MicroC/OS-II [27], is provided for implementing preemptive real-time multitasking.

The Dynamic C's lack of double precision arithmetic functions makes it more difficult for the Rabbit to do precision arithmetic. Since the Rabbit only supports single-precision floating point numbers, round off error can creep into Rabbit-based arithmetic routines rapidly. With IRMA, nearly all floating point arithmetic tasks, such as altitude-azimuth to right ascension-declination coordinate conversion, have been offloaded onto the PC-based CP in order to preserve arithmetic precision, increase speed of program execution, and take advantage of higher level languages (like Perl) and external libraries that require less developer effort.

## 2.3 IRMA MC Control

The IRMA MC forms the hub of the IRMA control and data acquisition system. The MC is tasked with controlling each of the electronic devices interfaced to it. Control tasks include turning a unit on or off, commanding it to do some task (either by setting a logic level or sending an explicit command with parameters), monitoring its status, and responding to external interrupts. Communication with IRMA's hardware components is performed through digital I/O lines or 2-wire serial channels. A block diagram showing the MC's hardware interfacing appears in figure 2.6.

The digital I/O and serial port mappings on the IRMA MC, appearing in figure 2.5, show that roughly two thirds of IRMA's I/O is devoted to output (blue boxes), while one third is devoted to input (pink boxes). Colored boxes outlining one or more boxes depict DIO lines reserved for specific functions. Parallel port C maps four sets of read and write lines to four serial channels. The red box spanning lines 4 through 7 on parallel port D, along with lines 6 and 2 on parallel port E are reserved for Ethernet communications when networking is enabled. Enabling additional hardware functionality on the Rabbit consumes even more DIO lines – an important consideration when planning hardware interfacing at the outset of a project. The Rabbit slave port driver demonstrates how using extra features rapidly consumes DIO resources. Inclusion of this driver into the IRMA III design was dropped when it was realized the slave driver would require 14 DIO lines. In addition to the lines lost to Ethernet, less than 10 DIO lines on the RCM2100 would remain for interfacing peripheral hardware. This was the factor that led to the decision to use the

relatively slow 2-wire serial connection linking the master and Alt-Az controllers.

| | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| Parallel A | Alt-Az Controller Reset | MUX 3 | MUX 2 | MUX 1 | Chop Interrupt Enable | ΔΣ ADC SCLK | ΔΣ ADC SDI | Sun Shutter |
| Parallel B | | | GPS Timemark | BB Shutter Motor Overcurrent | Shutter Limit 2 | Shutter Limit 1 | Sun Sensor | ΔΣ ADC SDO |
| Parallel C | Used by programming port RX | Used by programming port TX | Cryo Cooler RX | Cryo Cooler TX | GPS Receiver RX | GPS Receiver TX | Master Controller / Alt-Az RX | Master Controller / Alt-Az TX |
| Parallel D | ARXA In Eth BD7 | ATXA Out Eth BD6 | ARXB In Eth BD5 | ATXB Out RSTDRV | BB Shutter | BB Heater | Power Monitor +24V OK | Power Monitor +5V OK |
| Parallel E | Bandpass Filter 455 Hz | Ethernet IOWB I6 Output | Notch Filter 60 Hz | BB Shutter OC Latch Reset | Optical Chopper | Ethernet IORB I2 Output | Sample Interrupt SDO1 | Not used |

Figure 2.5: IRMA Master Controller Digital I/O Pin Mapping. Pink boxes represent input lines, blue boxes represent output lines, white boxes represent bidirectional lines.

### 2.3.1 Shutter

The shutter, which also serves as a calibration source, consists of a 130 x 137 x 17 mm hollow aluminum block mounted in a track, driven by a lead screw. At opposite ends of the track are two slotted optical switches[60], both of which are mapped to two DIO lines. When the optical beam is open, a logic value of 0 is returned. When the beam is closed, a value of 1 is returned. Metal tabs that actuate the opto switches are placed at opposite sides of the shutter. The IRMA MC software polls these lines and returns the values to the

IRMA CP, which it uses to determine when shutter movement has completed.

DIO lines PB2 and PB3 (parallel port B, bits 2 and 3), are mapped respectively to the shutter-closed and shutter-open opto switches. The fact that both opto switches are open (both reading high) when the shutter is not in the open or closed position provides shutter status: these two bits, when shifted into bit positions 0 and 1, can be interpreted as status codes:

| **Code** | **State** |
|---|---|
| 1 | open |
| 2 | closed |
| 3 | moving (or jammed) |

The shutter is commanded to open by clearing bit 3 of parallel port D. The shutter closes by setting bit 3. There is no way to set speed or stop the shutter once it has been set in motion. Digital logic in the central electronics stops shutter motion automatically once one of the opto sensors has been interrupted. Shutter software traps for the case that the motor may not automatically turn off once it reaches its destination position by reversing the current shutter direction if the shutter does not finish moving in some predetermined time. The default timeout is 40 seconds. This is an attempt to minimize damage if the shutter jams.

### 2.3.2 Sun Shutter

The parabolic mirror inside the IRMA unit collects and focuses light at the detector. If the unit is pointed directly at the sun, the focused sunlight is intense enough to

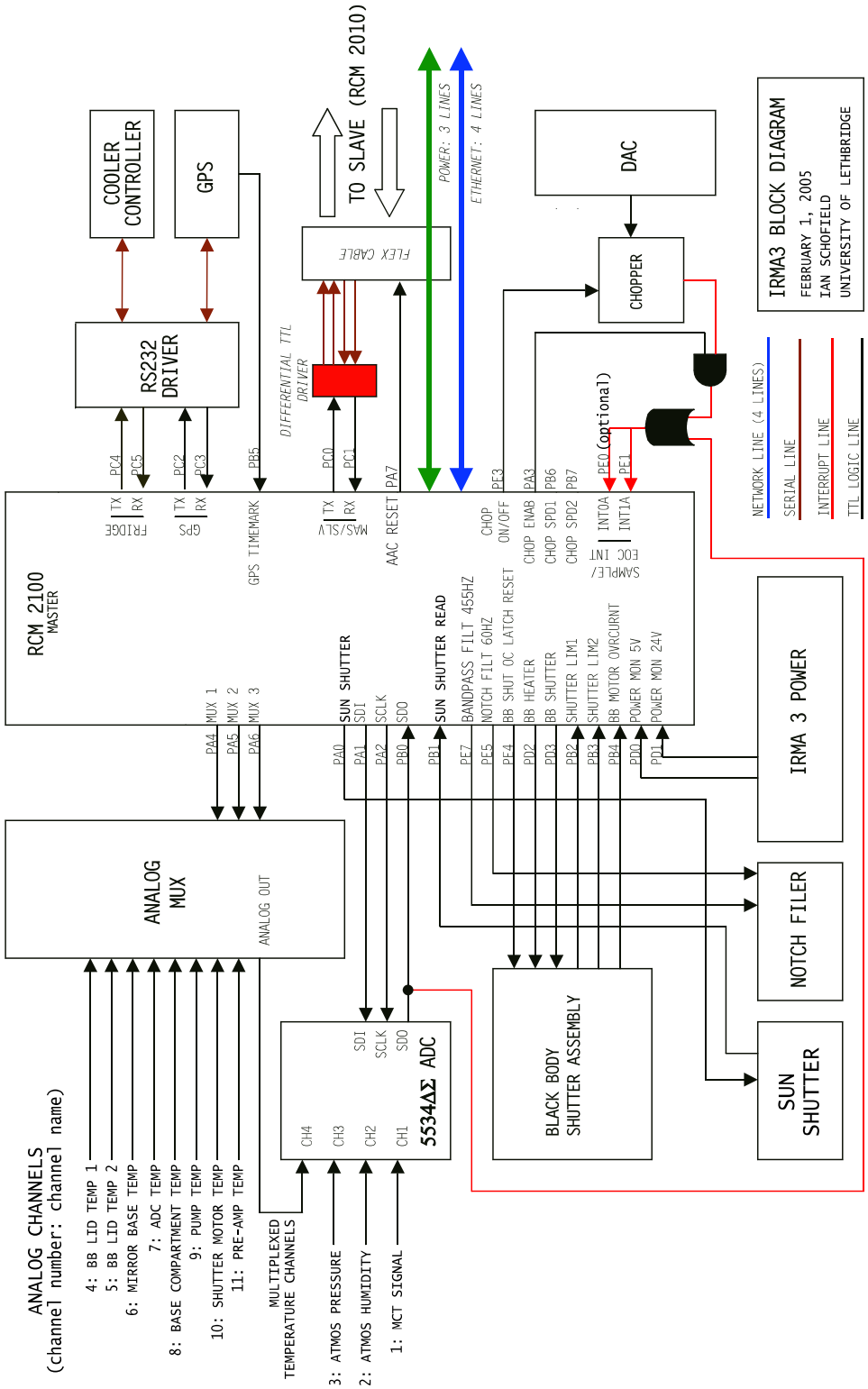


Figure 2.6: IRMA Master Control hardware block diagram and pin mappings.

burn a hole in the filter covering the detector. This has occurred twice in the past with earlier models of IRMA. To prevent this from happening, a solenoid-operated shutter independent of software, sweeps into place whenever a bright light body, such as the sun, comes within 15 degrees of the detector's field of view. A small hole, on-axis to IRMA's field of view, contains a photocell that detects bright light sources. The sun shutter can also be controlled via software to open or close, and is mapped to line 0 on parallel port A. Setting this line opens the sun shutter, while clearing it causes the sun shutter to close. Manual sun shutter control is useful for certain diagnostic tests and provides additional protection during testing and commissioning phases.

### 2.3.3 Calibration Source

A calibration source, attached to the underside of the shutter, is used to calibrate IRMA's IR detector. It consists of a carbon-black epoxy enamel textured coating deposited on a thin, metallic film heater. The coating has a high emissivity at infrared wavelengths. The blackbody can be heated by passing an electrical current through the film. Current is turned on or off by setting or clearing bit 2 of parallel port D. When the shutter is closed (where it covers the optical aperture) the blackbody is in position for taking calibration measurements.

### 2.3.4 Stirling-Cycle Cooler

A Hymatic NAX025-001 Stirling-cycle cryo cooler is responsible for cooling IRMA's IR detector to 70 K. The cylindrically-shaped unit is equipped with a vacuum chamber. The vacuum is required by the cryo cooler to reach cryogenic temperatures. The IR detec-

tor is attached to the tip of the cold finger, which is the only part of the cryo cooler which achieves cryogenic temperatures.

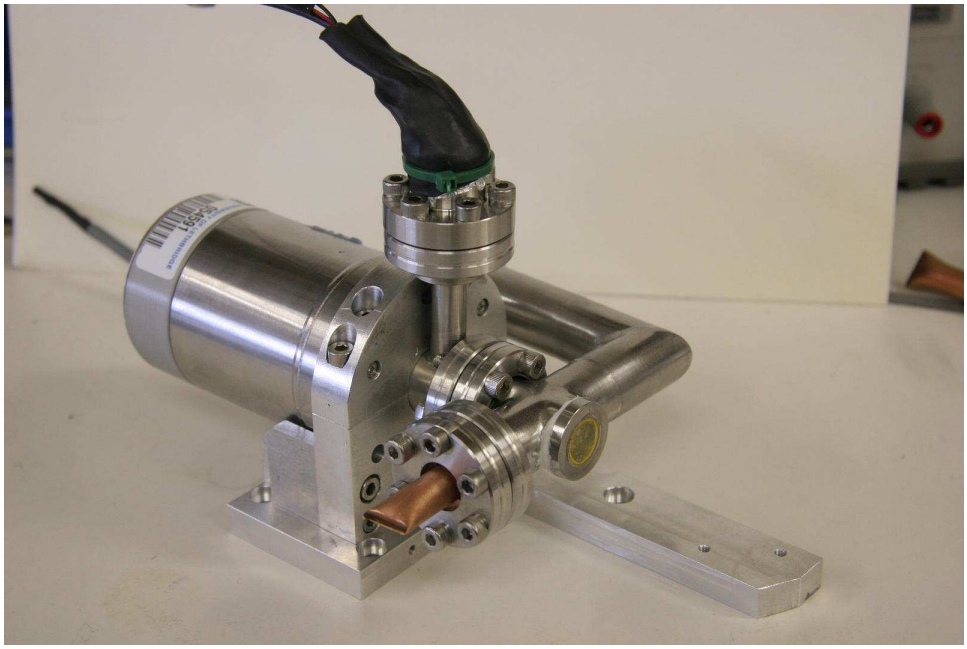

Figure 2.7: IRMA vacuum vessel. Wiring for the cold finger temperature sensor and the detector output is fed through the small tube pointing up. The getter is located in the long elbow section to the right. The pinch off tube is connected in the left hand flange.

The vacuum chamber surrounding the cold finger is evacuated to 1x10$^{-4}$ mbar. This vacuum, which is designed to last for roughly five years, must have a leak rate no greater than 1x10$^{-15}$ mbar cm$^{-2}$ s$^{-1}$ in order to allow the cryo cooler to operate at its target temperature. The chamber, shown in figure 2.7, is a T-shaped vessel with two arms on either side and an anti-reflective-coated ZnSe window. IR radiation enters the vacuum vessel through this window, illuminating the IR detector directly behind it. The two arms act as access points to the chamber. One arm connects to the vacuum pinch off tube, a

copper tube which connects to a turbo-pump during evacuation. While attached to the pump, the tube is pinched off using a precision crimping tool, which cold-welds the copper tubing, creating a permanent vacuum seal. The vacuum chamber is e-beam welded to the cooler body. The other arm contains a SAES ST172/HI/16-10/300C getter, a device designed to absorb gas molecules that naturally outgas from the vacuum vessel walls. The getter is activated by passing an 4 amps of electrical current through it for 5 minutes, heating it to 900 C[43].

The Hymatic Stirling-cycle cryo cooler controller unit accepts high-level commands encoded in ASCII strings over its RS-232 serial port (female DB-9 connector), which allows for interfacing to external computer hardware. IRMA communicates with the cryo cooler controller over serial port B, which is mapped to parallel port C, lines 7 and 6 (PC7 and PC6). PC6 is the serial transmission (TX) line, while PC7 is the serial receive (RX) line. It should be noted that for all of the Rabbit's serial lines on parallel port C, the odd lines (7, 5, 3, 1) are TX lines, while the even lines (6, 4, 2, 0) are RX lines.

IRMA command packets are translated into appropriate Hymatic serial strings and sent to the controller in order to control the cryo-cooler's behavior. Likewise, data from the cryo-cooler, such as cooler temperature, is extracted from the Hymatic serial data strings and encoded into IRMA data packets. The cryo-cooler serial communication channel operates at 4800 bits per second, 8 data bits, 1 stop bit, no parity (8N1). Rabbit/Cryo-cooler control is encapsulated in the custom-written **hymatic.lib** Dynamic C library.

Commanding the cryo-cooler to go to a target temperature is straight forward: one sets the cooler's set point to some temperature in degrees Kelvin, then sets the cooler

into auto mode. The cryo-cooler then begins the process of cooling down at a set rate, based on a factory-set internal profile, until it reaches target temperature. The cryo-cooler will maintain its set point until instructed otherwise. Turning off the cryo-cooler involves setting it to manual mode, then setting it to stopped mode. It is not desirable to cut power to the cooler during operation, as this may damage the piston that oscillates inside the cold finger.

### 2.3.5 Alt-Az Controller

The Alt-Az controller (AAC) is a custom-built electronics board based around a Rabbit Semiconductor RCM2010[49] controller module. The AAC acts as a slave on behalf of the MC, as it does not perform actions or generate data on its own. It only acts when commanded by the MC by means of a 19.2 kbps 2-wire serial channel, mapped to both Rabbit's serial port D (lines 0 and 1 on parallel port C).

The AAC is responsible for moving the Alt-Az mount to specified elevation and azimuth coordinates, thus it concerns itself completely with motion control and communicating with the MC. Alt-Az control is offloaded onto a separate processor because the MC lacks the DIO line capacity required to serve all hardware control functions. Additionally, the MC is already burdened with handling network communication, data acquisition, and device control duties. Details on the MC-AAC serial communications protocol is contained section 3.6.2.

### 2.3.6 IR detector

Infrared radiation is detected and converted to measurable voltages by a MCT photoconductive detector, manufactured by Kolmar Technologies. The detector is sensitive to wavelengths from 5 to 20 microns. A 19 micron highpass filter placed in front of the IR detector filters out wavelengths less than 19 microns, resulting in a narrow 2 micron ( 50 $cm^{-1}$) wavelength band of radiation reaching the detector. The detector changes its resistance as a function of the radiation falling upon it. This change is sensed as a voltage, which is fed to the ADC. The signal voltage is a measure of flux (in watts) from 20 micron emissions reaching the detector, and is proportional to the strength of the 20 micron absorption line. A radiative transfer model developed by Ian Chapman during his thesis work at the University of Lethbridge, called ULTRAM (University of Lethbridge Transmission and Radiance Atmospheric Model) is used to convert the line strength into millimeters of PWV[5].

### 2.3.7 Chopper Wheel

A 5-blade reflective chopper wheel modulates the incoming optical beam at roughly 450 Hz. A notch on the perimeter of the wheel is used as a sample trigger point to force A/D sampling on the same blade, eliminating signal variation due to dirt and imperfections on each of the chopper wheel blades[38]. The chopper wheel's rotation rate determines the A/D sampling rate, as the chop notch signal is mapped to the IRMA MC's external interrupt line. When the MC detects a low to high transition on its interrupt line, it calls its data collection interrupt service routine, implemented almost entirely in assembly code.

The chopper wheel is enabled and disabled by setting or clearing bit 3 on parallel port E.

Chopper wheel speed is user selectable by setting bits 6 and 7 on parallel port A. Because only two bits are available for speed settings, four distinct speeds can be selected. The speed setting is fed into a serial DAC, which presents a corresponding voltage level to the chopper wheel's motor control module. This function has been deprecated in future versions of IRMA III. For all IRMA models, the chopper wheel's default rotational speed is 5400 rpm.

## 2.4 Delta Sigma Analog to Digital Converter

The heart of the IRMA data acquisition system is a Cirrus Logic CS5543[8] 4-channel 24-bit delta sigma ($\Delta\Sigma$) analog to digital converter (ADC). The IR signal, atmospheric pressure, relative humidity and eight temperature channels are sampled by the ADC. Given that the ADC has only four input channels, the eight temperature channels are selected via an 8-channel analog multiplexer (MUX), permitting the 4 channel ADC to accept eleven signal sources. The Maxim MAX4638 8-to-1 analog MUX is mapped to DIO lines 4, 5 and 6 on parallel port A. Line 6 is the most significant bit (MSB) and line 4 is the least significant bit (LSB) of this 3-bit sequence. Placing binary values 0 through 7 on these three lines selects one of the MUX's eight channels.

Unlike other ADC designs, the $\Delta\Sigma$ ADC contains a relatively simple 1-bit analog A/D sampling module combined with sophisticated digital signal processing circuitry. One of the benefits of the $\Delta\Sigma$ is that it is primarily a digital device, making it inexpensive to produce, as well as being linear across its input voltage range, as it has only two analog

inputs. ADCs are capable of performing very high resolution A/D conversions despite only being able to sample at 1-bit resolution because they use of massive oversampling, noise shaping and digital filtering to achieve near 24-bit sample resolution [24].

### 2.4.1 Delta Sigma Signal Processing

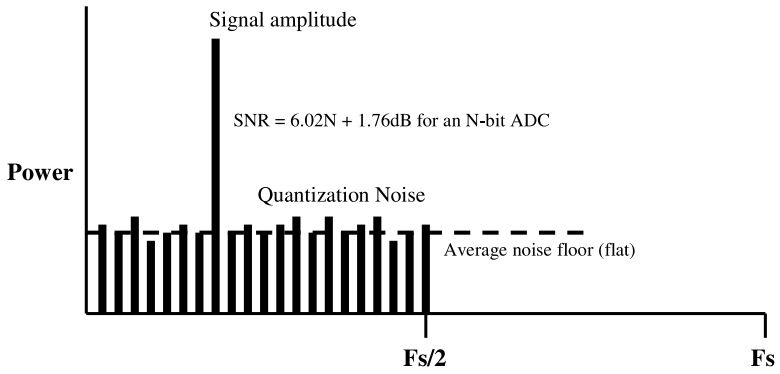


Figure 2.8: FFT diagram of an n-bit A/D conversion with sampling frequency Fs. Diagram[44].

Oversampling can be visualized by taking the Fourier transform (FT) of the signal and plotting its power versus frequency. As shown in the figure 2.8, the input signal contains a single frequency, which appears as a single frequency bin. Noise, however, is distributed evenly across the signal bandwidth from 0 Hz to half the sampling frequency, defining the signal's noise floor.

Oversampling (figure 2.9) involves sampling the input signal at rates higher than twice the Nyquist frequency. In essence, oversampling uses signal averaging to reduce quantization error that manifests itself as noise in the signal by repeatedly sampling the

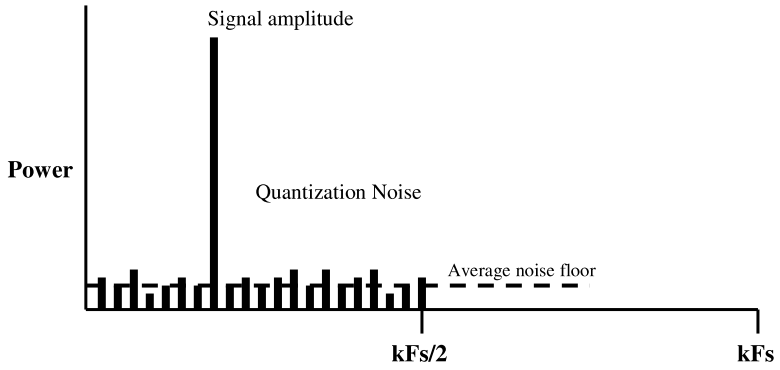


Figure 2.9: FFT diagram of an n-bit A/D conversion with sampling frequency kFs, oversampled by k times. Noise floor has been lowered due to oversampling. [44].

signal and calculating the average signal value. Signal averaging improves the signal to noise (SNR) by the square root of the number of samples[33]. For example, if a signal is sampled 100 times, the average signal remains the same, while the noise, assumed to be random, is reduced by a factor of $\sqrt{100}$, or 10[12].

The SNR of a sample obtained from an $N$-bit $\Delta\Sigma$ ADC is shown to be[24]:

$$SNR = 6.02N + 1.76dB \tag{2.1}$$

which implies that a 1-bit A/D conversion has an SNR equal to 7.78 dB. Clearly, a higher SNR can be achieved by increasing $N$, the number of sampled bits of precision. This is not possible with $\Delta\Sigma$ A/D converters, as they only contain a 1-bit converter. Increasing the oversampling rate on a 1-bit ADC by a factor of 4 increases the SNR by 6 dB, which corresponds to a single bit increase in sample resolution. Quadrupling the oversampling rate for each additional bit of precision can lead to excessively high oversampling rates: to achieve a 24-bit resolution sample, $4^{23}$ times oversampling would be required.

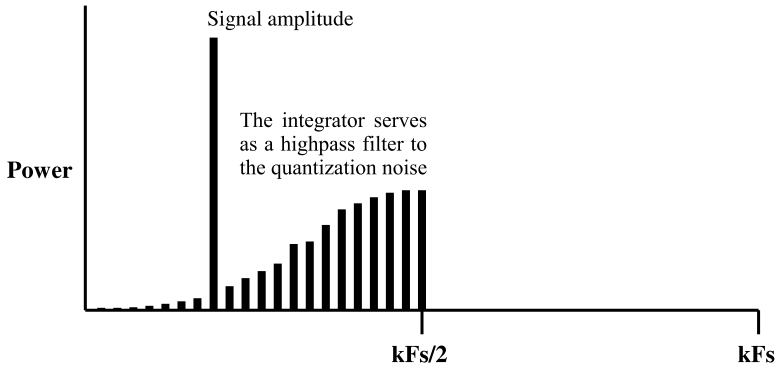


Figure 2.10: Effect of the Delta-Sigma modulator changing the distribution of high-frequency quantization quantization noise, or noise shaping[44].

The $\Delta\Sigma$ modulator deals with the limitation of oversampling as a means to increase resolution by pushing high-frequency noise beyond the frequency range of interest (figure 2.10), resulting in the attenuation of 9 dB of quantization noise for every factor of 2 increase in the oversampling ratio. It is now feasible to achieve a high SNR (low quantization noise) with a moderate oversampling rate. The $\Delta\Sigma$ ADCs integrator is responsible for this effect, called noise shaping. Again, the total quantization noise has not dropped, but rather its distribution along the bandwidth has been changed.

One or more sinc filters are used to filter out the remaining quantization noise. By filtering out frequencies beyond the band of interest (figure 2.11), the low frequency bands are relatively noise-free, enjoying a superior SNR. A time decimation filter placed after the low-pass filters are used to reduce the data rate of the output data stream[24].

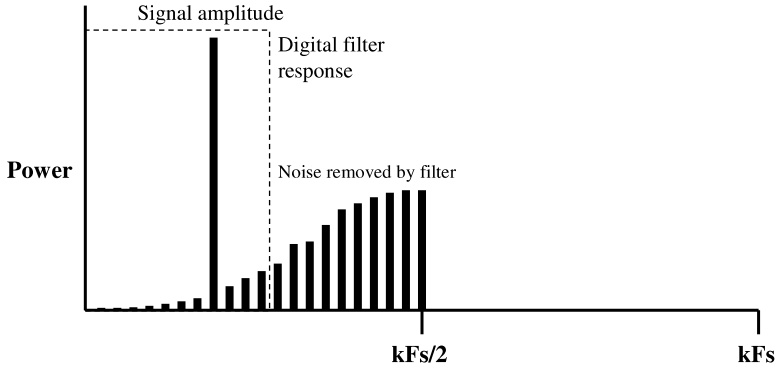


Figure 2.11: Affect of a digital filter on quantization noise[44].

### 2.4.2 Structure of the Delta Sigma ADC

A first-order $\Delta\Sigma$ modulator is a simple A/D converter design consisting of a difference amplifier, an integrator, a comparator (1-bit ADC) and a 1-bit DAC, as shown in figure 2.12. An input signal $X_1$ feeds into the difference amp, which outputs the difference in volts between the analog output of the modulator and the input signal. This is the *delta* portion of the delta sigma modulator. The *delta* output $X_2$ is fed into an integrator, the sigma, which calculates a rolling average of the input signal. The sigma output $X_3$ is then sampled with a comparator, which acts as a 1-bit ADC. If the sigma signal is greater than ground, the comparator outputs a 1 (full scale voltage), otherwise it outputs a 0 (ground). The resulting bit stream from the comparator $X_4$ is split: one half goes to the digital filter section of the $\Delta\Sigma$ modulator, the other half is fed back into the difference amp after passing through a 1-bit DAC. The DAC output $X_5$ is full scale voltage if the input is greater than ground, or 0 volts otherwise.

The bit stream of 1s and 0s emerging from the comparator, when averaged over $N$

samples, gives a value indicating the proportion of ones to zeros. The density of ones in the output bit stream indicates the proportion of the input voltage to full scale. For example, if the average of the output bit stream from the $\Delta\Sigma$ modulator read 0.5, 50 percent of the bits in the bitstream are ones, indicating that the ADC input voltage is close to 50 % of full scale. The higher the number of samples included in the average, the greater the accuracy of the A/D sample value. Consequently, high resolution A/D conversions taken with a $\Delta\Sigma$ ADC require a high degree of averaging, resulting in a high latency between taking the sample and producing the result. It is the issue of latency which makes $\Delta\Sigma$ ADCs unsuitable for sampling rapidly changing, high frequency sources.

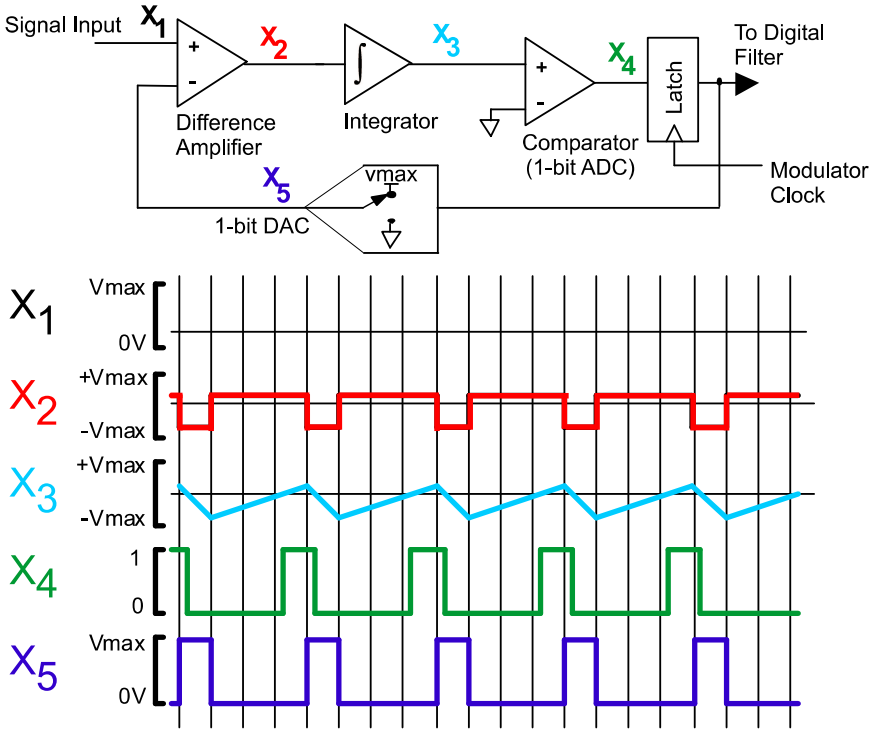


Figure 2.12: A first-order Delta-Sigma modulator[44]

### 2.4.3 Cirrus CS5534 Delta Sigma ADC Structure and Operation

The CS5534 is a serial controlled device, yet it does not use one of the Rabbit's serial channels. Rather, it uses a clocked 3-wire serial interface where each line is mapped to a discrete DIO line. Serial data must be explicitly modulated on its DIO lines by the Rabbit. When the CS5534 is enabled by holding its chip select (CS) pin low, serial commands are fed into its serial data in (SDI) line, which is mapped to Rabbit DIO output line 0 on parallel port B. Data from the CS5534 is received on DIO input line 1 on parallel port A. The CS5534's serial clock input (SCLK) must must transition from low to high in order make the $\Delta\Sigma$ accept a bit of data.

For example, if one were to input the hexadecimal number A (decimal 10) into the CS5534, one would input the bit pattern 1010 one bit at a time into SDI, strobing the SCLK pin (low to high) between each bit. Likewise, when reading data from the CS5534, one would set the SCLK line, read the SDO line, then clear the SCLK line. The strobing sequence must be repeated for each bit being read or written. The CS5534's read and write cycles are shown in figure 2.13.

The data conversion cycle begins with a command requesting an A/D conversion. The request is sent in the form of a serial stream of hi-low bits sent over the ADC's SDI line. Once the stream has been received, the ADC clears the SDO line, which is normally high, and does not set it again until the signal integration period is complete. This period ranges from 1.5 ms to 538 ms, depending on the ADC channel's word rate configuration. When the SDO line transitions from low to high at the end of the integration period, it alerts the Rabbit 2000 controller that the sample is ready to read. To read the sample,

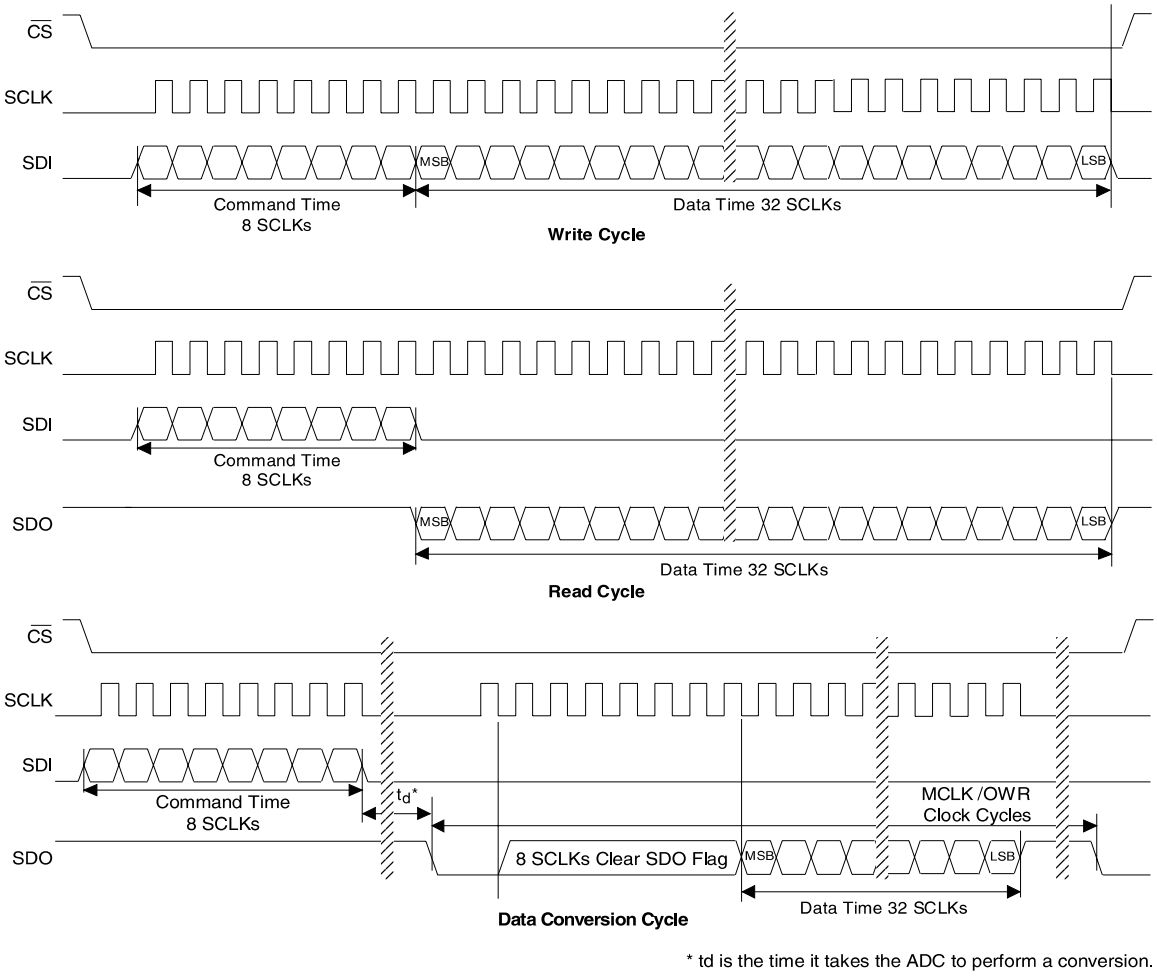


Figure 2.13: CS5534 Delta Sigma ADC Timing Diagram[8].

the SCLK must be strobed 8 times (low followed by high), after which 32 data bits can be strobed out. The readout data appears on the SDO line. The resulting sample is contained in the most significant 24 bits of the 32 bit word. The remaining 8 bits are discarded.

The maximum communication rate with the CS5534 is limited by the maximum signaling rate of the SCLK. The minimum time span between signal transitions on the SCLK line is 250 ns (4 MHz). Given that the Rabbit's maximum signaling rate using

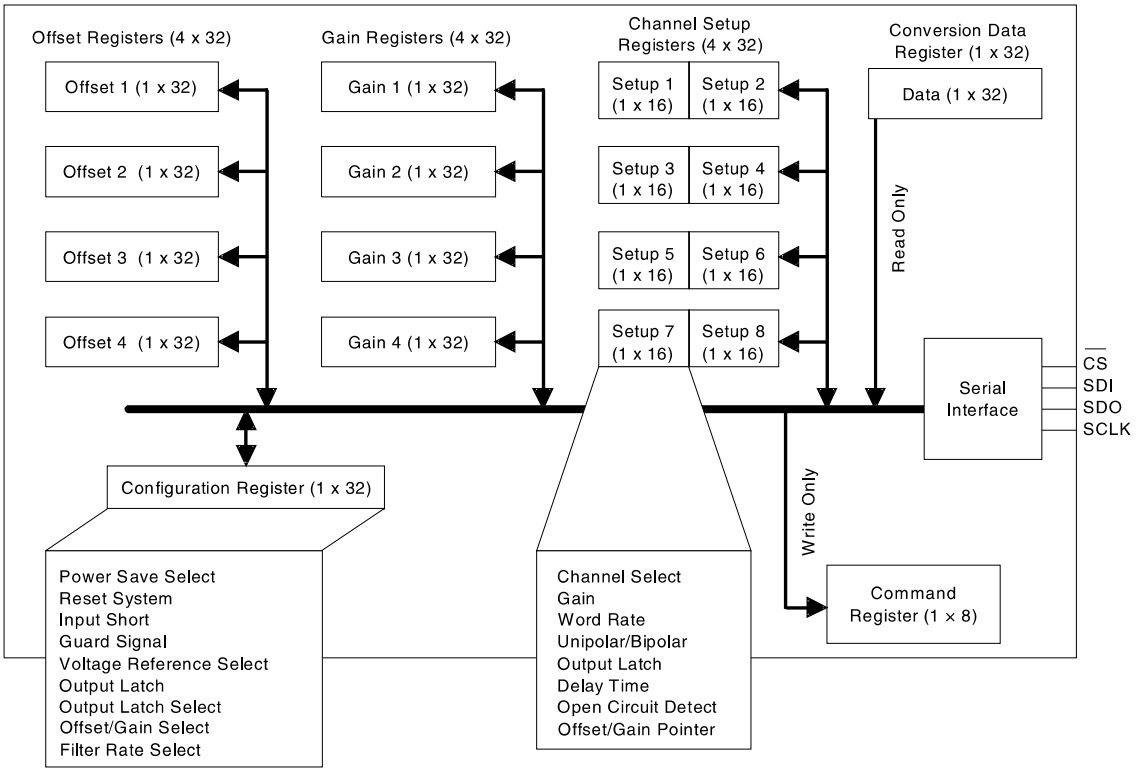


Figure 2.14: CS5534 Delta Sigma ADC Register Layout.[8]

highly optimized assembly language I/O routines is 1 MHz, there is no chance of the Rabbit controller overrunning the CS5534[8].

Configuration and operation of the CS5534 $\Delta\Sigma$ ADC is performed through its relatively complex register set, as shown in figure 2.14. The CS5534 write-only command register is 8 bits wide, and accepts 8-bit command strings via its three wire serial interface. The data register, also read-only, is 32-bits wide, and holds A/D conversions. The remainder of the CS5534's register set are configuration registers. The most significant of these are its four channel setup registers (CSR) that store settings associated with each input channel: namely sample resolution, gain and polarity. Samples can be represented in either signed or

unsigned 32-bit integers by respectively configuring polarity to either bipolar or unipolar. Signed bipolar values have a range of $\pm 2^{23}$, while unsigned unipolar values range from 0 to $2^{24}$-1.

The CSR gain setting effectively amplifies the values produced from the ADC by compressing the ADC input span, and is useful when sampling low-amplitude input signals. It is defined in equation 2.2 as:

$$InputSpan_V = \frac{(VREF_{HI} - VREF_{LO})}{n \times a} \tag{2.2}$$

where $n$ can be defined as 1, 2, 4, 8, 16, 32 or 64, and $a$ is defined as 1 for unipolar conversions, or 2 for bipolar conversions. All A/D channels on IRMA are sampled with a gain of 1 and a 2.5 V input span. The 2.5 V high reference voltage is supplied to the ADC by a Maxim MAX6126_25[35] high-precision, low noise voltage reference. The low reference voltage is ground. The MAX reference voltage chip is used to provide the ADC with an extremely clean (noise free) and accurate reference voltage that is stable over temperature (3 parts per million per degree C deviation) and time (20 parts per million deviation per 1000 hours).

Word rate is the most tangible setting associated with an A/D channel. It is also often the most confusing. Word rate is not a measure of the integration period or the sampling rate, but rather a means of describing the A/D sampling resolution. Elapsed time of conversion (in seconds) can be calculated using the following word rate equations[7] using the word rates listed in table 2.2,

$$T_{convert} = \left(7608 + 5120 \times \left(\frac{3850}{OWR}\right)\right) \times \left(\frac{1}{MCLKfreq}\right) \tag{2.3}$$

$$T_{convert} = 7592 \times \left( \frac{1}{MCLKfreq} \right) \tag{2.4}$$

Equation 2.3 is used for all word rate modes excluding the lowest resolution mode, word rate = 3840, which uses equation 2.4. MCLKfreq refers to the 4.9 MHz clock signal required to drive the $\Delta\Sigma$ electronics. Typically this involves connecting a 4.9 MHz crystal to pins 11 and 12 on the CS5534 chip. OWR refers to the output word rate. These two different calculation methods stem from the fact that the CS5534 uses different filters for the low-resolution 3840 word rate compared to the other word rates.

| Word rate | Integration (ms) | Noise-free bits |
|---|---|---|
| 3840 | 1.5 | 13 |
| 1920 | 3.6 | 16 |
| 960 | 5.7 | 17 |
| 480 | 9.9 | 17 |
| 240 | 18.2 | 18 |
| 120 | 35 | 21 |
| 60 | 69 | 21 |
| 30 | 136 | 22 |
| 15 | 269 | 22 |
| 7.5 | 538 | 23 |

Table 2.2: CS5534 $\Delta\Sigma$ ADC sampling resolutions, gain setting of 1.

The `CS5534ADC.LIB` library encapsulates a collection of C functions handling configuration and data acquisition of the CS5534 using the Rabbit RCM2100.

### 2.4.4 Global Positioning System (GPS) Board

IRMA obtains accurate time and positional information from a GlobalSat DK-ER101[51] GPS receiver, pictured in figure 2.15. A compact credit card sized device 9 mm thick, the GPS board continuously emits a formatted serial string every second over its 4800 bps serial port. The serial stream does not conform to RS-232 voltage levels, requiring that the output signal be boosted with an RS232 transceiver chip, a Maxim MAX233. The RS-232 standard defines logic 1 and logic 0 signals be differentiated by a minimum of +3.0 V and -3.0 V, or a maximum of +15.0V and -15.0 V respectively. The IRMA MC is

Figure 2.15: GlobalSat ER-101 GPS module.

interfaced to the GPS via serial port C, which is mapped to DIO lines 3 and 2 on parallel port C. The serial TX line to serial C is not connected to the GPS in order to ensure that

no spurious serial data reaches the GPS serial input, particularly during system power-up, which can potentially lock up the GPS serial data output stream. Logic levels on Rabbit 2000 DIO lines fluctuate then the MC software is uploaded into the MC's flash memory.

Every second, the DK-ER101 GPS board emits a burst of ASCII data conforming to the NMEA standard. NMEA, which stands for the National Marine Electronics Association, established the NMEA 0183[15] standard in the early 1980s, which defines how GPS data are structured in a serial data stream. The IRMA MC, when queried for the current GPS time, or commanded to synchronize its on-board real time clock (RTC), will eavesdrop on the input serial line (serial port C) until a string terminated with a carriage return-linefeed (CR-LF) is encountered. Once date-time information is extracted from the raw NMEA string, the IRMA MC increments the current time by one second and waits for the next GPS time marker, upon which it immediately sets its RTC.

The GPS board requires an external antenna in order to receive GPS signals. A compact antenna is attached to the outside of the IRMA receiver compartment. The GPS board is sensitive to signal quality, which when degraded, will emit in the serial stream a flag which indicates the current data are invalid. At the same time, the GPS will substitute the current time date calculated by its own RTC.

### 2.4.5 Notch and Bandpass Filters

A 60 notch filter is used to remove 60 Hz power line noise from the IR signal. This frequency is switched to 50 Hz for deployment at sites where mains operates at 50 Hz. A 455 Hz bandpass filter can be enabled to reject all frequencies above and below the 455 Hz chopper wheel frequency. The 60 Hz notch filter is mapped line 5 on parallel port E,

and the 455 Hz bandpass filter mapped line 7 on parallel port E. Both filters are enabled by setting their respective lines, while disabling the filters requires clearing their respective lines.

## 2.5 IRMA Alt-Az Controller

The Alt-Az controller (AAC) is responsible for pointing the Alt-Az mount that holds the IRMA unit. As such, a master-slave relationship exists between the MC and AAC because the AAC does not initiate any actions or produce any data unless commanded to do so by the MC. Command packets sent over the 19.2 kbit/s serial connection between the MC and AAC instruct the AAC to move the axes to a specified azimuth and elevation, return system status, or return the current X and Y axis positions. The AAC software allows for position queries while an Alt-Az movement is taking place. Motor control consumes the majority of DIO lines on the AAC Rabbit. As shown in the pin map shown in figure 2.17, all lines except for the MC-AAC serial communication channel are DIO lines.

### 2.5.1 Rabbit Semiconductor RCM2010 Controller Module

The RCM2010 controller module is the control computer that handles motion control and communication for the AAC. Based on the Rabbit 2000 CPU, the RCM2010 has less memory than the MC's RCM2100 controller module and lacks an Ethernet controller chip. It is, however, smaller than the MC's RCM2100 core module. Specifications of the

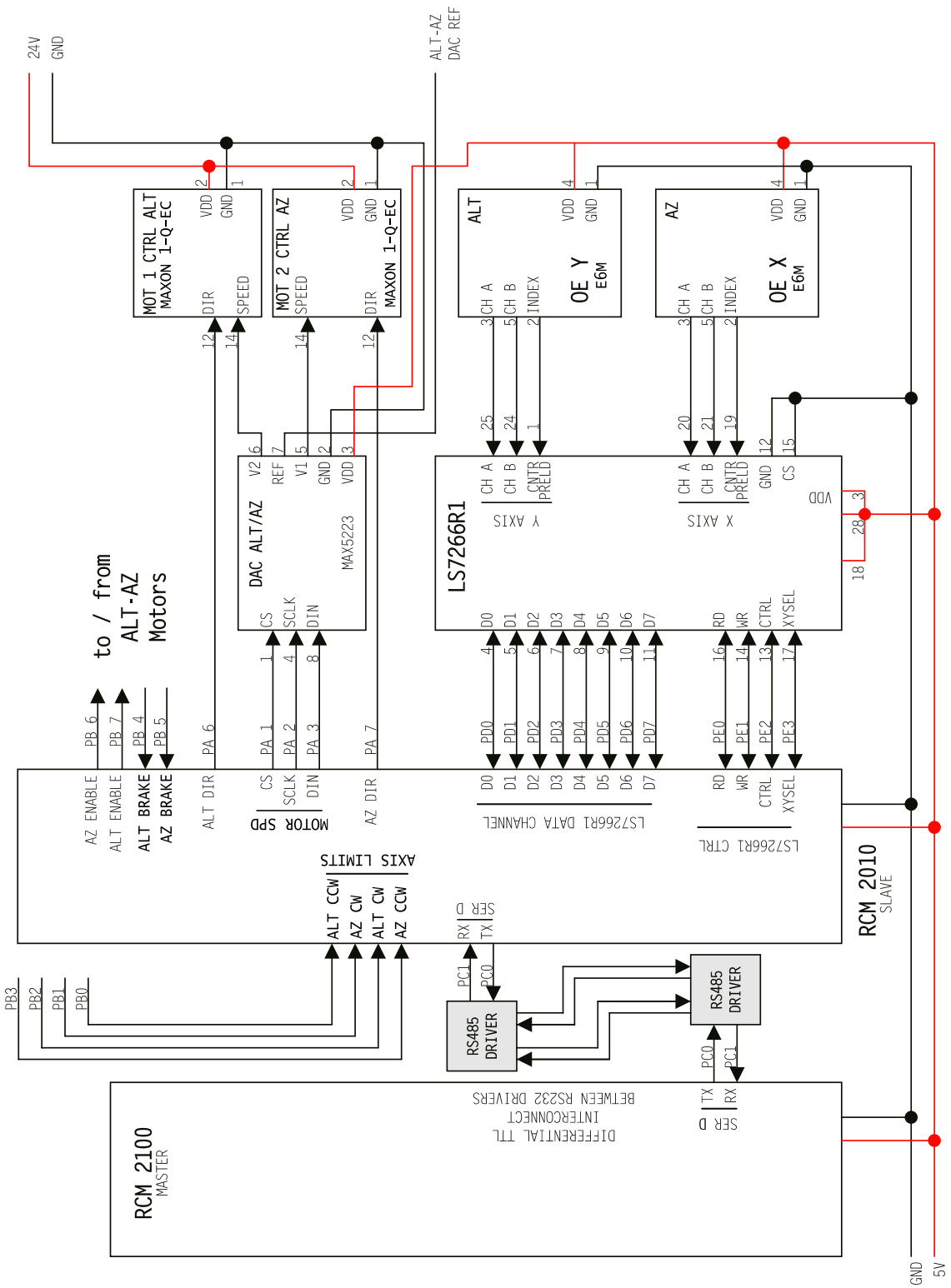


Figure 2.16: IRMA Alt-Az hardware block diagram with pin mappings.

| | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|
| Parallel A | Motor Direction AZ | Motor Direction ALT | Brake AZ | Brake ALT | Motor DIN (data in) | Motor SCLK (serial clock) | Motor CS (chip select) | Not used |
| Parallel B | AZ Motor Enable | ALT Motor Enable | Not used | Not used | AZ Limit B | ALT Limit B | AZ Limit A | ALT Limit A |
| Parallel C | Not used | Not used | Not used | Not used | Not used | Not used | Master Controller / Alt-Az RX | Master Controller / Alt-Az TX |
| Parallel D | ← LS7266 Optical Encoder Data Bus (read/write) → | | | | | | | |
| Parallel E | Not used | Not used | Not used | Not used | OE X/Y Axis Select | OE CTRL | OE WR | OE RD |

Figure 2.17: IRMA Alt-Az controller pin mapping. Blue boxes refer to output lines, pink boxes refer to input lines, and white boxes represent bidirectional lines.

RCM2010 are listed in table 2.3.

### 2.5.2 Motion Control

#### Maxon Motor Controllers

Alt-Az articulation is powered by two Maxon EC167129 low-noise 50W brushless DC motors, each coupled with a Maxon 1QEC50V[36] digital motor control unit. Motor speed is controlled by applying a DC voltage to the speed input of each Maxon motor controller. A Maxim 5223 8-bit 2-channel serial DAC allows the IRMA AAC software to adjust the speed of both axes with 256 levels of voltage control. The motor controller is

| Feature | RCM2010 |
|---|---|
| Microprocessor | 25.8 MHz Rabbit 2000 |
| Memory: Flash | 256 KB |
| Memory: SRAM | 128 KB |
| Serial | 4 channels, max 115 kbps (async) |
| DIO | 40 TTL lines |
| Real Time Clock | yes |
| Timers | Five 8-bit times, one 10-bit timer |
| Connectors | Two 2x20 pin, 2mm IDC headers |
| Power | 5V +/- 0.25V, 130 mA |
| Dimensions | 58 x 48mm x 14mm |

Table 2.3: Rabbit 2010 Core Module Specifications.

configured to accept 0 to 2.5 V input voltage, which drives the motors from 500 to 12,500 RPM respectively. AAC software limits motor speed to 8000 RPM, which is the maximum rotational speed that the gear box should be driven, as stated by the manufacturer. Axis rotation is geared down substantially by a 1621:1 azimuth gear head and 1621:1 altitude gear head. An additional 8:1 gear reduction is provided by belts connecting the motors to their respective axes. During development it was found that without applied voltage, the Maxon motors still rotated. Therefore, a braking system was required to hold the axes stationary when not being rotated. Braking is applied by setting bits 4 and 5 (for altitude and azimuth respectively) on parallel port A.

Azimuth and altitude motor controller enable lines are mapped to output lines 7 and 6 on parallel port B. Motor controllers are enabled by setting these lines, while clearing these lines disables the controllers. Azimuth and altitude motor direction is controlled by DIO output lines 7 and 6 respectively on parallel port A. Setting either of these two lines sets the corresponding axis into clockwise (CW) rotation, while clearing puts the corresponding

axis into counterclockwise (CCW) rotation.

The azimuth axis is capable of rotating aproximately 370 degrees. The altitude axis can rotate approximately 198 degrees. To prevent rotation beyond these limits and prevent the cabling connecting the articulating parts from being damaged, optical limit switches, similar to the ones used in the blackbody shutter, are found at the maximum CCW and CW rotational limits. Optical sensors automatically disable the motors and set one of the two limit lines when they are interrupted by a metal tab attached to the rotating housing. Limit detection is independent of software in order to eliminate the risk of runaway axis movement damaging the mount if the software were to fail. Altitude CW and CCW limits are respectively mapped to input DIO lines 2 and 0 on parallel port B. Azimuth CW and CCW limits are respectively mapped to input DIO lines 1 and 3 on parallel port B. When a limit line is set, a limit has been encountered, while when a limit line is clear, the axis angle is within safe rotational limits.

#### Maxim MAX5223 Serial 8-Bit DAC

Axis motor speed is controlled with an 8-bit 2-channel Maxim 5223[34] serial digital to analog converter (DAC). The 5223 has a 3-wire serial communications interface involving a chip select line (CS), a serial clock line (SCLK) and a data input line (DIN). Voltage is individually adjustable on each of the 5223's two analog outputs, A and B. Voltage can be set between 0 V to full scale (the input reference voltage) in 256 equal steps. Analog output channel A is mapped to the azimuth motor controller, while analog output B is mapped to the elevation motor controller.

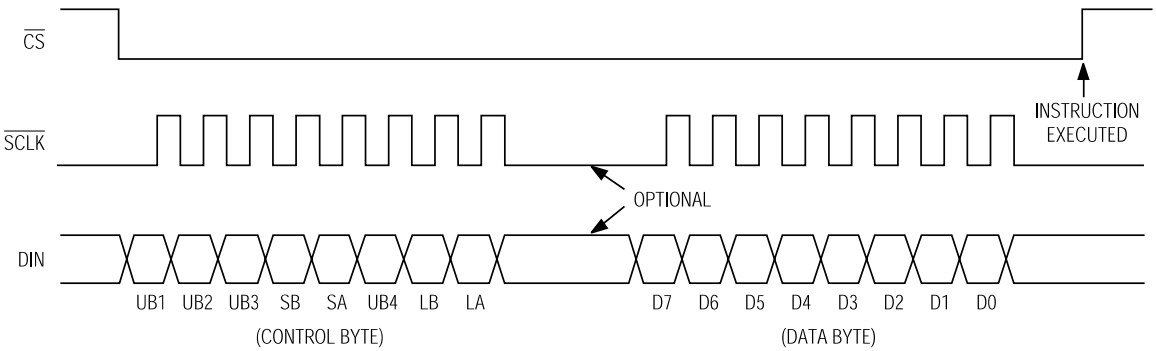


Figure 2.18: Maxim MAX5223 Serial 8-Bit DAC 3-Wire Interface Timing Diagram. The SCLK signal can be modulated at a maximum rate of 25 MHz (40 ns). Data should be placed on the DIN pin at least 20 ns before SCLK makes a low to high transition, and be held for at least 20 ns[34].

Commanding the 5223 involves clearing the CS line, writing a 16-bit word into the 5223's internal shift register, and setting the CS line, which refreshes (changes) the analog outputs. This sequence is shown in the 5223's timing diagram in figure 2.18. The SCLK line controls the process of writing data to the 5223. Data bits are read into the shift register on the rising edge of each SCLK pulse. All the communication lines are mapped to Rabbit parallel port A: CS (active low) is assigned to line 1, SCLK to line 2 and DIN to line 3.

The command word, as seen in table 2.4, is divided into 2 parts: the leading byte, or control byte, contains 8 configurable bits, where setting bits 7 (LA) and 6 (LB) loads a new value into DAC register A and B respectively with the value contained in the trailing data byte. The data byte can contain an unsigned 8-bit value that defines the proportion of output voltage to the DAC's reference voltage. Table 2.4 shows an example command word needed to output 1 V on DAC output, based on a 2.5 V reference voltage (the reference used to drive the Alt-Az axes). The proportion of full scale voltage corresponding to 1 V

can be converted to an 8-bit value using equation 2.5.

$$DAC_{input} = \frac{V_{out}}{V_{ref}} 255 \tag{2.5}$$

This value, which works out to decimal 102, appears in the data field of the DAC command word. Bit field D0 is the LSB, while bit field UB1 is the MSB. Bits are transmitted from right to left, starting with UB1 and ending with D0. Functionality of the 5223 on the Rabbit RCM2010 (AAC) is contained in the custom-written Dynamic C library oe3.lib.

| Data Byte | | | | | | | | Control Byte | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D0 | D1 | D2 | D3 | D4 | D5 | D5 | D7 | LA | LB | UB4 | SA | SB | UB3 | UB2 | UB1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 2.4: Maxim MAX5223 DAC serial command word format.

**Optical Encoder**

Axis positions are measured via two US Digital[62] E6M optical encoders and one US Digital LS7266R1[61] encoder to microprocessor interface chip. The two optical encoders each employ 2096-line per revolution optical encoder wheels. When operating in quadrature mode, the LS7266R1 interpolates the raw encoder signals (sine and cosine outputs) to obtain 8192 lines of resolution per revolution, or 2 arc seconds per encoder step[54]. Quadrature mode is based on the optical encoder generating sine and cosine signals (each generating one cycle per tick), whereby the decoder chip counts the zero crossings of both the signals. Given that the crossings occur at 90, 180, 270 and 0 degrees, four counts per tick can be detected. The LS7266R1 detects and counts ticks from the encoders (in either mode), where the count is relative to a fixed mark.

The LS7266 optical encoder chip interfaces to the AC Rabbit over 12 DIO lines. Data to and from the 7266 is carried over an 8-bit bidirectional data bus, mapped to pins 7 through 0 on parallel port D. Port D is bidirectional, thus software must set the appropriate data direction depending on whether data are being written or read. Disregarding the chip select line, which is permanently tied high, there are four lines used for controlling the LS7266: the control/data, read, write and X/Y axis select. Read and write (LS7266 pins 16 and 14 respectively) are active low, and are used for enabling reading or writing to the chip. The control/data line (LS7266 pin 13) selects whether data registers (low) or control registers (high) are selected. Similarly, the X/Y axis line (LS7266 pin 13) selects whether the X axis counter (low) or Y axis counter (high) is selected.

The filter clock (LS7266 pin 2) must be fed with a frequency between 10 KHz and 35 MHz to operate in quadrature mode. Additionally, analog inputs A and B for each axis channel must be fed with corresponding A and B signals from the respective optical encoders. The A/B inputs for both axes are enabled by setting the A/B input enable lines (LS7266 pins 18 and 1) high. These lines are permanently tied high on the AAC main board. The filter clock is fed with a 10 MHz signal. A clock signal is not required if the LS7266R1 is operated in non-quadrature mode, which provides 2048 lines of resolution per rotation.

Each optical encoder has a unique index mark on their encoder wheels. An index strobe signal is emitted from each optical encoder when the index mark is encoded. The index strobe from the X and Y axes are fed into pins 19 and 1 respectively. The LS7266R1's 24 bit counter can either be reset or set with a preset value when strobed with the index

signal. The index mark is provided as fiducial marker, but the IRMA Alt-Az uses the optical limits on both axes as references instead.

The LS7266R1's register structure provides some insight on how the chip is controlled. All commands involve communication over the 8-bit data bus to either the control registers: RLD, CMR, IOR, IDR and FLAG, or the data registers: 3-byte preset register and 3-byte output latch. The write-only 3-byte preset register is selected when the control/data line is low. The write only control registers (RLD, CMR, IOR and IDR) are selected by setting line control/data line and placing a 2-bit binary value on data bits 6 and 5. Codes 00, 01, 10 and 11 select RLD, CMR, IOR and IDR respectively for writing. The X-axis status FLAG register is selected by clearing the read line (and setting write), clearing the X/Y axis select line, and setting the ctrl/data line. The Y-axis status FLAG register is performed similarly, except the X/Y axis select line is set. Reading a byte from the 8-bit data bus will return the contents of the FLAG register. The 3-byte output latch is selected by clearing the read line (and setting the write line), and clearing the control/data line. Detailed instructions on register selection using the control lines is found in the LS7266R1 data sheet's chip access table[61].

Reading or writing a byte is performed by strobing the read or write line, both of which are active low. When the read or write line performs a low to high voltage transition, a read or write byte transfer occurs. Given that the counter output is 3 bytes wide, the data must be read out a single byte at a time, the LSB being read first. An internal byte pointer is automatically incremented after a read is performed. The byte pointer is reset by setting bit 1 of the RLD register, and must be performed after reading the counter. This

is the most common command sent by the AAC to the LS7766R1. Reading axis position, another common function, involves commanding the LS7266R1 to transfer the contents of the 24-bit counter to the 3-byte output latch. This is performed by setting bit 4 in the RLD configuration register. The LS7266R1's functionality is encapsulated in the Dynamic C library `oe3.lib`, which wraps many of complex sequences needed to control the optical encoder chip in easy-to-use functions.

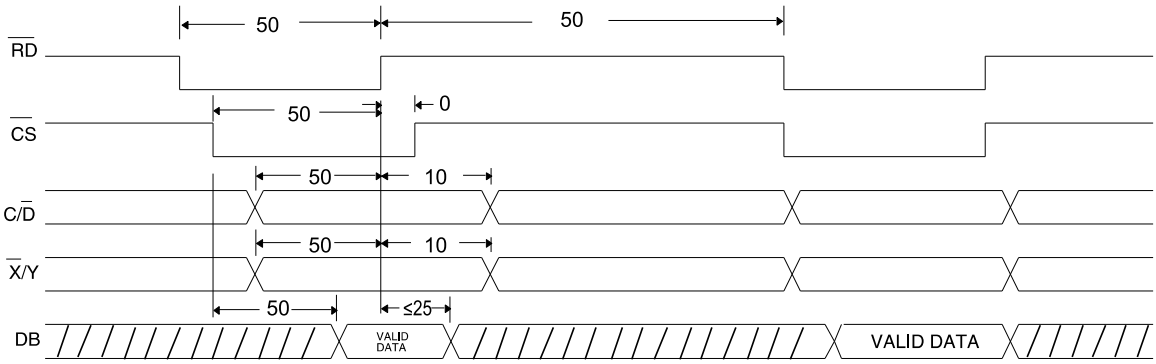


Figure 2.19: US Digital LS7266R1 read cycle timing (in ns)[61].

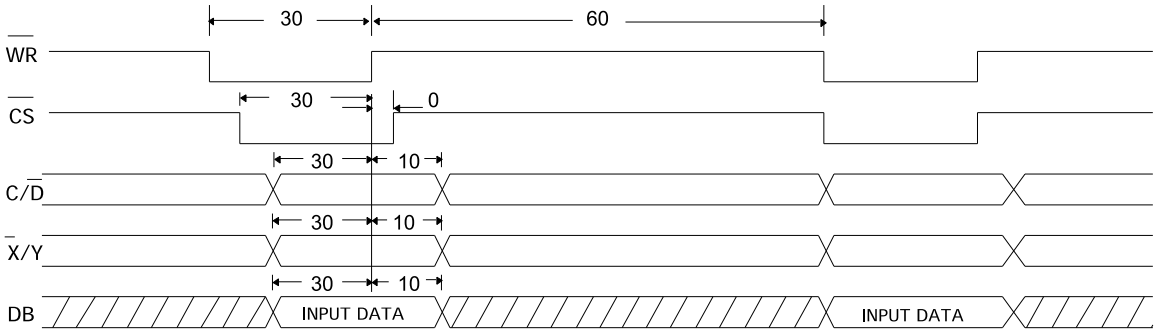


Figure 2.20: US Digital LS7266R1 write cycle timing (in ns)[61].

## 2.6 Conclusion

The details surrounding IRMA's hardware interfacing are complex, in particular, IRMA's digital I/O and serial connections, which are critical to IRMA's operation. The chop interrupt line is perhaps the most important, for without it, IRMA's ability to collect data would come to a halt. This was the cause of failure in one of the IRMA units undergoing testing at the Smithsonian Submillimeter Array on Mauna Kea in the fall of 2004. Equally critical is IRMA's internal Ethernet communication lines, which failed (for yet to be determined reasons) during testing at the Gemini South observatory in Chile, in February 2005. The challenge to the programmer interfacing these devices to a controller is to write software to deal with potential hardware failures gracefully, and provide feedback to the operator when a hardware failure has occurred.

# Chapter 3

# IRMA Software Structure

This chapter will discuss the structure of the IRMA software, as opposed to the function of the IRMA software, which is touched upon in chapter 4. This discussion will look at the relationships among the CP, MC and AAC, the multi-tasking task structure of each of the programs, the communication structures and protocols used, and the structure of the input and output files consumed and produced.

## 3.1 IRMA Software Architecture

IRMA's software structure can be described as distributed, modular, multi-tasking and real-time. It is distributed such that its functions are divided among three programs hosted on three separate computers, all of which communicate asynchronously with one another. It is modular by the fact that its programs are structured using top-down refinement, where the overall problem is subdivided into smaller pieces called functions. Finally, IRMA software is multi-tasking in that it performs certain *tasks* in parallel through the use

of a real-time multi-tasking kernel (RTK), MicroC/OS-II[27], running in the background. Certain actions performed by the MC and AAC software are designed to happen at specific intervals, always occurring in a timely, predictable manner regardless of the workload the system might be under. This is the definition of a hard real time system[27], which the MicroC/OS-II RTK provides. As such, the MC and AAC can be considered real-time software.

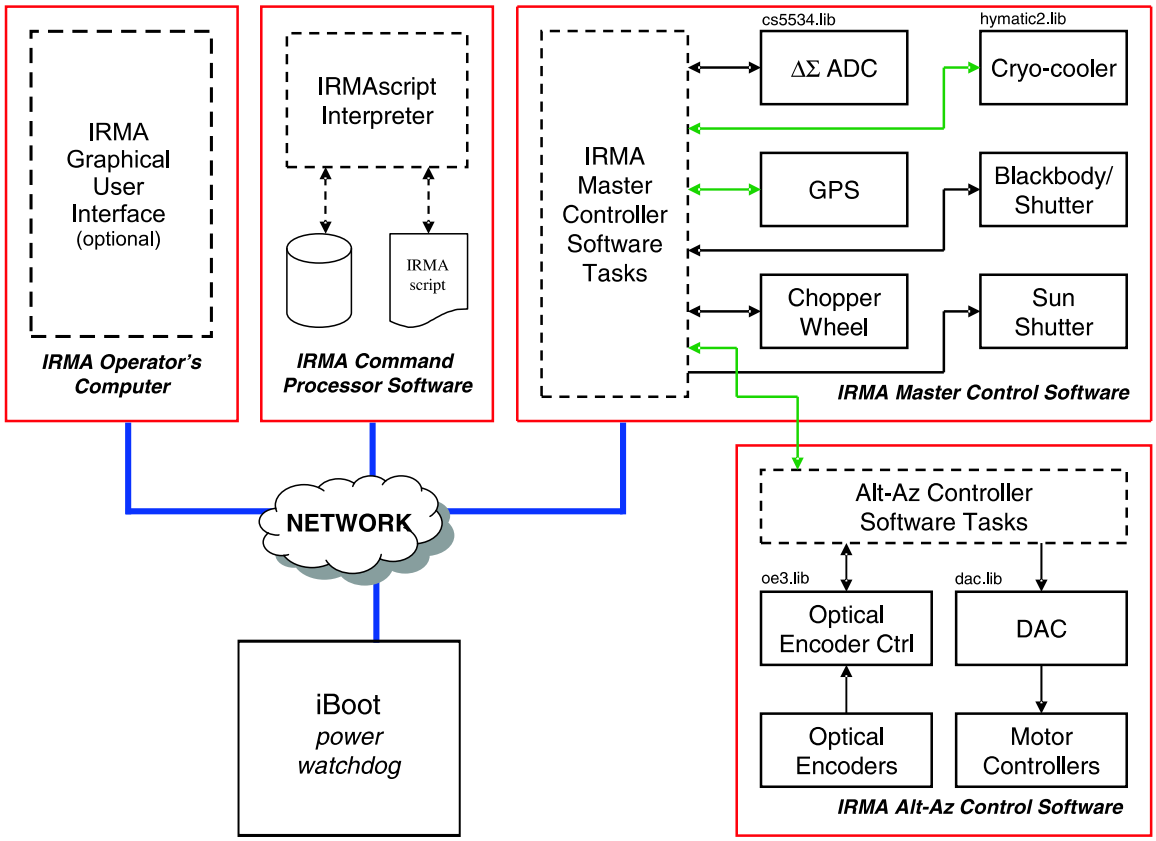


Figure 3.1: IRMA control software software structure shows typically shows four major software components (shown in red): the graphical user interface (GUI), command processor (CP), master controller (MC) and Alt-Az Controller (AAC).

The top level view of the IRMA control software, appearing in figure 3.1, shows

each of the independently executing software entities outlined in red. Working clockwise from the top left hand corner of the diagram, the modules include the optional IRMA graphical user interface (GUI), the command processor (CP), the master controller software (MC), and the Alt-Az controller (AAC), shown in the lower right corner. The iBoot power watchdog unit, outlined in black, is an networked power controller that cycles main power to the IRMA MC/AAC if IRMA fails to transmit a heartbeat network packet to the iBoot within some prescribed period of time. The user has the option of running IRMA's GUI on the CP or on a remote machine. If the operator chooses to run IRMA without a GUI, he or she can simply log into the CP via SSH and run the `irmaExec` interpreter from the command line.

The operator is provided with two means to control IRMA: via the command line, where the `irmaExec.pl` interpreter is called directly, or by using the GUI interface. The GUI does not execute on the operator's computer, but rather on the CP machine. Figure 3.1 shows the GUI running in parallel with the IRMA script interpreter. However, running the GUI is optional; IRMA can be operated via the command line alone. Under X, the UNIX/Linux graphic display server, the IRMA GUI is transmitted (or exported) to the operator's machine. This method of graphics display performs well where a high-bandwidth network connection is in place, such as in a LAN setting. In situations where network bandwidth is limited, using the IRMA GUI should be avoided because the program responsiveness drops dramatically, making the program almost impossible to use.

Each software entity executes on its own hardware. The CP and GUI run on a PC running Linux, and are written in Perl and Tk. Tk is a platform-independent GUI toolkit,

which can be called from Perl programs. Both Perl and Tk are available for Windows and for nearly every UNIX (or UNIX-derived) operating system. Besides Linux, the IRMAscript interpreter has been successfully tested on a Macintosh system running OS 10.2.8. The MC runs on the RCM2100 microcontroller module attached to the IRMA motherboard, and the AAC executes on the RCM 2010 microcontroller module attached to the Alt-Az motherboard. The CP software exists as a Perl executable called `irmaExec.pl`. The MC and AAC software exist as bootable software images that reside in their respective Rabbit microcontroller flash memories.

An Ethernet local area network (LAN) forms the communication link between the CP and MC, as well as between the operator and the CP (or GUI). Network links are depicted as light blue lines in the diagram. A serial link, drawn in green, connects the MC to the AAC. Communication among the CP, MC and AAC is asynchronous. No module knows when the other will initiate a communication with the other. What is certain, however, is who initiates the conversation. A master-slave hierarchy exists among the modules. The GUI communicates with the CP via OS system calls. The GUI and the CP can be considered to be a single entity in the context of this discussion, thus it serves as the source of all commands to the MC and the AAC.

The CP always initiates communication with the MC and AAC, as prescribed by the currently executing IRMAscript. The MC addresses the AAC only when requested by the CP. Data always flows back to the CP. Data originating on the AAC is passed back to the CP via the MC, while data originating on the MC is passed back to the CP directly. Consequently, commands which query the AAC have longer latency times than queries to

the MC. This is because AAC-bound commands must make two hops to their destination, in addition to the fact that the MC-AAC serial communications link is slower than the Ethernet link connecting the MC and CP. Alt-Az status commands, such as `ALTAZ READ POSITION`, typically require 100 ms to execute.

The software that drives the MC and Alt-Az is constructed out of multiple real-time tasks. The CP software runs within a single task except when it is in data collection mode, where it forks a separate task dedicated to receiving scan telemetry from the MC. Tasks are independent software frames of reference which run in parallel with one another. They can be thought of as mini-programs which execute independently, without affecting one another. Tasks are time-multiplexed with the CPU in order to create the illusion that each task is being simultaneously executed. In reality, each task is alloted a short period of exclusive access to the CPU. With the IRMA MC and AAC software, all tasks are not equally served, but rather, are serviced according to their priority, and whether they are waiting on an event to happen in another process. The black and green arrows appearing the MC and AAC represent DIO and serial connections respectively. The arrows do not depict individual lines, but rather, generalized data connections, and their directions. Double sided arrows show bidirectional data channels, while single sided arrows show unidirectional channels. A comprehensive description of IRMA's DIO and serial connections is found in chapter 2 dealing with IRMA's hardware.

### 3.1.1 IRMA's Languages of Implementation

As previously mentioned, the IRMA software is designed using a modular, top-down refinement methodology. This is markedly different from object-oriented (OO) design,

which requires an OO language like C++ or Java, whereby the overall problem is decomposed using OO design techniques such as encapsulation, inheritance, and polymorphism. Encapsulation associates data with the functions that manipulate them into structures called objects, and is a form of data hiding. Inheritance promotes software re-usability by allowing new objects (or classes in their non-instantiated form) to be formed from existing objects in addition to new code. Polymorphism enables objects to accept a variety of data input as opposed to writing separate functions for every expected type of input, as is required by C[11].

Dynamic C, the proprietary C compiler produced by Z-World allows the software designer to further organize his or her program's structure using custom libraries in addition to functions. Libraries allow the designer to group similar functions into separate files, in order to prevent the main program file from becoming a long unmanageable list of functions. Dynamic C's inclusion mechanism differs from standard ANSI C in that header files (dot .h files) are replaced by dot .lib files, and the `#use` directive is used in place of of `#include` directive.

Libraries specific to each of IRMA's peripheral hardware components were developed. They appear as solid boxes inside each of the software boxes, as shown in figure 3.1. Additional libraries were developed to handle specific problems, such as CRC checksum calculations. Additional libraries, in particular libraries for MicroC/OS, TCP/IP, and GPS data string parsing are provided with the Dynamic C development software. The list of custom libraries appear in table 3.1.

| **Library** | **Target** |
|---|---|
| OE3.LIB | US Digital LS7266 R1 Optical Encoder Controller |
| DAC.LIB | Maxim MAX5223 2-channel 8-bit serial DAC |
| HYMATIC2.LIB | Hymatic Cryocooler |
| CRC.LIB | CCIT-CRC (16-bit CRC algorithm) |
| CS5534ADC.LIB | Cirrus CS5534 Delta Sigma ADC |

Table 3.1: Custom libraries used in IRMA MC and AAC.

The difficulty with this approach is that it is easy to get lost in the sea of functions and lose sight of the whole, even when libraries are included. It is not always easy to see the data relations among the logical divisions within the software, nor is it possible to easily distinguish library functions from functions local to the given program file. This is one area where the object-oriented approach would be advantageous. For example, when a request to perform a scan is received by the MC's network communication module, it must pass the scan parameters, called a *job*, to the software module responsible for performing data collection. The process to get the job from the network communications module to the data collection module involves a long chain of function calls. In addition, there is the tendency for the number of constant definitions and global variables to mushroom. Global variables are the primary means to pass data between software tasks in the IRMA software.

The command processor software, responsible for interpreting IRMAscript source code files into IRMA network command packets, is written in Perl, a popular cross-platform programming language that is feature-rich, easy to write, and easy to extend. Being an interpreted language, Perl is a capable rapid development tool, as no compilation and linking is required. Considerable computational overhead is brought to bear on the processor when executing a Perl-based application, especially those that call a large number of Perl modules

(which is the case with the command processor software). Executing the CP software on a 333 MHz PC requires 10 - 15 seconds for the Perl interpreter to compile the CP software into Perl byte-code. Even though Perl is an interpreted language, like other modern interpreted languages the source code is initially compiled into a simpler, machine-code-like statements called bytecodes, which can be efficiently and quickly interpreted by a virtual machine. The virtual machine (VM) permits the language to be platform independent. Java also uses a VM to execute bytecode.

Perl is resource hungry in terms of memory and CPU cycles, making it unsuitable for hosting on an embedded processor such as the Rabbit. The reasons for choosing Perl over other compiled languages is twofold. First, it offers powerful regular expression processing capabilities, and second, several people in the IRMA research group have experience with Perl programming. A Regular expression is a language description mechanism allowing the precise definition of patterns of symbols (a string) by means of another string, defined by a set of syntax rules[2]. Regular expression matching, or pattern matching, is is the technique used to convert IRMAscript statements into equivalent 3-tuple command codes, which the MC understands.

## 3.2 IRMA multi-tasking Structure

### 3.2.1 Event Driven Programs

The flow of control within the IRMA MC and AAC software is event driven. Event driven programs can be visualized as a big loop, where the program blocks (or waits) at the top of the loop, waiting for input. When input arrives, the program determines what has

to be done from the message, performs the appropriate actions, and returns to the top of the loop to wait for a new request. This describes the general operation of a generic event-driven program, which includes most GUI-based user-driven programs[42]. Rather than user input, such as mouse clicks or key presses, IRMA responds to binary command packets arriving over the network. In essence, this description accurately describes the control flow of IRMA's MC and AAC software.

### 3.2.2 Multiprogramming and Real Time

Parallel execution of tasks (multi-tasking, or multiprogramming) combined with real-time performance (adherence to deadlines) is required by IRMA's control software. For example, when the Alt-Az mount performs a servo-controlled movement concurrently with the serial communications task that continuously monitors the serial port for commands, the motion control task relies on a timing task to update servo loop calculations every 100 ms. Moreover, the servo loop must be updated exactly at this rate in order for the servo control algorithm to function correctly. The ability to meet deadlines in a timely manner within a multi-tasking environment is the defining attribute of real-time programming[27].

All modern operating systems attempt to achieve some sense of real-time performance. The Linux 2.0 kernel uses two separate scheduling schemes for non-real-time and soft-real-time performance. Hard real-time systems guarantee that critical tasks will complete on time, and delays have fixed bounds. MicroC/OS-II fits into this category. Soft real-time systems, such as Windows NT or the Linux kernel, give critical tasks priority over tasks of lesser importance, but do not guarantee that operations will complete within fixed deadlines, nor do delays have fixed bounds. This is because real-time systems must know in

advance the durations specific operations (such as I/O), and this is impossible to do with systems that use virtual (disk-based) memory or secondary storage[56].

The Linux 2.0 kernel uses scheduling classes: time-sharing scheduling to share the CPU among many tasks (or processes) equitably, where real-time performance is not important, and soft real-time scheduling to implement near-real-time performance, which is necessary for applications such as 3-D graphics or video. The time sharing algorithm uses a prioritized credit based algorithm, where each process in the ready-queue (a queue within the scheduler that contains a list of ready-to-run processes) is assigned a number of scheduling credits, which effectively defines its priority. The scheduler selects the task with the highest number of credits from the ready queue and runs it for a predetermined time, called a time quantum. In Linux, the time quantum is implemented by decrementing the task's credits upon every CPU clock tick. When the running task's credits are exhausted to zero, the scheduler suspends the running task, and runs the next highest priority task. The act of preempting a task and selecting another task to run is called a context switch. In most operating systems, context switches occur when the running task blocks on I/O, which often involves waiting for a key press or data block transfer to complete, or when the task has run its course within its time quantum, and is preempted by the scheduler.

The Linux real-time scheduler operates similarly in that it always selects the task with highest priority from a circular task queue, or in the case of multiple equal priority tasks, selects the task that first entered the queue, called first-in-first-out (FIFO) scheduling. Again, the processes are alloted a time quantum in which they have full access to the CPU. When the time quantum is up, the scheduler suspends the running task, and selects the

next task appearing in the circular ready queue, without regard to task priority. This type of scheduling is called Round-Robin, and guarantees that each process gets $\frac{1}{n}$ units of the available CPU time, where $n$ is the number of processes in the circular ready queue[56].

MicroC/OS-II uses a priority-based scheduler. No time quantum is assigned to tasks. Rather, the scheduler switches tasks when the running task blocks on an event, such as a timer, semaphore or mutex, or when a previously-suspended higher-priority task is ready to run. A semaphore[56] is a synchronization and communication mechanism that is often used to restrict access to a resource shared by two or more processes, thus preventing multiple processes from simultaneously using a single resource. Fundamentally, semaphores are special variables that can be atomically set (i.e., the process of modifying the semaphore cannot be interrupted until completed) to one of two states: wait or signal. When a process wants to use a shared resource, such as the serial port, it accesses semaphore using the wait operation. If the resource is not being used by another process, the semaphore is decremented, and the process proceeds to use the resource. When the process has finished using the resource, it increments the semaphore, signaling to the other processes that the resource is again available. If the semaphore is decremented to 0, the resource is made unavailable. Any process that reads the semaphore is suspended by the OS and waits until the semaphore is incremented by the process using the shared resource. The depth of the semaphore determines how many processes can concurrently access the resource. For example, a semaphore of depth 3 can allow up to three processes to share the resource, while a semaphore with a depth of 1, also known as a binary semaphore or mutex, only allows a single process to access the resource. In IRMA, the MC-AAC serial communication

channel and the CP-MC network communication link are protected by mutexes, since these channels can only accommodate a single user at a time.

Round-robin real-time scheduling is not supported by MicroC/OS-II because every task is required to have its own priority level, and round-robin scheduling requires that all participating tasks have equal priority. Furthermore, the MicroC/OS-II kernel does not support task preemption by means of a time quantum[27]. Dynamic C does provide a time slicing function, but it is not compatible with MicroC/OS-II.

Instead, the software developer must explicitly design the multi-tasking structure of each of the tasks by strategically assigning task priorities and placing blocking mechanisms within each task in order to put tasks into the ready queue, and make them eligible for scheduling. Blocking mechanisms include millisecond sleeps, waiting (or pending) on inter-process communication (IPC) structures such as event flags (similar to UNIX signals or Windows messages), or waiting on shared resources to become available using inter-process synchronization objects such as mutexes or semaphores[27].

## 3.3 MC and AAC Task Structure

The multi-tasking structure of IRMA's MicroC/OS-II based programs (the MC and AAC) are based on having the high-priority tasks perform their tasks in short bursts, then sleep for a defined interval or wait on an event, in order to open up slack time in which the lower-priority tasks can execute. When the lower level task completes its activities, it will sleep for a prescribed period or block on an event, allowing the next lower-priority task to execute. The order of execution continues down the priority hierarchy until all the tasks

have completed.

Task priority is assigned according to the degree to which a task can tolerate being preempted. In priority-based preemptive multi-tasking, tasks can preempt other tasks having lower priorities than themselves. For example, a task having a priority of 10 can preempt low priority tasks with priority levels greater than 10. However, this same task can in turn be preempted by higher priority tasks having priority levels less than 10. The key is to establish which tasks can be preempted and for how long, giving the most critical task that cannot tolerate preemption the highest priority. All tasks are subject to preemption if an ISR is present. This should not pose a problem, since an ISR by nature (should) execute and exit as quickly as possible.

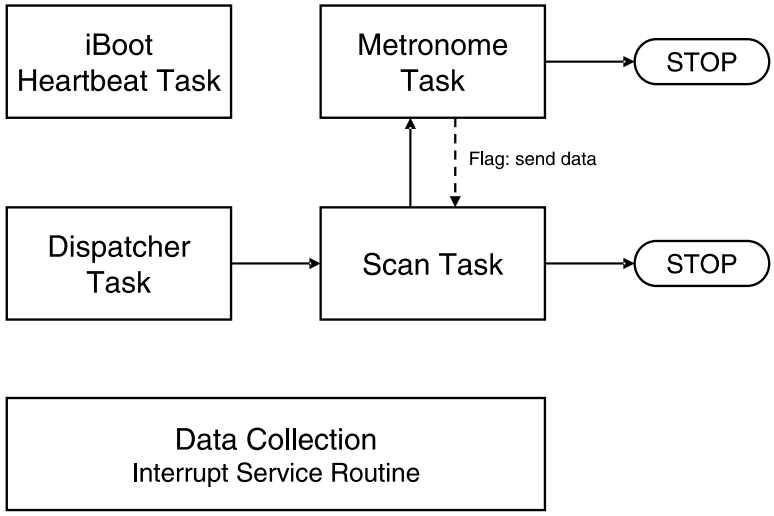


Figure 3.2: IRMA master control software: task structure during scanning.

The MC software's task structure, appearing in figure 3.2 contains two primary

tasks that are constantly running: the iBoot power supply watchdog task, and the dispatcher task. The iBoot task has the highest priority of any task running on the MC. The task broadcasts a UDP packet every 10 seconds, which is picked up the iBoot device, a power supply containing an embedded processor and network interface. The iBoot is configured to expect status packets at a predetermined interval. If the next expected packet fails to arrive within the interval, the iBoot assumes the device (the IRMA MC) has experienced software failure, and proceeds to cycle the power to the MC motherboard, forcing a hard reset. This is a critical activity that must neither be delayed nor preempted for an extended period of time, which explains the rationale for making this a high priority activity.

The dispatcher task is the most active task within the MC, and is primarily concerned with receiving commands from the CP. Since the majority of commands are classed as short-duration, meaning they take less than a second to execute, they are allowed to execute within the dispatcher task. Long duration commands such as scans, however, must be executed outside the context of the dispatcher task, because the dispatcher must return to its primary duty of listening for incoming commands. When a scan is requested, the dispatcher task forks the scan task (shown as a solid arrow in figure 3.2), then returns to wait for incoming commands. This allows short duration tasks to run concurrently with the long duration task.

The scan task is responsible for constructing data packets and sending them to the CP. It also initiates the data collection process, driven by the data collection ISR and the 450 Hz chopper wheel notch interrupt signal. The scan task starts the metronome task, and enables the ISR to trigger on the external interrupt. The metronome, whose priority is just

below the priority of the iBoot task, but greater than the scan task, counts the data points collected by the ISR, fetches the current Alt-Az coordinate from the AAC for each data point, and signals the scan task (by means of an event flag) to construct and transmit the data packet when 19 points are collected. Once the scan task has successfully transmitted the data packet, it returns to wait on the data transmit event flag, shown as a dotted arrow in figure 3.2. When data collection is terminated, the scan task and metronome task are both instructed to terminate themselves.

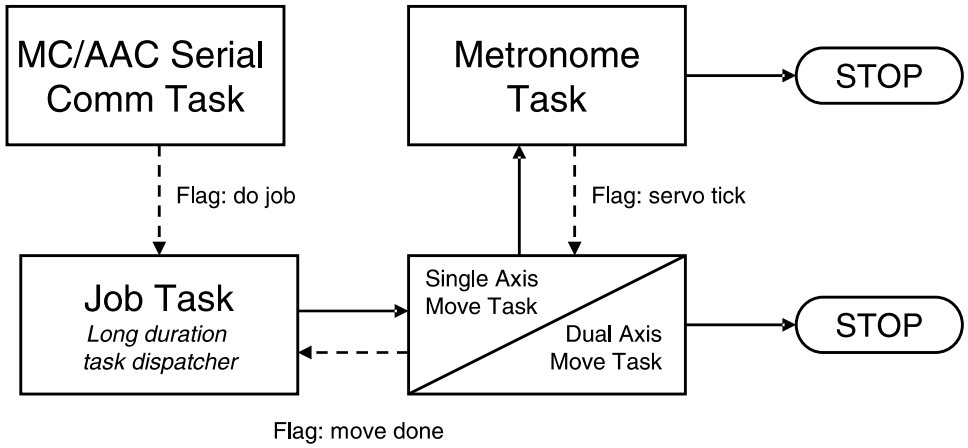


Figure 3.3: IRMA Alt-Az controller: task structure of servo movement.

The AAC's task structure, appearing in figure 3.3, parallels the MC's structure. Its serial communication task, analogous to the MC's dispatcher task, waits for commands from the MC. Most commands are short-duration, allowing them to be executed within the communication task. Axis movement tasks are classed as long-duration, thus they must be run in parallel in their own task(s) concurrently with the serial communication task. The

serial communication task signals the job task by means of an event flag to wake up and dispatch the axis movement task(s). The job task starts the single axis move task if a single axis has been specified, or the dual axis move task if both axes have been requested to move. Either movement task starts the metronome task, the highest priority task in the AAC, to control servo loop timing. A proportional-integration-derivative (PID) servo tracking loop is used by the servo move task to move the axis from the initial to the destination angle. The move task refers to either the single or dual axis move task in this discussion. The axis move task waits on a 10 Hz servo tick event flag, emitted by the metronome task, which signals the move task to update the servo loop calculations. Once the movement has completed, the move task signals the job task (using a flag event) that the axis rotation has completed. After this, the metronome and move axis task suspend themselves. The priority levels of tasks in the AAC software place highest priority on the metronome task, followed by the axis move task (single or dual), serial communications task, and the job task.

It was necessary to introduce a second mode of axis movement to handle extremely slow movement, that is, movement slower than that achievable when driving the axis motors at their minimum RPM rates. Slews have the ability to perform periodic steps over a long period of time, thus lengthen the time to rotate from the initial to destination angle. The serial communication task signals the job task to wake up and start the appropriate axes control tasks. Each axis movement is controlled in its own task: one exists for altitude movement, and another for azimuth. Both axis tasks can be run concurrently, which requires that they each have unique priority levels. Since both tasks cannot have the same priority,

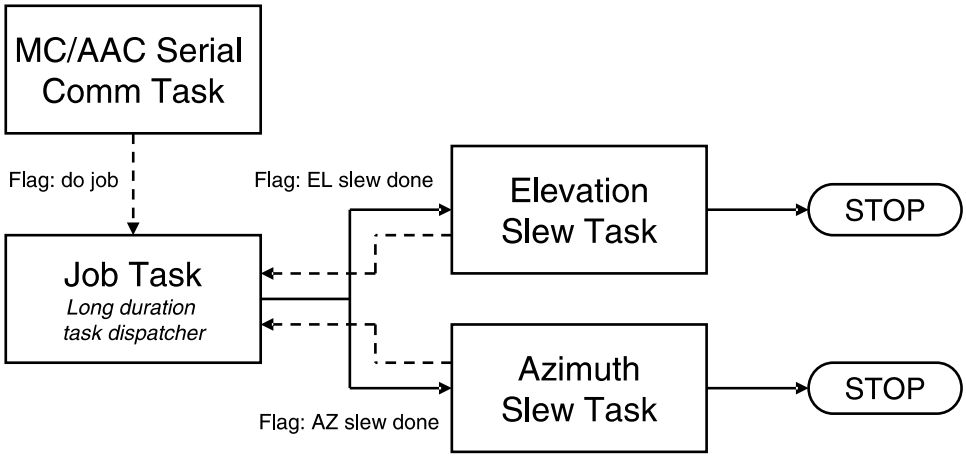


Figure 3.4: IRMA Alt-Az controller: task structure of slew (stepped) movement.

the slew elevation task has a slightly higher priority than the azimuth slew task. Given that skydip operations, which involve slewing the altitude axis, are performed more often than azimuth movements, preference was given to elevation movements. Slew tasks and the servo move tasks have priority levels that place them below the metronome task priority, but above every other AAC task.

Once one or both slew tasks have been started, the job task waits for flag events from the slew task(s). When the flag signal(s) are received, indicating either or both movements are complete and the respective slew control tasks are suspended, the job task returns to listen for new commands from the serial communications task. This process is shown in figure 3.4.

## 3.4 Data Collection Interrupt Service Routine

The MC's data collection ISR runs independently of the MC tasks, running in a level of software separate from that of the MicroC/OS-II tasks. Its behavior is determined by the chopper wheel, which generates a 90 Hz notch interrupt signal. The notch interrupt invokes the ISR, while the sampling parameters of the Cirrus CS5534 $\Delta\Sigma$ ADC, in particular, the ADC word rate, determines the rate of data collection. The duration of the ISR is determined by the duration of all the combined machine instructions making up the ISR code.

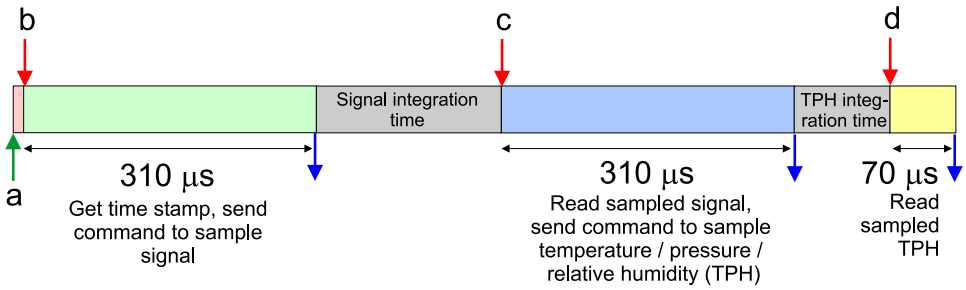


Figure 3.5: IRMA master controller: data collection ISR structure.

Figure 3.5 shows the ISR's sequence of events (not drawn to scale). The ISR is broken up into phases to prevent blocking on the relatively long $\Delta\Sigma$ integration periods, shown in gray. The CS5534 is configured to sample the IR signal at 23 noise free bits of resolution, the highest sampling resolution available on the $\Delta\Sigma$ ADC. The resulting integration time required by the $\Delta\Sigma$ is 538 ms. In contrast, around 380 $\mu$s is required to instruct the $\Delta\Sigma$ to start the A/D conversion and read the resulting sample. During the

integration periods, the MC does not halt execution, but continues normal operation.

Point A in figure 3.5 shows the beginning of the data collection cycle, when the external interrupt and chop interrupt enable are both enabled. The time lag between enabling the external interrupt and entering the ISR is shown in the pink region of the graph, and is dependent upon the rotational period of the chopper wheel, ranging between 0 and 11 $\mu$s. Point B in figure 3.5 marks the beginning of phase 1 of the ISR, where the command to start the high-resolution A/D conversion of the IR signal is strobed into the $\Delta\Sigma$. This phase begins by disabling the external interrupt (on the Rabbit) and notch interrupt enable gate. Next, the sample command is written to the ADC, and a time stamp is generated for this particular sample. Phase 1 concludes by re-enabling the external interrupt line and exits the ISR. The notch interrupt enable line is left disabled. Phase 1, which appears as the green region in the graph, requires 310 $\mu$s of execution time.

Point C in figure 3.5 marks the end of the A/D conversion, and the beginning of phase 2. This phase is triggered not by the chop interrupt, but by a signal transition on the $\Delta\Sigma$'s serial data out (SDO) line, which occurs when the A/D conversion has completed integration. Shown in blue, phase 2 begins by disabling the external interrupt, followed by strobing out a 32-bit word containing the 24-bit data word. Immediately following this, a command to sample one of ten temperature-pressure-humidity (TPH) channels is strobed into the $\Delta\Sigma$. During data collection, IR signal sampling is interleaved with sampling through the TPH channels in round-robin fashion. Phase 2 concludes by re-enabling external interrupts, but not the notch interrupt enable line, then exits the ISR. Total execution time for phase 2 is 310 $\mu$s.

Point D in figure 3.5 marks the end of TPH sample integration, and the beginning of phase 3 (shown in yellow), where the TPH sample is read from the ADC. The ADC's end-of-conversion signal, represented by a logic transition on its SDO line, triggers the third and final entry into the ISR. Upon entry, the external interrupt line is disabled, and 32 bits containing the 24-bit sample word is strobed out of the ADC. The index variable for the ISR's internal circular shared memory buffer, which stores the entire 19-sample data set, is updated. After this, the ISR exits, leaving external interrupts and the notch interrupt enable gate disabled. Both will be re-enabled by the metronome task when it wakes up. 70 $\mu$s of CPU time is consumed by phase 3.

In total, only 690 $\mu$s of time is spent executing ISR code compared to the duration of the entire data collection cycle of 583,690 $\mu$s. No more than 310 $\mu$s is spent in the ISR at any one time. This impressive performance can be attributed to the fact that the data collection ISR is written almost entirely in Rabbit 2000 assembly code.

## 3.5 Communication Packet Structure

The IRMA CP communicates with the MC by means of binary-formatted data packets, sent over a TCP (Transport Control Protocol) network connection. TCP is a connection-based protocol, analogous to a telephone system, which establishes a circuit between the two parties. TCP guarantees that the data transmitted reaches its destination (which may involve retransmission, if necessary), and that data is received in the order that was sent. As such, the TCP protocol is considered a reliable protocol.

TCP is more computationally expensive than the less reliable but more efficient

UDP, which broadcasts a simple packet containing the recipient's address and the data. The recipient may or may not receive the packet, which gets routed from host to host as it makes its way across the (inter)network to its destination. As such, its operation is similar to how letters are delivered by the postal system. TCP was chosen as the network protocol on which to base IRMA network communication because it was anticipated that IRMA's network infrastructure may be unreliable, subject to crosstalk, electrical noise and marginal cables or interconnects. This assumption proved to be correct during tests at the Gemini South observatory at Cerro Pachon, Chile, when an IRMA unit began to experience network communication failures.

IRMA network communication packets are structured binary data carried in TCP packets. The MC and CP software is responsible for constructing and dissecting IRMA network communication packets. The MC and CP's underlying Dynamic C networking library handles TCP/IP network transactions. CP-MC network communication uses binary packets rather than ASCII strings, since binary-formatted packets require less effort to parse than strings. Being that they only contain numeric codes, and all the codes have fixed lengths, a simple compact algorithm is capable of decomposing and parsing the packets. The MC is only an 8-bit microprocessor having limited memory resources, thus the developer must be mindful of efficiency. A generic network communications packet, pictured in figure 3.6, contains three primary items: a header, body, and a checksum. This packet structure is common to all network-based communications performed between the CP and MC, regardless of their function.

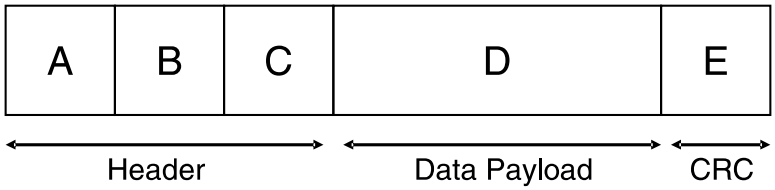


Figure 3.6: Generic IRMA network communications packet. A: Number of bytes in data payload (D). B: Packet number of the current packet group. C: Total number of packets in the current packet group. D: Data payload. E: CRC (Cyclic Redundancy Check) checksum.

IRMA network packets are aligned to 4-byte longword boundaries, making the 4-byte (32-bit) integer the the smallest data division within the packet. Headers contain three fields. The first field indicates the number of bytes (not 4-byte longwords) in the data payload. Field two indicates the identity of a packet within a block of packets, called a packet group. This labeling is necessary if a single data set is spanned across multiple IRMA network packets. Field three indicates the total number of packets within the current packet group. The fourth field contains the content of the packet. It contains its own structure depending on its type. An IRMA network packet contains at least one data item. The fifth field is the Cyclic Redundancy Check (CRC) checksum packet, used by the packet recipient to test the integrity of the packet. A CRC is a hash function which calculates a checksum word from a large block of binary data, which is appended to the end of a data packet to be transmitted. The recipient can easily recalculate the CRC value from the received data packet in order to detect transmission errors.

The 12-byte header and 4-byte CRC checksum act as a wrapper around the IRMA network communications packet, enabling the packet recipient to determine the size of the packet, identify the packet's position within a data block spanned across multiple packets, and verify that the packet contains no transmission errors. Reading packets is relatively

easy for the packet recipient: it must read 12 bytes, from which it can determine the number of successive bytes it must read from the socket. Once the recipient has read this number of bytes, it can assume it has read the entire packet, and proceed to calculate the CRC value over the entire packet excluding the final 4 bytes that contain the packet's embedded CRC. The recipient can be confident that the packet contains no communications errors if the packet's embedded checksum matches the recipient-calculated checksum.

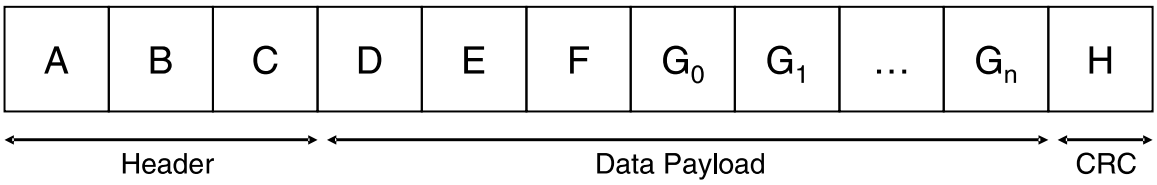


Figure 3.7: IRMA network communications command packet.

The command packet, pictured in figure 3.7, is one of the most common type of communications packet. The command packet is divided into 4 primary fields. The first field (D) contains the command code. The next two fields (E and F) contain the command modifiers associated with the given command code. IRMAscript commands are structured as a three-tuple: **[command] [modifier1] [modifier2]** in order to organize IRMA's functionality into families of commands. Zero to fifteen parameters (fields $G_0$ - $G_n$) can be associated with a command. The number of parameters associated with a particular command is fixed, thus the recipient (the IRMA MC) knows in advance how many parameters to expect and read from the network socket.

Data values being passed to or from the MC are represented as 32-bit integers.

Where floating point values must be passed, scaling is used to temporarily represent the floating point value as an integer. Pre-established scaling factors have been hard-coded into the CP and MC software for each data item requiring floating point - integer conversion.

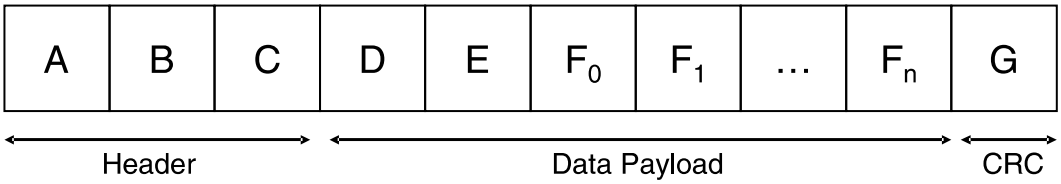


Figure 3.8: IRMA network communications data packet.

Data packets are sent by the MC to the CP when the given command is capable of producing data. Data-generating commands include status requests, such as current Alt-Az position, or scan requests, which produce scan data packets. The data packet, shown in figure 3.8, consists of a packet type flag (D), the number of data points (E), and the data values ($F_0$ - $F_n$). The packet type field is populated with value 3000, indicating that this packet is of type DATA_FIELD. A data payload produced by a scan contains 114 data points, or 19 6-tuples, each of which represents an IR signal sample along with its time stamp, current mount altitude and azimuth position, and an environmental reading (temperature-pressure-humidity). In total, this constitutes a 456 byte data payload.

## 3.6 IRMA Communication Protocols

### 3.6.1 IRMA Network Communication Handshaking Protocol

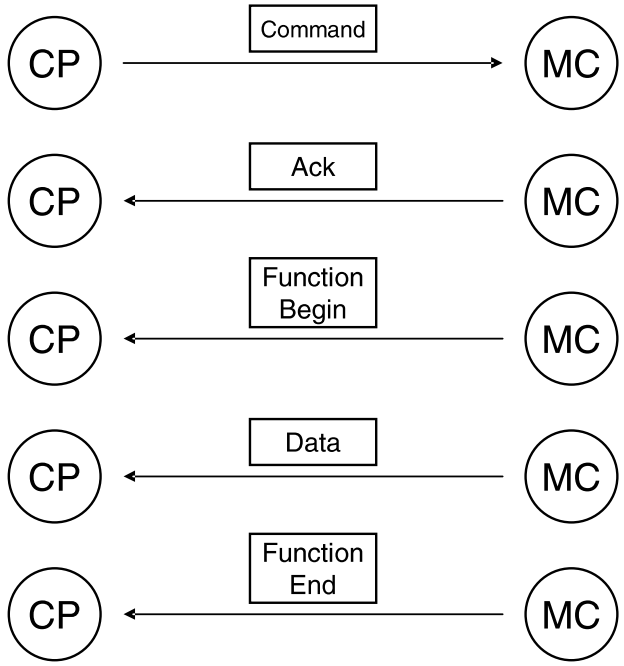


Figure 3.9: IRMA network communications handshaking sequence.

Network transactions between the CP and MC are conducted using a simple handshaking protocol closely based on the protocol used with the Herschel/SPIRE Test Facility Fourier Transform Spectrometer (TFTS)[53]. The TFTS packet communication protocol follows the European Space Agency (ESA) Packet Telecommand Standard[1]. This protocol is used for communication among electronic ground support equipment, and is similar to the protocol used by ESA to communicate with spacecraft. Prior to IRMA III, the author was involved in designing the control software for the TFTS, which was built to test the

SPIRE imaging Fourier transform spectrometer. SPIRE (Spectral and Photometric Imaging REceiver) is one of three instruments to be launched in 2007 as part of the European Space Agency's Herschel mission. The instrument will allow for high resolution imaging spectroscopy and photometry in the far infrared electromagnetic spectrum[26].

The IRMA protocol, pictured in figure 3.9 involves three steps or four steps if data (telemetry) is returned. After the command is received, the MC responds by sending an acknowledgment packet, a packet indicating the requested activity is has begun, a data packet (if applicable), and finally a packet indicating the requested activity has concluded.

The MC responds with an acknowledgment (ACK) packet whenever it receives a command packet. The ack packet, shown in figure 3.10, consists of a single 4-byte field (D) containing a code indicating whether the command was successfully received or not.

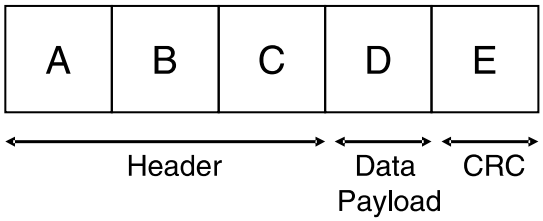


Figure 3.10: IRMA network communications acknowledgment (ACK) packet.

The ack code contains the value 1000 (**ACK_SUCCESS**) if the command packet was received without error, while a value of 1001 (**ACK_FAILURE**) indicates the packet contained an error, such as an invalid command code, or a data corruption detected by calculating its checksum. The function start packet (figure 3.11) consists of three 4-byte integer fields: an identification (ID) field (field $D_0$), a duration field (field $D_1$), and a field

indicating whether a data packet is about to follow this packet (field $D_2$).

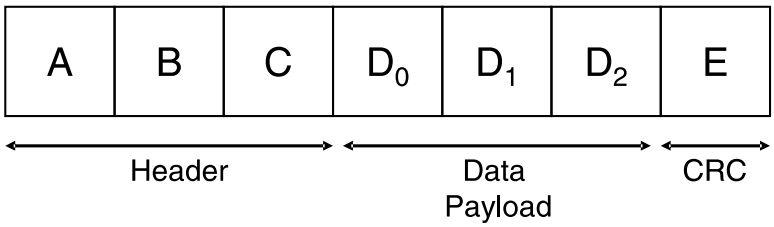


Figure 3.11: IRMA network communications function start packet.

The ID field contains the code **FUNCTION_START**, which has the value 2000. This code indicates the commencement of the requested function. The duration field can contain one of two codes: **DURATION_SHORT**, which is equal to 2010, or **DURATION_LONG**, which is equal to 2011. This code tells the CP the general duration of a given command. At the current stage of IRMA development, most commands are categorized as short duration, including functions that would be considered long. This is because long duration functions are run in parallel in their own task. As far as the CP is concerned, it only needs to know that the command it sent was received and executed. This field may be reassigned for different usage in later versions of IRMA. The **data present** field, when set to value 2020 (**DATA_PRESENT**), indicates that a data packet will follow this packet. When this field is set to 2021 (**DATA_NOT_PRESENT**), no data packet should be expected to follow.

The data packet consists of two header fields and up to 114 4-byte values. The first

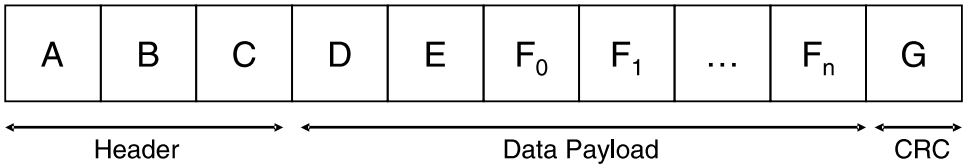


Figure 3.12: IRMA network communications data packet.

field (D) contains the value 3000, which represents the **DATA_FIELD** code. The second field (E) contains the number of data values to follow. Fields $F_0$ through $F_n$ constitute the data fields.

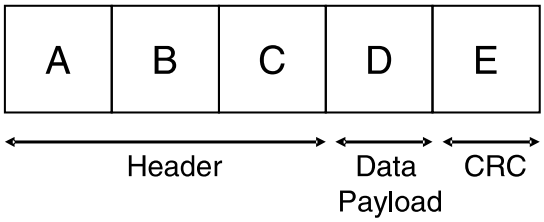


Figure 3.13: IRMA network communications function complete packet.

The function complete packet, appearing in figure 3.13 (field D), consists of a single 4-byte field that can contain one of two codes: **FUNCTION_COMPLETE_SUCCESS** (value 4000), signifying that the function completed without any errors, or **FUNCTION_COMPLETE_FAILURE** (value 4001), signifying that the function experienced errors during execution.

### 3.6.2 IRMA MC-AAC Serial Communications

Originally, it was planned that the communication protocol would be used for all communication among IRMA's three processors: the CP, MC and AAC. A high speed, 115 kbit/s 2-wire serial link was planned to link the MC to the AAC, since the AAC does not have a network interface. After extensive testing and debugging, it was found that the serial link was not acceptably reliable, being subject to communication lock-ups. The solution, as suggested on Rabbit Semiconductor's technical support bulletin board[23], was to drop the baud rate of the serial channel down to 19.2 kbit/s. This translates into roughly 2000 characters per second. Due to the considerably lowered bandwidth between the MC and AAC, the communication protocol had to be significantly simplified. The basis of the MC-AAC serial protocol is the **msCommPacketType** data structure, shown in figure 3.14: a fixed-size, 6-field data frame consisting of a command field, four data fields, and a CRC checksum field.

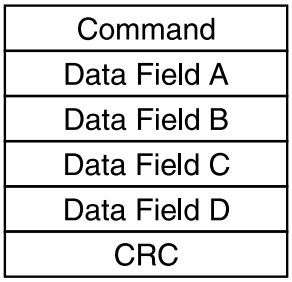


Figure 3.14: IRMA serial communications packet structure.

Serial packets differ from their network counterparts in that they are not transmit-

ted as binary data. Rather, serial packets are converted to ASCII character strings, and are encapsulated with a header and footer character. An example of a serial communications packet string appears in figure 3.15.

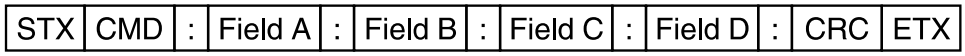


Figure 3.15: IRMA serial communications packet string.

The header character is the ASCII **STX** character (integer value 1). It is a non-printable character that is part of the base 7-bit ASCII character set. It is used to indicate to the recipient that a serial data stream immediately follows this character. The **CMD** command field tells the AAC what kind of function is being requested. These codes are part of a special AAC command set that is mapped to the IRMA Alt-AZ commands. Rather than a 3-tuple, AAC commands consist a single integer code. Table 3.2 lists each of the Alt-Az serial command codes and their aliases. The final character, an ASCII **ETX** code (integer value 2) is used to represent the end of a serial data transmission. By encapsulating a serial data stream with these two characters, the data reader can easily detect when a string starts and ends.

Serial transactions with the AAC are kept as simple as possible. The 24 byte serial communications packet is converted to ASCII characters for transmission, but for this discussion, the packet will be discussed in its binary form. The host initiating the transaction (which is always the MC), sends the serial communications packet to the AAC. The AAC receives the packet, converts it from ASCII to binary, and performs the function.

Once complete, the AAC responds with the same packet, this time with data fields populated if applicable.

| **Code** | **Alias** |
|---|---|
| 1 | ALTAZ_READ_CURRENT_POSITION |
| 2 | ALTAZ_MOVETO |
| 3 | ALTAZ_HALT |
| 5 | ALTAZ_PING |
| 6 | ALTAZ_SET_ALT_OFFSET |
| 7 | ALTAZ_SET_AZ_OFFSET |
| 8 | ALTAZ_SET_RTC |
| 9 | ALTAZ_SLEW_STATUS |
| 10 | ALTAZ_MOVE_AXIS |
| 11 | ALTAZ_INIT |
| 12 | ALTAZ_INIT_AXES |
| 13 | ALTAZ_INIT_SERVO_ELEV |
| 14 | ALTAZ_INIT_SERVO_AZIM |
| 15 | ALTAZ_INIT_MOTOR |
| 16 | ALTAZ_SLEWTO |
| 17 | ALTAZ_RD_POSLOG_RANGE |
| 18 | ALTAZ_RD_POSLOG_DATA |
| 19 | ALTAZ_INIT_POSLOG |
| 20 | ALTAZ_POSLOG_STATE |
| 21 | ALTAZ_REBOOT |

Table 3.2: IRMA AAC command codes sent over MC AAC serial link.

When the MC sends the serial packet, it populates the command code field with a serial command code. Function parameters, where necessary, are passed via data fields 1 through 4. A checksum is calculated over the entire packet excluding the last 4 bytes, and is stored in the last 4-byte field. The packet is converted into a serial ASCII string, where colons are used as field delimiters. The string is wrapped with **STX** and **ETX** characters

and transmitted over the 19.2 kbit/s serial link.

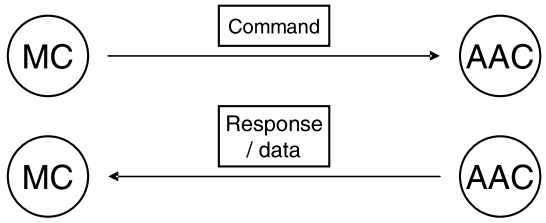


Figure 3.16: IRMA serial communications protocol.

The AAC has a real-time task devoted to monitoring serial data traffic. When it reads a **STX** character, it proceeds to read up to 80 characters or when at **ETX** character is encountered. A complete ASCII-serialized data packet populated with the full scale unsigned 32-bit integers along with 5 colon delimiters works out to 59 bytes. If the serialized packet was received without error, the AAC will respond with a the same packet, where the command packet is populated with the **MSCOMM_SUCCESS** code (integer value 101). Data is returned in the four data slots, and a checksum value is calculated and placed in slot 6.

A 5-second timeout is applied to every serial communications transaction. If the full serialized packet is not received within a 5 second period, the receiver assumes that the packet is lost, and responds with an error acknowledgment, shown in figure 3.18, where the command field is populated with the **MSCOMM_FAILURE** code (integer value 101), and each of the four data fields are populated with the **MSCOMM_ERR_TIMEOUT** code

| MSCOMM_SUCCESS |
|---|
| Data Field A |
| Data Field B |
| Data Field C |
| Data Field D |
| CRC |

Figure 3.17: IRMA serial communications packet: successful transaction.

(integer value 77,777,777). If the reader calculates a checksum different from the embedded checksum, an error packet with data fields populated with **MSCOMM_ERR_CRC** (integer value 66,666,666) is returned. Finally, if the incoming packet does not terminate within 80 characters (that is, no **ETX** character is detected before the 80th read character), an error packet with the data fields set to code **MSCOMM_ERR_BUF_OVERRUN** (integer value 55,555,555) is returned. These error code values were chosen as they are outside the scope of expected values returned by the AAC's optical encoder chip.

| MSCOMM_FAILURE |
|---|
| MSCOMM_ERR_TIMEOUT |
| MSCOMM_ERR_TIMEOUT |
| MSCOMM_ERR_TIMEOUT |
| MSCOMM_ERR_TIMEOUT |
| CRC |

| MSCOMM_FAILURE |
|---|
| MSCOMM_ERR_CRC |
| MSCOMM_ERR_CRC |
| MSCOMM_ERR_CRC |
| MSCOMM_ERR_CRC |
| CRC |

| MSCOMM_FAILURE |
|---|
| MSCOMM_ERR_BUF_OVERRUN |
| MSCOMM_ERR_BUF_OVERRUN |
| MSCOMM_ERR_BUF_OVERRUN |
| MSCOMM_ERR_BUF_OVERRUN |
| CRC |

Figure 3.18: IRMA serial communications packet: failed transactions.

## 3.7 IRMA Configuration and Data Files

This, the final section on the chapter on IRMA's software structure, is concerned with the configuration files required by the CP, MC and AAC, the data files produced during scans, and the setup requirements for the CP software.

### 3.7.1 IRMA CP Configuration

The IRMA CP software, unlike the MC and AAC software, runs within the context of an operating system and the Perl language interpreter. The CP software's ability to run is dependent on whether the Perl language is installed, if all the necessary Perl modules (libraries) are installed, and if all the configuration files, IRMA scripts and helper files are in the correct relative paths.

#### Compiling a Rabbit 2000 Boot Image

When setting up the IRMA system for the first time, the latest MC and AAC software should be loaded into their respective Rabbit controller's flash memory. The Rabbit Field Utility (RFU) program, bundled with the Dynamic C development software, is a small Windows-based program that loads Rabbit boot images (files having the `.bin` extension) into flash memory. Images are created within Dynamic C by performing a target-less compilation of the given Rabbit source code. First, the target board type must be defined by selecting *Define target configuration* from the *Options* menu. MC software must be compiled for the 22MHz RCM2100, CPU revision 1Q5T. AAC software must be compiled for the 25MHz RCM2010, CPU revision 1Q2T. Second, the line `#define R2K_VER_IQ5T` must be

present in the constant definition block of the MC source code. If a revision IQ3T processor is present on the RCM2100, the definition should be changed to `#define R2K_VER_IQ3T`. If a revision IQ2T module is being used, it is critical that the definition be changed to `#define R2K_VER_IQ2T`. The IQ2T definition selects an alternate compilation of the MC software that calls special work-around functions involved in setting up the data collection ISR. More details on this issue is found in the footnote below[1]. Third, the TCP/IP configuration information must be defined via the following constant definitions: _PRIMARY_STATIC_IP, _PRIMARY_NETMASK, MY_NAMESERVER, and MY_GATEWAY. Next, the compiler must be configured to target boot images for flash memory by selecting *Compiler* from the *Options* menu, then click on the radio button titled *Code and Bios in Flash*. The last step is to compile the boot image by selecting *Compile with defined target configuration* from *Compile to .bin file*, in the *Compile* menu. *Include debug code/RST 28 instructions*, found in the *Compile* menu, should **not** be selected. Complete instructions on using the Dynamic C development environment can be found in the Dynamic C User's manual[65].

### Preparing the Target Platform for the IRMA CP Software

The CP software is primarily designed to operate on UNIX or UNIX-like operating systems, such as Linux. First, Perl version 5.8.x should be present in order to ensure all

[1] If a new RCM2010 processor module is present in the Alt-Az unit, the chip will likely be revision IQ5T, which requires that the AAC software be compiled for revision IQ5T. At the time of writing, **no operational IRMA unit uses an RCM2100 microcontroller whose chip revision number is less than IQ3T**. Chip versions is of particular importance to the MC software becuase it uses external interrupts, and version IQ2T of the Rabbit 2000 processor contains a bug in its interrupt detection circuitry. More information on this problem can be found in Z-World's technical note TN201: Rabbit 2000 Microprocessor Interrupt problem, www.zworld.com/documentation/docs/refs/TN301/TN301.pdf. Again, this bug does not affect any currently operational IRMA MC board, as all the legacy (IQ2T) RCM2100 boards have been retired or discarded. This issue, however, is significant, because if a IQ2T board were installed, the ISR would not work without some modification.

the necessary Perl modules will compile. One can determine the version of Perl installed on one's system with the command `perl --version`. Version 5.8.1 is found on the in-house IRMA CP platform. Next, a compiler, preferably the GNU C compiler, should be present. The Perl modules were compiled using gcc version 3.3.2 and 3.3.3. GNU Fortran is also necessary in order to build some of the Perl modules. The TCL/TK software suite should also be installed, as it supports the GUI libraries used by the IRMA GUI software.

Once the languages are installed, the SLALIB positional astronomy library, followed by PGPLOT graphics subroutine library, should be installed. Next, the Perl modules should be installed. Table 3.3 contains a list of modules necessary to run the CP software. These modules can be installed downloading them individually from www.cpan.org and installing them manually (consult README file contained in each module archive), or by performing an automated network based installation. Issuing the command (with root privileges) `perl -MCPAN -e shell` from a shell starts the network installation environment. Modules can be installed by typing `install <module name>`, where the module name is found in the module column. The modules should be installed in the order they are listed in the table. The directory **IRMAdata** must be created. It will contain scan data collected by the MC. It should be owned by user **irmauser** and belong to group **users**. Naturally, a user account for **irmauser** will have to be created beforehand.

The next step is to place the IRMA CP software archive somewhere on the host system. The easiest method is to obtain the source tree from alpha.physics.uleth.ca via CVS using the command `cvs checkout IRMA`. To retrieve a source tree via CVS from the

| Module | Description |
|---|---|
| Bundle::CPAN | Perl module network installer helper utilities |
| Time::Piece | Object Oriented time objects |
| Time::ParseDate | Module for parsing both relative and absolute dates |
| Tk | GUI toolkit for Perl (Perl/Tk) based on TK 8.0 |
| Astro::SLA | Perl interface to SLAlib positional astronomy library |
| Astro::Constants | Physical constants for use in astronomy |
| DateTime | Date and time module useful for converting dates |
| DateTime::Locale | Localization support for DateTime.pm (above) |
| DateTime::Format::Strptime | Parse and format strp and strf time patterns |
| Tk::Date | Date/time widget for perl/Tk |
| Spreadsheet::ParseExcel | Extract information from an Excel spreadsheet |
| IPC::ShareLite | Light-weight interface to shared memory |
| Math::Round | Perl extension for rounding numbers |
| GD | Interface to Gd Graphics Library |
| Compress::Bzip2 | Interface to the bzip2 compression library |
| PerlIO::gzip | Perl gzip/gunzip compression utilities |

Table 3.3: Perl modules used by the IRMA CP Software.

source file server, alpha.physics.uleth.ca, the user must have the CVSROOT and CVS_RSH environment variables set. These can be set inside the user's shell initialization file. For example, the following lines are found in the author's .tcshrc configuration file:

```
setenv CVSROOT ":ext:ian@142.66.41.12:/files/projects/IRMA/Software/cvsroot"
setenv CVS_RSH "ssh"
```

The CP source code root directory structure appears in table 3.4.

The IRMA GUI-based control programs are located in the root of the IRMA CP source tree: `IRMA.pl`, and `viewirma.pl`. `IRMA.pl` is the main GUI control program to drive IRMA. `viewirma.pl` is a utility to view archived IRMA data files. To run IRMA from the command line, change the current directory to `IRMA/HelperProgs/` and issue the command: `./irmaExec.pl <UNIT_NUMBER> <SCRIPT_NAME> IRMAscript.xls`,

| Directory | Description |
|---|---|
| `auto` | Contains the CCIT16_CRC shared library (custom Perl module) |
| `Config` | Contains configuration files, IRMAscript command listing (Excel file) |
| `HelperProgs` | IRMA CP executables stored here (irmaExec.pl) |
| `IRMA` | IRMA CP Perl modules stored here |
| `SCRIPTS` | Contains scripts defining IRMA's operation |

Table 3.4: IRMA CP source code tree.

where **UNIT_NUMBER** corresponds to the box number of the unit, and **SCRIPT_NAME** refers to the name of the script (contained in the `SCRIPT` directory) to be executed. Each IRMA unit, or box, has an identification number that is used to address individual IRMA units in multiple unit deployment situations.

### 3.7.2 IRMA Configuration Files

Each IRMA unit (box) has a unique configuration file, stored in the `/IRMA/Config` directory. The configuration file shown below provides the CP with the IP address and data port of the master controller, and supplies the gear reduction ratios, servo parameters and detector calibration constants to the MC.

```
************************************************
2004-01-01T00:00:00
IPaddress 128.171.116.72
Data_port 10072
Cooler TR282
Board 1
# Dummy calibration data for this time period
CalibrateLow 77_7.74e6
CalibrateHigh 319_6.82e6
************************************************
2004-08-09T15:00:00
# Unit returned from Hawaii
IPaddress 142.66.41.40
ElevGearReduction 128
```

```
AzimGearReduction 128
BeltReduction 8
MinMotorRPM 500
MaxMotorRPM 25000
MaxGearRPM 8000
elev_kProp 10.0
elev_kInteg 1.0
elev_kDeriv 1.0
azim_kProp 1.0
azim_kInteg 1.0
azim_kDeriv 1.0
************************************************
```

The file is broken into parameter blocks, which are delimited by lines of repeating asterisks. A time stamp appears at the head of the block, which establishes the date/time when the immediately following parameters took effect. The parameters within a block include all lines following the time stamp, up to but not including the next block delimiter line. A parameter line consists of a label followed by a value, and is terminated with a carriage return. A whitespace separates the label from the value. Comments can be included in the configuration file by typing a pound sign "#" (or octothorp) at the beginning of the line.

When `irmaExec.pl` is executed, either through the `IRMA.pl` GUI, or via the command line, it reads in the box file specified by the box number command line parameter, accepting parameter fields whose time stamp is closest to the current time/date. For example, if `irmaExec.pl` were executed on some date in 2004, it would accept the IP address values from the command block dated `2004-08-09T15:00:00`, rather than from `2004-01-01T00:00:00`, which is chronologically 8 months earlier (approximately). Since the parameters `Data_port`, `Cooler`, `Board`, `CalibrateLow`, and `CalibrateHigh` do not appear in the more recent parameter block, these values are accepted.

The following list parameters can be defined in a CP configuration file. If parameters are not defined, default dummy parameters are assigned in their place.

**Data_Port**
The TCP/IP socket port that is used by the MC to send scan data to.

**Antenna**
The identification number of the antenna that the given IRMA unit is associated with. This parameter is not always used.

**ElevGearReduction**
The gear reduction ratio of the gear box driving the elevation axis.

**AzimGearReduction**
The gear reduction ratio of the gear box driving the azimuth axis.

**BeltReduction**
The gear reduction ratio caused by the drive belt. The total gear reduction ratio of a given gear is the sum of its gear box reduction ratio and the belt reduction ratio.

**MaxMotorRPM**
This is the vendor-specified maximum motor rotational rate, generated when full scale voltage is applied to the motor controller unit.

**MinMotorRPM**
This is the vendor-specified minimum motor rotational rate, generated when zero volts is applied to the motor controller unit.

**MaxGearRPM**
The maximum recommended rotational rate of the gear head (not the motor). This value is provided by the motor vendor.

**elev_kProp, elev_kInteg, elev_kDeriv**
Servo constants for the elevation axis motor. The three constants refer to the proportional, integration and derivative constants (PID), which must be determined by the user by tuning the servo algorithm.

**azim_kProp, azim_kInteg, azim_kDeriv**
Servo constants for the azimuth axis motor. The three constants refer to the proportional, integration and derivative (PID) constants, which must be determined by the user by tuning the servo algorithm.

**Location**
This refers to the name of the site where this given IRMA unit is located.

**Cooler**
The model number of the cryo cooler associated with this given IRMA unit

**Board**
An ID number which identifies the IRMA motherboard associated with this given IRMA unit.

**CalibrateLow**
This is the ADC count when the IR channel measures the unpowered shutter blackbody calibration source (cold). Calibration of the calibration target in hot and cold states (powered and unpowered) relates the IR measurement with a temperature reading from the same target.

**CalibrateHigh**
This is the ADC count when the IR channel measures the powered-up (hot) shutter calibration source. See the description of CalibrateLow for calibration details.

## 3.8 IRMA CP Data File Structure

When a scan is requested, the CP software forks a child process that reads scan data packets from the MC. The CP software opens a file in the `/IRMAdata` directory, and writes ASCII text to this file as data packets arrive. A typical data record appears as a space-delimited string, terminated with a carriage return. One data record corresponds to a single A/D sample. The example below shows a sample on channel 1 (the IR signal channel), taken March 28th, 2005 at 3:00 AM. The azimuth and altitude position of the mount have not been initialized.

| | | | | |
|---|---|---|---|---|
| 1 | 7680596 | 2005-03-28T03:00:21.803 | 3962.9004 | 3953.8477 |
| A | B | C | D | E |

Field A contains the channel number of the given sample. Channel numbers range from 1 through 11 inclusive. ADC channel usage is listed in table A.5. The raw A/D sample, given in ADC units out of the maximum full scale value, appears in field B. An ISO-formatted date/time stamp appears in field C. Field D contains current azimuth position, followed by current altitude position in field E. Azimuth and altitude positions are given in

degrees. When the Alt-Az mount has not been initialized, the default altitude and azimuth positions are 90,000 optical encoder units. This translates into 3955.07 degrees if one divides 90,000 by 8192/360 (or approximately 22.75), the number of optical encoder units in one degree.

The IRMA MC software automatically organizes scan data files into data directories unique to the IRMA unit that produced the data. For example, scans from box 2 will go into directory `/IRMAdata/IRMA_2/year`. The directory named <**year**> is the 4 digit year in which the data file was colleced. Data are additionally organized into directories labeled with the date the data were taken. Finally, data files are truncated on the hour, so a single data file will span no longer than one hour. Data collected over a long period, say 12 hours, 30 minutes, will be spanned across 13 files, the last file containing only a half-hour's worth of data.

## 3.9 Conclusion

This chapter has shown that the IRMA software is not a single executable entity – it is distributed across three separate hardware hosts in order to share the computational load. Given IRMA's distributed nature, robust communication protocols are used on IRMA's two primary communication channels: the CP-MC network link and the MC-AAC serial communication channel. Each of IRMA's software modules are structured to support multi-tasking through the use of the Micrium MicroC/OS-II real time kernel, which provides priority-based preemptive multitasking and real-time performance. The heart of the IRMA data collection system is the MC's data collection interrupt service routine, implemented

in assembly language in order to achieve maximum execution speed. Much of complexity of the IRMA software is due to its distributed multi-tasking nature, while the software routines responsible for controlling hardware and collecting data are generally quite simple. IRMA's complex structure, however, provides a solid foundation for the system allowing flexible control and extensibility.

# Chapter 4

# IRMA Software Modules

The previous chapter focused on the structure of IRMA's software components. This involved examining the relationships among IRMA's software components and the inner structure of the MC and AAC. This chapter will examine the algorithms powering some of the significant mechanisms within the IRMA software components, as well as some of the theory behind these algorithms. These mechanisms include the IRMA CP's IRMAscript language interpreter and the AAC's motion-control routines.

## 4.1 IRMAscript Language Interpreter

The Command Processor software, `irmaExec.pl`, is essentially a translator program, converting human-readable IRMAscript language into machine-readable binary packets that can be efficiently transmitted over the network and easily decoded. IrmaExec is also an interpreter, because it controls the behavior of the program, specifically in flow control (looping and branching). A comprehensive description of IRMAscript's syntax is

found in appendix A.

When IRMA software development was just beginning, it was anticipated that IRMA would need a highly flexible, fine-grained control mechanism considerably more powerful that that used for IRMA I or II. Previously, IRMA I's user interface consisted of a simple graphical user interface (GUI), and IRMA II's GUI relied on web page forms. It seemed logical that a custom language would serve IRMA's requirements. The IRMAscript control language grew out of these efforts.

In hindsight, a more elegant and powerful solution would have been to control IRMA using Perl, with the IRMA specific commands encapsulated in a custom-written Perl module. This would not be particularly difficult, given that the IRMAscript interpreter is written in Perl. Perl modules function as libraries that extend the functionality of the Perl language. Perhaps in a future version of IRMA, IRMAscript will be replaced with a driver library. However, some complex instrumentation systems do use custom scripting languages for control.

It must be emphasized, however, that IRMAscript is an ad-hoc implementation of an interpreted computer language. It is reminiscent of early pseudo-code interpreters, which translated assembly language-like mnemonic commands into their equivalent machine language statements[32]. IRMAscript's grammar is limited by the interpreter's crude design, which is driven entirely by regular expression pattern matching. The language, however, is adequate for performing the tasks required by IRMA, i.e., perform repetitive sequences of hardware control commands while certain conditions are held, write data to files, and perform simple arithmetic and logic operations.

Interpreters and compilers must translate instructions from one form to another. While interpreters perform the actions specified by the statements in the source code as they are encountered, compilers function entirely as language translators, generating machine language instructions, which can be executed by the target computer directly at some later time. Compilers, along with interpreters, share similar internal mechanisms. A typical compiler contains a scanner, a parser, scope checker and code generator[16]. The scanner, parser and scope checker make up the front end of the compiler, while the code generator (the part that outputs the CPU-executable machine instructions) constitutes the back end of the compiler. IrmaExec contains minimal implementations of all four functions. A typical compiler/interpreter will contain well-defined modules or classes handling each of these functions.

Scanners are responsible for recognizing the tokens, or strings, that make up a language statement. This involves reading the input file (the source code), throwing away the white space, and determining whether the tokens are literal values (numbers), named variables or constants, or reserved words, that is, the recognized commands of the language.

Parsers are tasked with determining if an input command string, consisting of a sequence of tokens, conforms to the syntax of the language. This is the most complex phase of compilation, and in many compiler designs, it is the parser that drives the compiling process. Typically, the parser fetches tokens (words) from the scanner until it can assemble a valid language sentence, or statement, according to a fixed set of production rules (a grammar) and a rule-lookup mechanism (an automata). Automata are similar in form to state machines. There are many algorithms, some more involved than others, that are

designed to recognize whether strings belong to a language. The IRMAscript interpreter uses a simple language recognition machine, called a finite automata, along with a simple set of production rules to parse IRMAscript statements.

Scope checking and type checking are generally combined in a compiler. Since IRMAscript does not consider data type, and all variables are considered global, that is, visible throughout the program, this stage is not included in the interpreter.

Code generation takes a verified language statement and translates it into lower-level machine-readable codes that can be accepted by the target processor. Compilers generate an executable file out of the machine codes, or opcodes, while interpreters execute these statements on the target machine. IrmaExec generates machine-executable codes, contained in discrete binary packets, which it sends over the network to the IRMA master controller. Where the IRMAscript statement does not command an IRMA hardware component, such as a variable assignment or flow control, the interpreter will execute the statement directly within the context of the irmaExec process, which in turn is being executed by the Perl virtual machine.

### 4.1.1 Computer Language Theory

IRMAscript is a language, albeit a small, trivial one. From first principles, languages consist of a set of strings, which are in turn made up of finite sequences of symbols. Symbols can be any kind of character, such as binary numbers, or text characters. A string containing zero characters is known as the empty string $\epsilon$. The finite set of symbols that constitute a language is called an alphabet $\Sigma$. For example, the alphabet $\Sigma = \{a, b, c, ..z\}$

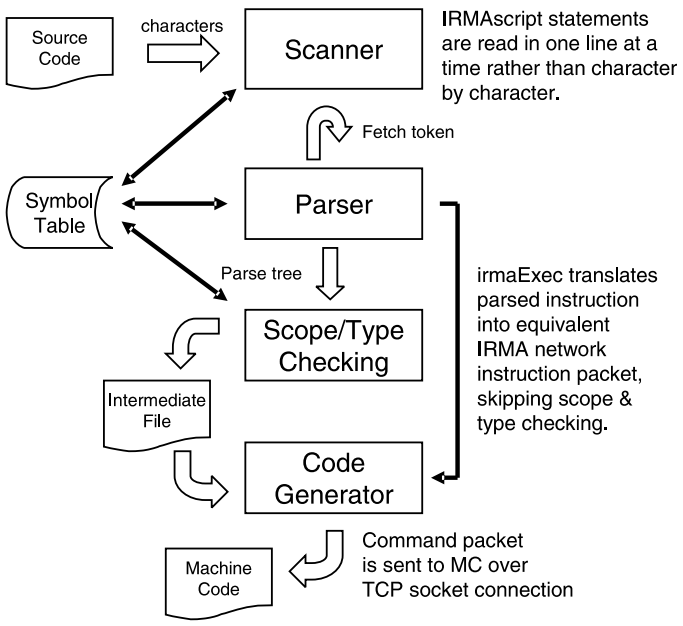


Figure 4.1: Block diagram of a typical compiler. IRMA's language interpreter skips scope and type checking since all variables are global and typeless.

can generate the language $L$ containing the strings $\{car, cat, bad, map\}$. Languages that contain zero strings are refered to as the empty set $\emptyset$. The question remains: how can a finite language made up of a finite set of strings be represented?

Languages can be defined inductively using the union $\cup$, concatenation, and Kleene closure $*$ set operations. Parenthesis are used to denote precedence. Union is sometimes expressed with the vertical bar symbol |, which denotes alternatives. Concatenation is analogous to the logical **and** operation. A Kleene closure is a set operation that defines a set of strings that can be generated by concatenating zero (the empty string) or more strings from another set of strings. Regular expressions (RE) can be written using set operations, parenthesis and alphabet symbols to describe languages or words in the language. The following list of rules defines a regular expression[30]:

1. The empty set $\emptyset$ and each member of alphabet $\Sigma$ are REs.

2. If symbols $\alpha$ and $\beta$ of alphabet $\Sigma$ are RE's, then $(\alpha\beta)$ is an RE. This amounts to concatenation (the **and** operation).

3. If symbols $\alpha$ and $\beta$ of alphabet $\Sigma$ are RE's, then $(\alpha \cup \beta)$ is a RE. This amounts to union (the **or** operation).

4. If symbol $\alpha$ is a regular expression, then $\alpha$ is an RE.

5. Nothing else is a RE.

For example, the RE $(a|b)^*c$ describes the set of all strings concluded by a single symbol $c$. Furthermore, if $\Sigma = \{a, b\}$, then the set of all strings of length 2 can be described by the RE $(a|b)(a|b)$. The fact that regular expressions can describe patterns of symbols makes them particularly useful in identifying words, or lexemes, that make up a language statement. Regular expressions, however, cannot describe nested constructs[30] of unspecified depth, such as found in a complex mathematical expressions. Thus, regular expressions can only describe regular languages. The IRMAscript language interpreter uses regular expressions to identify (or accept) IRMAscript statements, so IRMAscript can be considered a regular language. As such, IRMAscript does not permit nesting of expressions: only one, or in some cases two, expressions are permitted per statement, such as in the case of the `while` statement.

Regular expressions can describe strings of a language, but they cannot recognize if a string belongs to a given language. A language recognition device following the steps described in an algorithm must be used instead. Finite automata can be used to model

the algorithm driving language recognition devices. In essence, finite automata are models of minimal computers possessing no memory. Additionally, any regular language can be recognized by a finite automata[30].

A finite automata (FA) can be visualized as a black box that can be in one of $n$ discrete states depending on the input fed into the black box, by means of a metaphorical input tape. On the tape are printed input symbols that are part of some language. The black box has a reading head, which is analogous to the head on a tape recorder. When the read head passes over a character on the tape, the internal state of the black box changes to a new state, depending upon the character read and the previous state of the box. The read head can move forwards or backwards along the tape. Initially, the black box is in the start state. A string is accepted when the black box, reading the tape from left to right, reads a character that puts it into a final state.

Formally, a FA consists of the 5-tuple $M = \{\Sigma, S, \delta, s, F\}$, where $\Sigma$ alphabet of symbols, $S$ is a finite set of states, $\delta$ is a transition function (rules that define when the FA moves from state to state), a single start state $s$, and a set of final accepting states $F$. The transition function $\delta$ can be described in a table, showing how the given state and input symbol maps to a specific output state. Directed graphs, in the form of a state diagram, are commonly used to describe the transition function. FA's come in two forms: deterministic finite automatae (DFA) and non-deterministic finite automatae (NFA). While the state of a DFA is completely determined by its input and its current state, a NFA has the ability to choose one or more possible paths (as indicated in its diagram) after reading its input. Regardless of the type of FA used, for every NFA there exists an equivalent DFA. Therefore,

NFAs are convenient to use for diagramming purposes, as they are simpler to draw, as they have fewer state transitions. For example, the NFA that accepts the string $L = aba$ appears in figure 4.2.

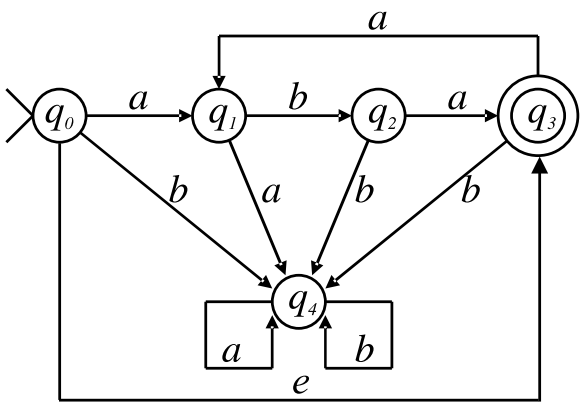


Figure 4.2: Directed graph of a NFA that accepts the language $(aba)^*$

Directed graphs are constructed from a series of nodes and arrows, where each node represents a NFA state, and each arrow is labeled with an input symbol. The node labeled $q_0$ has an arrow pointing to it, which indicates it is the start state. After reading its input tape, the NFA can choose to go to state $q_1$ if the input was an $a$, or go to state $q_4$ if the input was a $b$. Final state, $q_3$ appears as a double circle. The graph shows that any number of $aba$ character sequences, which includes zero sequences as indicated by the $e$ symbol, are accepted by the NFA. Any other input sequence results in the NFA entering an error state, meaning the input was identified as not belonging to the given language.

A NFA or DFA can be easily expressed in a programming language, as it is nothing more than a state machine. The NFA described in figure 4.2 can be expressed with the following C code:

```
state = q0;
while(ch = getc(FILE) != EOF)
{
   if(ch == 'a')
   {
      switch(state)
      {
      case q0:
         state = q1; break;
      case q1:
         state = q4; break;
      case q2:
         state = q3; break;
      case q3:
         state = q1; break;
      case q4:
         state = q4; break;
      }
   }
   else if(ch == 'b')
   {
      switch(state)
      {
      case q0:
         state = q4; break;
      case q1:
         state = q2; break;
      case q2:
         state = q4; break;
      case q3:
         state = q4; break;
      case q4:
         state = q4; break;
      }
   }
}

if(state == q3)
   printf("string accepted\n");
else if(state == q4)
   printf("string not accepted\n");
```

The IRMAscript interpreter is written in Perl, a popular systems programming

language that has powerful regular expression pattern matching tools. The algorithm to match the string *aba* can be expressed in Perl simply as:

```
while(<FILE>)
{
   $statement = $_;
   if($statement =~ /aba/)
   {
      print "string accepted\n";
   }
   else
   {
      print "string not accepted\n";
   }
}
```

IRMAscript syntax is extremely simple in order to avoid nested statements, which would require a recursive parser. The general structure for hardware commands follows the pattern of one command and two command modifiers:

**[Command] [Modifier field 1] [Modifier field 2]**

followed by $n$ parameters, depending on the command issued. All other IRMAscript commands, such as flow control commands, delays, and console I/O, may use one or no modifiers, according to their function. In order to make the language easily readable, a three-tuple syntax is used to divide the commands into families, where each family consists of subcommands that perform specific fine-grained functionality relating to that family. For example, the functions dealing with the Master Controller's real time clock are encompassed in the RTC command family.

Inside the interpreter, the input source code file is opened, and each line is read individually, split into tokens along the whitespace contained in the statement, and entered

```
RTC  READ  DATE_TIME
RTC  READ  EPOCH_TIME
RTC  SET   ARBITRARY_TIME
RTC  SET   DATE_TIME
```

into an associative array structure, which operates like a hash table. A line is identified as a string of text terminated with a carriage return character (ASCII decimal code 10). Line numbers, generated by the interpreter, formulate the hash key. This mechanism allows convenient access to individual IRMAscript statements by simply passing a line number to the associative array, which are often referred to in Perl terminology as a *hash table*.

Once the entire script has been read into the program code hash table, the interpreter executes a program by iterating through the range of line numbers and their associated IRMAscript statement, starting at line zero. Iterative loops (`repeat` $n$`..endloop`) as well as conditional loops (`do..while`) require a mechanism to permit jumping back to the top of the loop. Upon entering loop for the first time, an array indexed with the current loop nesting depth is given the line number of the statement immediately following the loop head only after its index is incremented. This is the target address that the interpreter uses to jump to the top of the loop when it encounters the bottom of the loop. The loop nesting index allows the interpreter to keep track of which loop head address it should use, according to which loop is active. Iterative loops also use an array (indexed to the current loop nesting) to store the loop iteration count. This variable is initialized to the `repeat` parameter and is decremented on each pass through the loop. The interpreter exits the loop when the iteration variable is decremented to zero, then decrements the loop nesting array.

```
10 repeat 3 // outer loop head
11    repeat 2 // inner loop head
12       print "some_string,\n"
13    endloop
14 endloop
```

In the preceding example, the interpreter will repeat the inner loop three times, as defined in the outer loop. The loop nesting index before entering the loop structures is zero. Upon entering the outer loop, the loop nesting index is incremented to 1. Upon entering the inner loop, the loop nesting is incremented to 2. Entering a loop increments this index, while exiting a loop decrements it. Values stored in the array at a particular index value are therefore unique according to the current loop nesting level. The loop iteration index is used the same way: entering a loop increments its index, while exiting a loop decrements the index. Below is a snippet of the actual code that controls the `repeat..endloop` construction.

```
if($statement[0] =~ /^REPEAT/)
{
   # dereference argument 1
   if( defined($variables{$statement[1]}) )
   {
      $statement[1] = $variables{$statement[1]};
   }

   if($statement[1] == 0)
   {
      $loopNesting++;
      $iterations[$loopNesting] = 0;
      $pc = moveToEndOfBlock();
      $pc++;
   }
   else
   {
      $iterations[$loopNesting] = $statement[1];
      $loopBase[$loopNesting] = $pc+1;
      $loopNesting++;
```

```
            $pc++;
        }
        $tokenMatch = 1;
        last SWITCH;
    }

    if($statement[0] =~ /^ENDLOOP/)
    {
        if($loopNesting > 0)
        {
            $iterations[$loopNesting-1]--;

            if($iterations[$loopNesting-1] < 1)
            {
                $pc++;
                $loopNesting--;
            }
            else
            {
                $pc = $loopBase[$loopNesting-1];
            }
        }
        $tokenMatch = 1;
        last SWITCH;
    }
```

IRMA hardware command functions must be converted from the three-string human-readable format to the numeric three-digit command code embedded in network command packets. The IRMAscript hardware command set is stored separately from the interpreter in an Excel spreadsheet file in tabular format, which to allows for easy modification. One of the first tasks performed by irmaExec upon start-up is to read the command set spreadsheet and extract the three digit code based on a simple algorithm. IrmaExec uses the **Spreadsheet::ParseExcel** Perl module to parse the Excel spreadsheet. The command, modifier1 and modifier2 codes are generated by sequential counters that increment when data contained in the corresponding command, modifier1 and modifier2 spreadsheet

columns show transitions. Once calculated, the three-digit code is entered into the **commandCodeHash** hash table, using the command and two modifier strings as keys.

For example, the **gps** family of commands, as shown in table 4.1 appear as the second block of related commands in the command set spreadsheet, thus their command code equals 2. GPS commands that handle reading data are identified by their first modifier field equaling 1, while serial port control commands are identified by the number 2. Each individual reading or serial function is uniquely identified by means of modifier field 2. The goal of this naming scheme is to uniquely identify each hardware control command. When modifying the command set spreadsheet, new entries must be added to the bottom of the block of commands for some given command family, in order to prevent changing the numbering scheme. A list of command strings and their corresponding codes can be generated within irmaExec by uncommenting the print statement immediately following the label **CMDSET_PRINT**. Command codes are hard-coded in the MC software running on the Rabbit.

| **String** | | | **Numeric** | | |
|---|---|---|---|---|---|
| **CMD** | **MOD1** | **MOD2** | **CMD** | **MOD1** | **MOD2** |
| GPS | READ | DATE_TIME | 2 | 1 | 1 |
| GPS | READ | EPOCH_TIME | 2 | 1 | 2 |
| GPS | READ | LAT_LON | 2 | 1 | 3 |
| GPS | SERIAL | OPEN | 2 | 2 | 1 |
| GPS | SERIAL | CLOSE | 2 | 2 | 2 |

Table 4.1: GPS command codes: string versus numeric representation.

The IRMAscript interpreter makes extensive use of Perl language features, in par-

ticular, hash tables and regular expression pattern matching. Further specialized functionality, such as parsing spreadsheets or calculating CCIT-16 CRC checksums, is implemented via Perl modules. The interpreter is initialized via configuration files, which includes its own command set, as contained in an Excel spreadsheet file. The overall structure of the IRMAscript interpreter can be summarized in the following pseudocode:

```
initialize command code hash table

open irmascript file (read)
do
   read line from file
   split line into fields, put into array
   put array in source code hash table with key = line count
   increment line count
until reach EOF

initialize program counter "pc" to 0

do
   get statement from source code hash table using key = pc
   pattern match statement on command, modifier1 and modifier2
   look up command code using keys command, mod1 and mod2
   make command packet
   send command packet to MC according to network comm protocol
while pc < total lines in program
```

This concludes a general overview of the structure and theory behind the IRMAscript interpreter. A complete description of the IRMAscript language is found in appendix A.

## 4.2 Alt-Az Controller Software

The altitude-azimuth (Alt-Az) mount is capable of pointing the IRMA MC unit to any Alt-Az coordinate in the sky with 1 encoder unit precision; roughly 1/22 of a degree. This is derived from the fact that a the optical encoder contains 8192 ticks per a 360 degree

revolution: one encoder tick equals 360/8192, or roughly 1/22 degrees. The elevation axis has 198 degrees of rotation, which allows it to slew (or rotate) from one horizon to another. The azimuth axis, meanwhile, is capable of rotating roughly 365 degrees. The AAC operates internally on the local (or horizon) coordinate system, using encoder units as its basis of angular measurement. Celestial objects, however, are located in terms of right ascension (RA) and declination (Dec).

The equatorial coordinate system is used to locate celestial objects on the celestial sphere, an imaginary spherical shell that surrounds the Earth. Just as one locates an object on the earth's surface using 2 coordinates, latitude for the Y-axis and longitude for the X-axis, one locates celestial objects on the surface of the celestial sphere using declination (DEC) for the Y-axis and right ascension (RA) for the X-axis[13]. Coordinate conversion is done outside the IRMA system, although early on in IRMA development, a RA-DEC Alt-Az conversion library was created and tested for inclusion in the IRMA MC software.

### 4.2.1 Alt-Az Initialization

Prior to using the Alt-Az mount, physical elevation and azimuth reference points, or fiducial markers, must be determined through a process called homing the axes. The homing process involves a sequence of steps, as shown in the `altaz_init.irma` script. Initially, the LS7266R1 optical encoder chip (OE) must be reset, and its elevation and azimuth counters set to 10,000.

Axis initialization process, which is known as homing, involves determining the location of both the clockwise (CW) and counterclockwise (CCW) rotational limits of the elevation and azimuth axes. The homing procedure for the elevation axis follows the se-

quence illustrated in figure 4.3. Initialization begins with the elevation axis rotating in the CCW direction until it encounters and overshoots the CCW limit, shown as the black arrow starting at point *a*. The AAC backs out of the limit, as shown by the red arrow, until it reaches the threshold of the CCW limit. The AAC makes note of this position, and rotates the elevation axis in the CW direction, as indicated by the blue arrow starting at point *c*. When the CW limit is encountered and overshot, the AAC halts the axis, backs up slowly in the CCW direction, as shown by the green arrow, until the elevation axis is on the threshold of the CW limit. The AAC takes note of the position, calculates the rotational range (in optical encoder ticks) from difference between the two limits, then initializes the elevation axis position counter to 10,000.

Position 10,000 is the starting position from which all rotational movements are made, and represents 0 optical encoder ticks from the CW limit. An offset value of 10,000 was chosen to prevent the OE counter from wrapping to 16,777,216 when it reverses direction and rotates below the 0 angle mark. This will occur when the axis is near the 0 angle threshold. The OE control chip contains a 24-bit counter which produces unsigned binary coded decimal (BCD) values. CCW rotation causes the OE counter to increment, while CW rotation causes the OE counter to decrement. IRMA reports the elevation and azimuth positions with the offset subtracted from the raw OE counter values. The elevation axis has approximately 198 degrees of free rotation, allowing the IRMA unit to slew 180 degrees from horizon to horizon.

The homing sequence on the azimuth axis is largely identical to that described

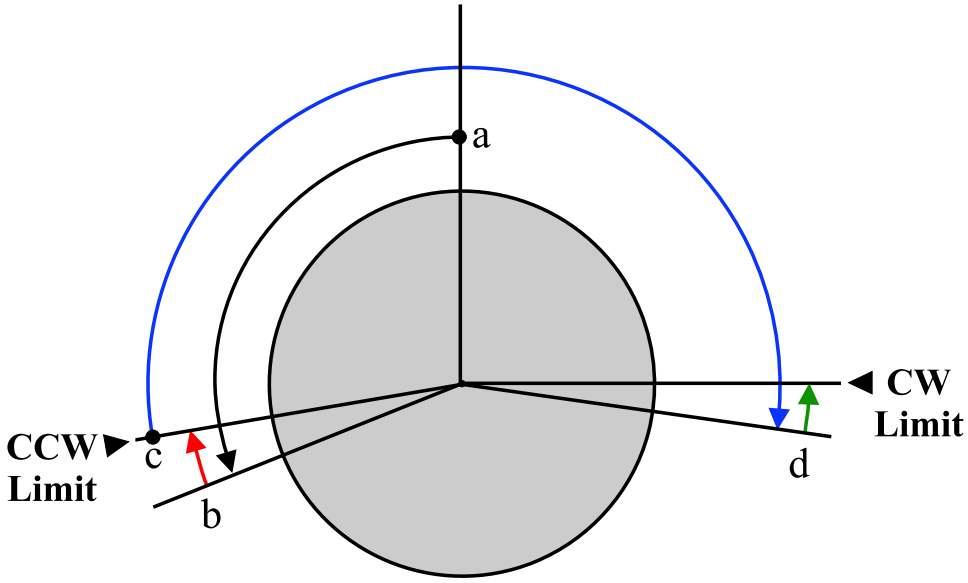


Figure 4.3: Initialization sequence of the elevation axis. Initialization, also called homing, follows the rotation sequence illustrated by the four arrows labeled a through d. Homing begins with a CCW rotation (a), a high-precision search for the CCW limit (b), a CW rotation to the CW limit (c), concluding with a high-precision search for the CW limit (d). The azimuth axis homing procedure follows the same sequence of events. The range of rotation on the azimuth axis, however, is slightly greater than 360 degrees.

for the elevation axis. The only difference is the azimuth axis' ability to rotate a full 360 degrees. The degrees of rotation between the CW and CCW limits is slightly greater than 360 degrees, due to the design of the optical limits mechanism.

### 4.2.2 Alt-Az Offsets

Internally, IRMA considers 0 degrees elevation and 0 degrees azimuth as the counter clockwise physical limits of both axes, which are defined by their respective opto switches. The elevation opto limit roughly corresponds to the physical horizon. When IRMA is pointing at its default elevation *home* position (optical encoder reading 0), its field of view dips slightly below the horizon. With azimuth, however, there is no physical

correspondence between azimuth home position and true north. As a consequence, IRMA requires the use of elevation and azimuth offsets, which must be applied to the encoder readings when planning axis moves.

Optical encoder counter readings form the benchmark that IRMA measures its position against. For example, 90 degrees on the elevation axis is located directly above the IRMA unit (zenith). IRMA Alt-Az movements are specified in terms of degrees (in degrees, minutes, seconds) above the horizon, not in degrees relative to its current position, or distance. IRMA interprets movement destinations in the same way. The benefit of using absolute angles is that it simplifies the use of offset angles.

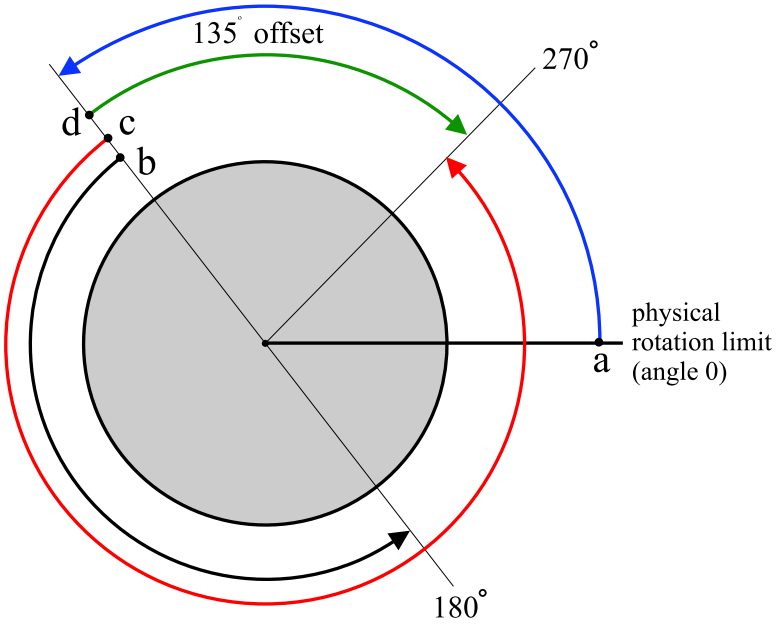


Figure 4.4: Azimuth axis rotation examples with an offset (here defined as 135 degrees). The blue arrow (a) shows a rotation to 0 degrees. The black arrow (b) shows a rotation to 180 degrees. The red arrow (c) shows a rotation to 270 degree, which wraps across the physical rotation limit. Since the destination lies 45 degrees beyond the physical limit, the AAC would rotate the axis in the CW direction, shown as by the green arrow (d).

Offset angles are added to both elevation and azimuth OE readings. Movements relative to the offset angle are illustrated in figure 4.4. If one were to move 180 degrees (the black arrow labeled b) and the offset angle were 135 degrees (the blue arrow), the resulting destination angle would be 180 + 135 = 315 degrees. In the case of elevation angles, it is possible to obtain negative angles if the axis dips below the angle of offset. This does not occur with angle measurements on the azimuth axis because it is capable of rotating a full 360 degrees. It is possible to request azimuth angles that wrap across the optical limit. The red line (labeled c) shows the path of rotation resulting from a request to rotate to 270 degrees, given a starting angle of 180 degrees, and a 135 degree offset. The solution is to subtract 360 from all destination angles greater than 360 and take the absolute value, which in this case is 45 degrees.

$$dest_{deg} = |(destAngle_{deg} + offset_{deg}) - 360_{deg}| \tag{4.1}$$

The path of rotation taken by the axis appears as the green arrow d, which rotates in the CW direction (towards physical angle 0) in order to avoid the 360 degree physical limit.

### 4.2.3 Axis gearing and speed

The axis rotation speed is determined by the input voltage value into the voltage controller and the gear reduction ratio for a given axis. Additionally, each gearbox has a maximum recommended rotation rate, which is fixed at 8000 RPM. The relationship between input voltage and motor rotation speed is linear, and can be determined from the manufacturer's motor and gear specifications. The parameters in question are shown in table 4.2.

| **Motor Attribute** | **Value** |
|---|---|
| Elevation Gear Reduction Ratio | 1621:1 |
| Azimuth Gear Reduction Ratio | 3027:1 |
| Belt Gear Reduction Ratio | 8:1 |
| Maximum Motor Speed (2.5V) | 12,500 RPM |
| Minimum Motor Speed (0V) | 500 RPM |
| Maximum Recommended Motor Speed | 8000 RPM |

Table 4.2: Maxon motor parameters.

The minimum and maximum motor speeds provide the slope of the linear equation defining the relationship between input voltage and output axis speed, shown in equation 4.2. The full scale voltage is 2.5 V. An offset of 500 RPM defines the Y-intercept of the equation. Voltages are in volts, and all speeds are defined in RPMs.

$$axisSpd_{\frac{rev}{min}} = \left( \frac{\left( MaxMotSpd_{\frac{rev}{min}} - MinMotSpd_{\frac{rev}{min}} \right)}{FullscaleVoltage_V} \right) InputVoltage_V + MinMotSpd_{\frac{rev}{min}} \tag{4.2}$$

Equation 4.2 gives the output axis speed in RPMs without considering the effect of the gear reductions due to the gear head and drive belt. Also, the output motor speed must be truncated at 8000 RPM. For example, an input voltage of 1.25 V (half of full scale) translates into:

$$axisSpd_{\frac{rev}{min}} = \frac{\left( \frac{\left( 12{,}500\,\frac{rev}{min} - 500\,\frac{rev}{min} \right)}{2.5_V} \right) 1.25_V + 500\,\frac{rev}{min}}{3027 \times 8} \tag{4.3}$$

Since the output motor speed (in RPM) is less than 8000, the value is legal, allowing us apply the gear reduction by dividing the motor rotational speed by the product of the gear reduction ratios, resulting in a net rotational speed of 0.27 RPM, or 3.72 minutes per revolution.

It is useful to know the net rotational speed of the axis in terms of ticks per second along the DAC value required to generate this speed. Ticks per second is calculated by:

$$axisSpd_{\frac{ticks}{s}} = \frac{\left(netMotorSpd_{\frac{rev}{min}}\right)_{\frac{ticks}{rev}}}{60_{\frac{s}{min}}} \tag{4.4}$$

Optical encoder ticks per second is the unit of speed used by the IRMA AAC. There are 8192 ticks per revolution (360 degrees). Each tick is equivalent to 2.63 arcminutes, as calculated by $\left(\frac{360_{deg}}{8192_{ticks}}\right) 60_{\frac{arcminutes}{deg}}$. To determine the net motor speed in terms of an input digital to analog (DAC) value (range: 0 - 255), simply substitute $FullscaleVoltage$ with 255, which is the full scale 8-bit DAC value. This effectively changes the slope of the linear equation defining this relationship.

### 4.2.4 Servo Motion Control

The elevation and azimuth axes require fine motor control in order to perform point to point moves that are accurate within one encoder unit. Motor speed must be controlled as to gently accelerate and decelerate the motors, thus avoid damaging the gear heads with sudden starts and stops. A trapezoidal-shaped velocity versus time profile will produce this kind of motion (see figure 4.5), and can be easily generated using basic kinematic equations for constant acceleration.

Servo-controlled motion control is generated by calculating a position versus time displacement profile, based on user-supplied speed and a constant acceleration value. The displacement profile generating function, known as `update_mp_position`, breaks the curve into three phases: the acceleration phase, cruise phase and deceleration phase. They can be clearly seen in the velocity versus time plot in figure 4.5. To generate this profile, the

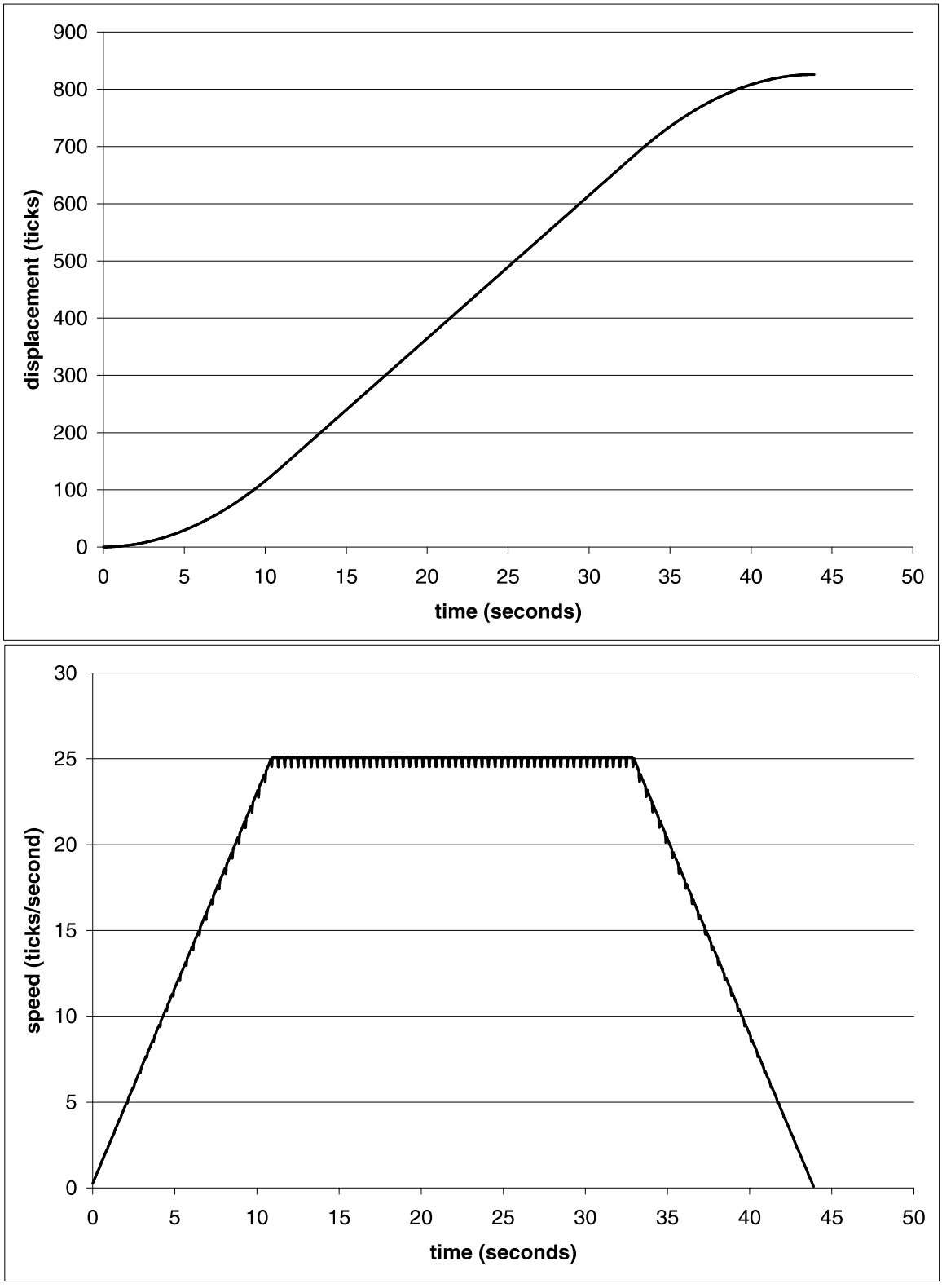


Figure 4.5: Displacement and velocity paths, generated by IRMA's servo motion control software. This path describes a 36.3 degree (826 ticks) rotation at 20 ticks per second.

AAC must calculate a unique displacement at a prescribed rate given a target speed and acceleration. Since deceleration has the same magnitude as acceleration, the position at which deceleration should begin can be calculated by subtracting the acceleration distance from the requested move distance. The remaining distance between the acceleration and deceleration phases is the cruise distance.

When the AAC begins to perform a servo-controlled movement, it zeros a time counter, zeros the displacement variable, and enables a high-priority task that signals the servo movement every 50 ms to read its current position and calculate a theoretical position along the displacement curve, based on the elapsed time relative to the start of the movement. At each tick increment in the servo movement, the displacement curve is in one of three states: acceleration, cruise and deceleration. Each state uses a unique algorithm to calculate its theoretical displacement from the start of the movement, given the current time. The displacement curve's state, which is initialized to the acceleration state, is promoted to the next state when the newly calculated displacement crosses the end of acceleration threshold, or the start of deceleration threshold. When a displacement is found to cross the end of deceleration threshold, the destination has been reached, and axis movement his halted by enabling the axis brake.

Displacements for the theoretical displacement profile are generated using motion equations of constant acceleration, which are found in any first year college physics textbook. During the acceleration phase, displacement is calculated using equation 4.5, where $D$ is in ticks, $T$ is in seconds, and $A$, the user-defined constant acceleration value, is defined in

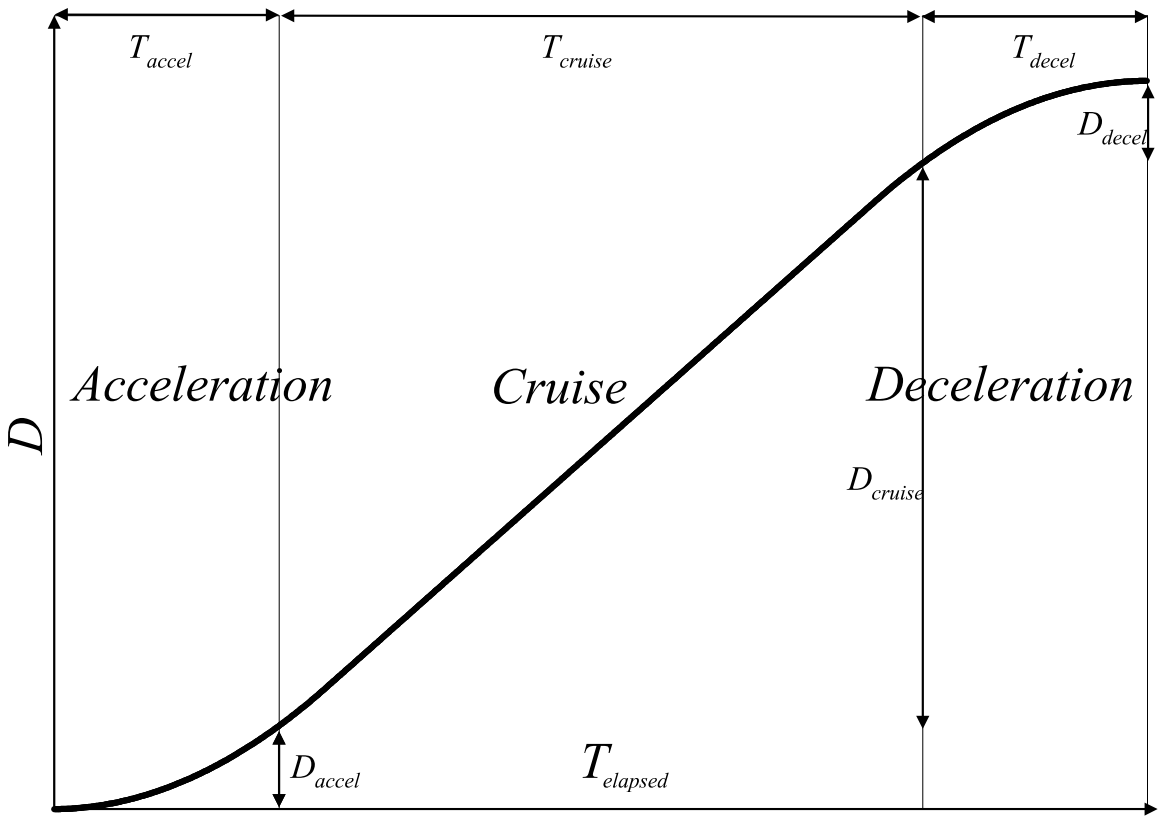


Figure 4.6: Displacement curve generation. Each region of curve: the acceleration, cruse and deceleration phases, has a unique equation for calculating displacement $D$.

$\frac{ticks}{seconds^2}$.

$$D_{ACC} = \frac{1}{2}A(T_{elapsed})^2 \tag{4.5}$$

The cruise phase is characterized by constant velocity. Displacement during this phase is calculated by equation 4.6. Velocity is measured in $\frac{ticks}{seconds}$.

$$D_{CRUISE} = D_{ACC} + (V(T_{elapsed} - T_{ACC})) \tag{4.6}$$

Calculating displacement during the deceleration phase, defined in equation 4.7, is more complex as it must take into account times and displacements from the previous phases. Time $T$, velocity $V$ and displacement $D$ remain in the same units defined in the previous

phases. Equation 4.8 defining $R$ (in seconds) and equation 4.9 defining $Q$ (in ticks) represent intermediate steps in calculating $D$.

$$D = Q + (\frac{1}{2}(-A)R^2) \tag{4.7}$$

$$R = T_{elapsed} - T_{ACC} - T_{CRUISE} \tag{4.8}$$

$$Q = D_{ACC} + D_{CRUISE} + (V(T_{elapsed} - T_{ACC} - T_{CRUISE})) \tag{4.9}$$

The displacement profile must be tracked over time, whereby the axis in question reaches position $s$ at time $t$, as dictated by the profile. The algorithm that performs the tracking, that is, conforms the physical behavior of the machine to the desired behavior is called its *control law*, or control algorithm[33].

Control systems, such as IRMA's AAC, are closed loop systems as they use feedback after applying input to the plant, which in this case is a voltage driving the motor controller. The difference between the desired result (the set point) and the feedback value (typically measured from a sensor) is the error signal, $E$. Control algorithms attempt to converge the error signal to zero. The degree to which this is done successfully is dependent upon the control algorithm being used, and the nature of the system being controlled.

IRMA servo-based move command uses a proportional-integral-derivative (PID) servo loop as its control law. Proportional feedback control multiplies a gain constant $K_p$ with the error signal: thus $K_p$ is proportional to the magnitude of the error $E$. When the physical system deviates from the desired behavior a little, a small amount of correction is applied. When the physical system deviates by a more significant amount, however, much correction is applied. Depending on the amount of gain applied, this can result in overcompensation, or overshoot, which can cause the system to oscillate. Reducing

proportional gain can reduce overshoot and possible oscillation (called ringing), but it may introduce more steady state error. Using proportional control alone, it is difficult to balance the goals of reducing oscillation, decreasing convergence time, and reducing steady state error, because they typically compete with one another[63].

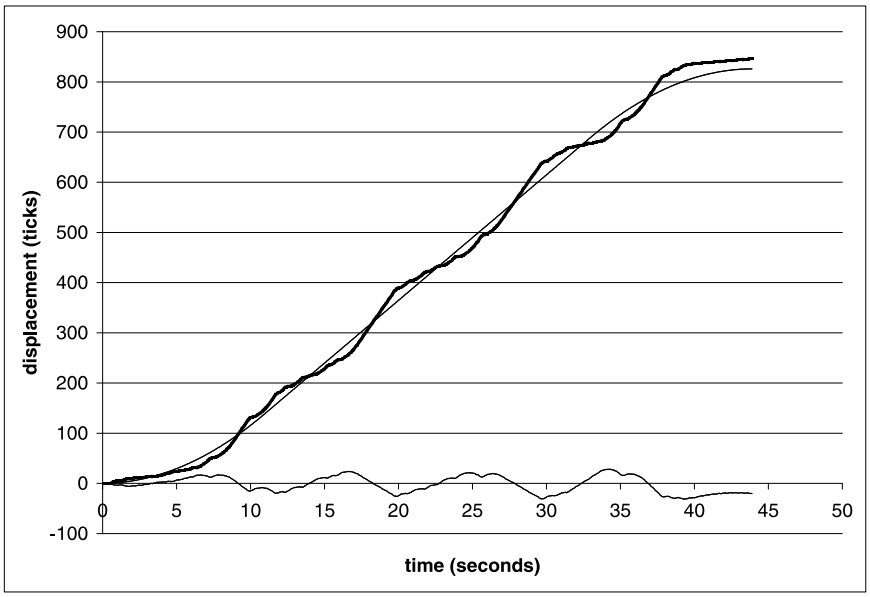


Figure 4.7: Motor speed oscillation due to poorly chosen or untuned P, I and D constants. The thick line represents the actual axis displacement from 0 to 826 encoder units (ticks). The thin S-shaped displacement curve represents the theoretical path that the PID servo loop attempts to track, represented by the thick line. The error signal is shown as the thin line oscillating about the X-axis.

Integration and derivative control terms are often included with the proportional term to achieve these control goals. Integration involves summing all the previous error terms and multiplying the result with a constant, $K_I$. The integral term is useful for increasing long-term accuracy by reducing error[33]. As $K_I$ is increased, the rate at which the error converges to zero increases. The derivative control is focused on the rate of change in the error signal. The derivative gain constant, $K_D$, is multiplied with the the derivative of

the error signal, which is defined as the change in the error signal over time. The derivative term acts to predict future system behavior based on what happened in the past. If the error has changed slowly in the past, it will likely do so in the future. The derivative term gives the controller the ability to generate a strong response against sudden changes in the error signal[33]. Combining derivative with proportional control allows control of the system's transient response (rate of convergence and oscillation control) without affecting steady state tracking[63]. Combining the P, I and D constants together yields the following equation:

$$MV = K_p E + K_I \int Edt + K_D \frac{dE}{dt} \tag{4.10}$$

Where $E$ is the error, $t$ is time and $MV$ is the manipulated variable, that is, the value that is to be fed back into the physical system, or plant. A block diagram representing this expression appears in figure 4.8.

In applying the PID control algorithm to IRMA's velocity tracking motion control source code, appearing in the following code snippet, the error signal (`fE`) is calculated as the difference between the theoretical position (`fRelPosTH`) along the displacement profile and the actual position (`fRelPos`) reported by the optical encoder (line 1). Calculation of the proportional term occurs in line 3. The integral term calculation (line 4) involves multiplying the integral constant with a running sum of each error value (`fIntegSum`) multiplied by the change in time, `fDeltaTime`. The time delta, which is set to 50 ms, determines the rate at which the PID servo loop checks its feedback and performs the necessary adjustment to the system. Finally, the derivative term is calculated by the derivative constant multiplied by the change in error (`fDeltaE`) over the change in time, as shown in line 6.

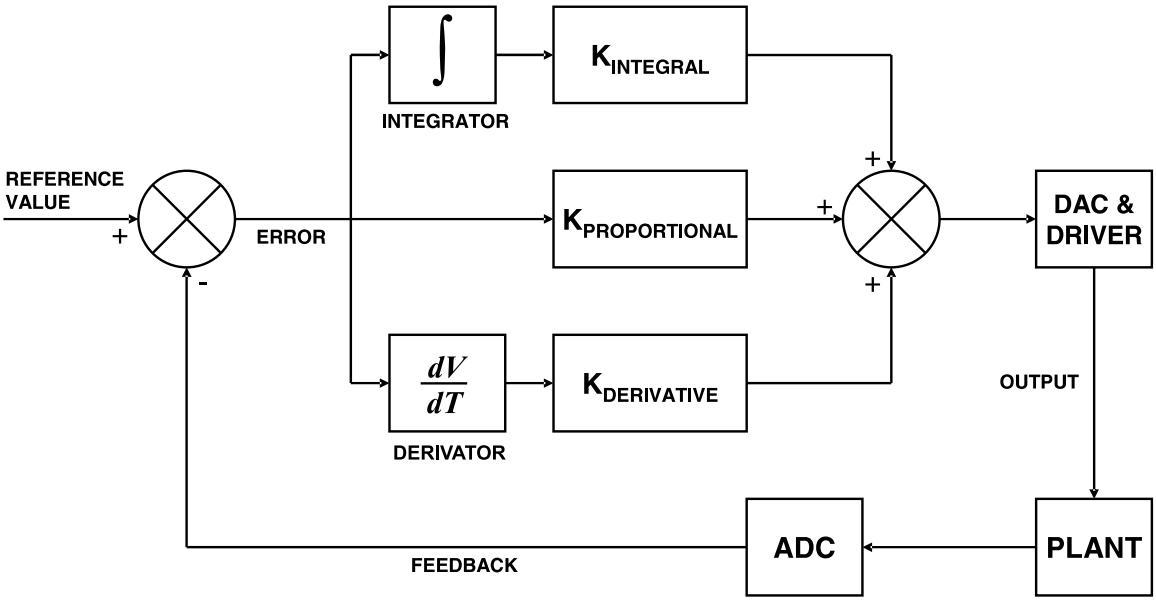


Figure 4.8: PID algorithm block diagram[64]

Once the three terms are added together (`fMV`), the resulting value is scaled to an 8-bit value that can be fed into the DAC. A full view of IRMA's PID servo routine is found in the `SingleAxisMoveTask` function, which appears in the AAC source code.

```
1  fE = fRelPosTH - fRelPos;
2  fDeltaE = fE - fPrevE;
3  fMVp = altazConstants.elev_kProp * fE;
4  fMVi = altazConstants.elev_kInteg * fIntegSum;
5  fIntegSum = fIntegSum + ( fDeltaTime * fE );
6  fMVd = altazConstants.elev_kDeriv * (fDeltaE / fDeltaTime);
7  fMV = fMVp + fMVi + fMVd;
```

## 4.3 Conclusion

This chapter has examined the IRMAscript language interpreter and the regular expression pattern matching algorithm that powers language statement parsing. This chapter has also examined the AAC software that controls the Alt-Az mount, focusing on the axis initialization process, how destination angles are calculated, and how trapezoidal velocity curves are calculated and translated into axis movements, using a PID servo tracking algorithm. If IRMAscript is going to continue to be used in future IRMA models, the interpreter would benefit from a redesign, where the regular expression-based parsing algorithm would be replaced with a more sophisticated parser, such as a top-down recursive descent parser. Eliminating IRMAscript entirely, however, and using a Perl module encapsulating IRMA system control commands would be a preferred solution, because IRMA scripts could be defined in Perl, which is a popular and versatile language.

# Chapter 5

# Future directions for IRMA

## 5.1 Testing Campaigns

### 5.1.1 Mauna Kea, 2004

IRMA III underwent initial field testing at the Smithsonian Submillimeter Array (SMA), Mauna Kea, Hawaii between May 24 and June 16, 2004. The testing campaign set out to demonstrate that multiple IRMA units could track PWV variations, and that its PWV measurements, when converted to phase variation measurements, could closely follow data from the Smithsonian Astrophysical Observatory (SAO) seeing monitor[43].

Figure 5.1 shows 4 and a half hours of data simultaneously collected by two IRMA units, each of which were attached to two SMA antennas separated 141 m apart. The black and red traces at the bottom of the figure show the spectral power (in watts) of the IR signal received by each antenna. The offset between the two signals is caused by using a common filter profile during data processing, and the lack of correct temperature sensor

calibration on one unit's blackbody. In reality, each IRMA unit's filter has its own unique profile, which needs to be applied to its own data set. The top trace shows the difference in spectral power measured between the two antennas. The two signals track each other closely, since the two antennas are pointed the planet Jupiter. A region of signal instability appears approximately 1500 seconds after the beginning of the observation session. For approximately 1500 seconds, one of the antennas pointed away from Jupiter, resulting in the unstable region where the signals do not correlate. Approximately 3000 seconds into the observation session, the unaligned antenna is pointed again at Jupiter. From this point onwards, the two signals show close correlation, as indicated by the top curve showing the difference in signal between the two antennas. Atmospheric turbulence appears as rapid fluctuation in the spectral power readings.

This particular observation session commenced in the early afternoon, around 14:00 local time (Hawaii standard time: HST). The spectral power data shows rapid fluctuations in the first 8000 seconds (roughly 2 hours) of the observation, indicating atmospheric turbulence. The turbulence decreases as the observing session moves into the evening. This atmospheric behavior is indicative of Mauna Kea's well known temperature inversion layer, where a layer of cold, moist air is trapped at lower altitudes by an upper layer of hot, dry air. As the ambient temperature drops in the afternoon, this inversion layer breaks down, allowing the cold moist layer to bubble up over the summit, resulting in the atmospheric turbulence appearing in this data[43].

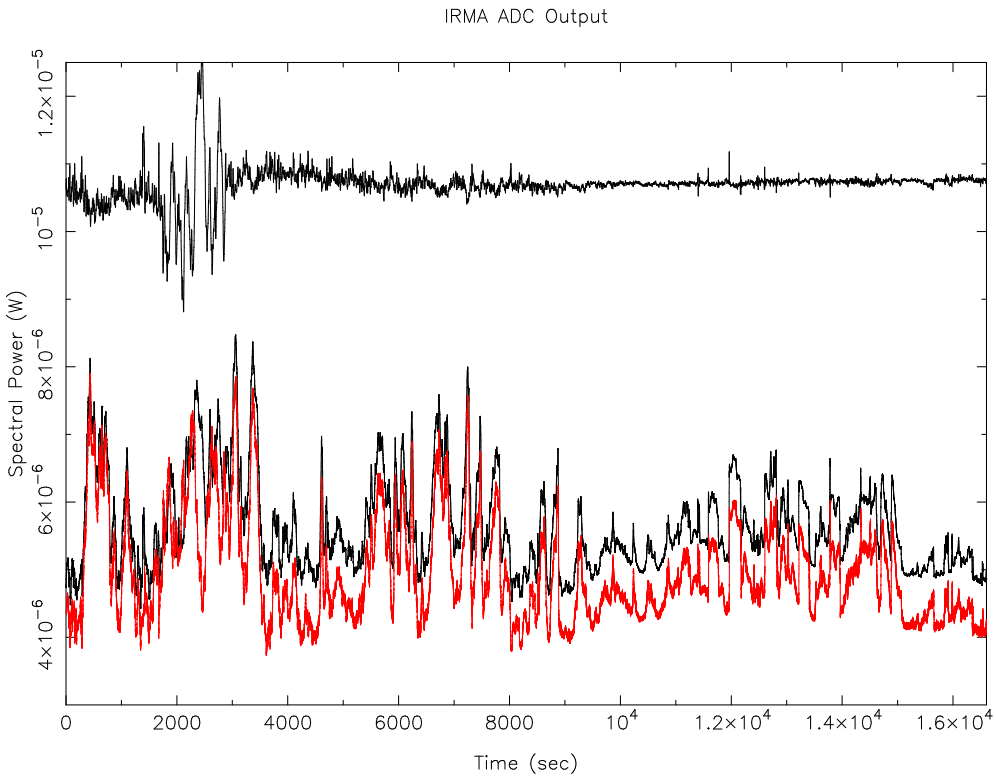


Figure 5.1: First set of simultaneous data taken by dual IRMA units at the Smithsonian Millimeter Array, Mauna Kea, Hawaii, June 15, 2004. This 4.5 hour data collection ran from 14:00 to 18:30 HST.

### 5.1.2 Gemini South

In February of 2005, two IRMA units were shipped to the Gemini South Observatory, atop Cerro Pachon in the Chilean Andes. A concrete pad was prepared for one IRMA unit, situated a few tens of meters from the Gemini telescope dome. IRMA's software systems went through extensive debugging, with special emphasis on it's Alt-Az software. The IRMA hardware and software operated as expected up until the final day of testing, when Ethernet communication problems began to occur, manifesting itself in extremely slow network transactions between the CP and MC. Pinging the MC from the CP showed packet losses ranging between 50 to 75 percent. The network communication problems appear to

have been solved after an alternate data/power umbilical cable was attached to the IRMA unit. The cause of this problem has yet to be determined. In the time that passed while IRMA lay idle, its vacuum had deteriorated, requiring its getter to be re-fired. The fragility of IRMA's hardware has been an ongoing problem. It is not unexpected, however, given that IRMA is still experimental.

## 5.2 Polar Deployment of IRMA

### 5.2.1 Antarctica

Over the past decade, attention has been directed towards Antarctica as a possible site for future astronomical observatories. Antarctica is attractive to astronomers for its low atmospheric water vapor content and low ambient temperature, both of which contribute to exceptionally low infrared sky brightness. Studies by Nggyen in 1996 show that the atmospheric thermal emission, centered around 2.36 microns, is darker than any other known observatory site, and is comparable to conditions 27 km above sea level[39]. Additionally, extremely low wind speeds have been reported at the Antarctic high plateau, which allows for optimal seeing conditions[58]. Seeing is inversely related to the amount of atmospheric turbulence; low turbulence results in higher observable angular resolution[25].

Particular attention has been given to Dome C, high on the Antarctic plateau some 3200 m above sea level[58]. The site is at sufficient elevation to place it above the 200-300 m band of turbulent atmosphere that extends from the Antarctic ice found at sea level. During the winter season, Dome C receives around 100-300 microns of precipitation, and has an average temperature of -60 C. Due to its extreme low temperatures and low

humidity, the atmosphere at Dome C has exceptionally low infrared darkness. In 2000, infrared brightness measurements of the wintertime atmosphere showed the site to be as much as 20 times darker than Mauna Kea in some regions of the 10 micron window.

Figure 5.2: Concordia Station, Dome C, Antarctica. The AASTINO remote observatory appears in the foreground as a green igloo[3].

Concordia station is a French/Italian research station located at Dome C. Unmanned, automated site testing stations have been placed at Concordia in order to characterize atmospheric conditions of the Antarctic high plateau. In 2003, an automated station called AASTINO (Automated Astrophysical Site Testing International Observatory), was deployed. Roughly the size of a travel trailer, AASTINO is a portable, autonomous, remotely-operated cabin housing numerous pieces of instrumentation. It relies on solar power as well as two Sterling cycle engines for power, and is connected to the Internet via an Iridium satellite telephone[28]. IRMA will arrive in Antarctica between November 2005 and January 2006, at the beginning of Antarctic summer. There it will spend a year on board AASTINO measuring water vapor content above Dome C. The IRMA detector box and Alt-Az mount will be mounted to the roof of AASTINO, while the command processor

PC will be housed inside the AASTINO cabin.

### 5.2.2 The Arctic

Characterization of the arctic environment for suitable astronomical observation sites is only in its infancy. Programs such as the European Southern Observatory's ESPAS[40] program (ESO Search of Potential Astronomical Sites) have been searching around the world for promising ground-based telescope sites. The arctic is an attractive region to locate observatories because it is more accessible than the Antarctic. Logistical infrastructure in the form of military and weather stations exist in northern Canada and Greenland. Like Antarctica, regions in the Arctic are extremely dry and cold. The Arctic winter is characterized by extremely calm, cold conditions with very little cloud cover; ideal observing conditions. Barbeau Peak, a 2616 m summit located on Canada's Ellesmere Island, has been identified by ESPAS as a candidate observatory location. Barbeau Peak is situated in an Arctic desert, receiving 18 mm of precipitation annually each winter (November through April). It has been suggested that the annual average night-time precipitation at Barbeau Peak may be close to 0 mm, making it even drier than Cerro Paranal, Chile (2635 m), considered to be one of the best observation sites on Earth[25].

### 5.2.3 Adapting IRMA to Polar Conditions

Long-range plans for IRMA include possible deployment in the Canadian arctic. Design of a completely autonomous, cold weather hardened IRMA is already underway. Deploying IRMA in the polar regions presents several challenges, in particular, the effect of the extreme cold on IRMA's moving parts. IRMA's chopper wheel axis bearing lubricant

becomes viscous at low temperature, which required that the chop wheel bearing be repacked with a new lubricant. The flex cable, which carries power and communication lines from the Alt-Az base to the IRMA detector box, becomes stiff when subjected to -80 C temperatures. Teflon-coated wires gathered into a bundle are being considered as a solution, as there is no suitable supplier for short lengths of Teflon flex cables.

Many conventional integrated electronic devices, such as single board computers, are not tested to function to specifications at temperatures lower than 0 C, although they may correctly operate at low temperature. At issue is the possibility that these devices may contain temperature sensitive components, in particular, certain types of capacitors, which will fail at sub-zero temperatures. Extended temperature range devices are verified to operate at temperatures as low as -40 C. The IRMA MC and AAC control computers are rated extended temperature operation. IRMA, however, must be able to survive temperatures as low as -90 C, the minimum expected temperature that could be experienced at Antarctica. Therefore, it has been necessary to perform environmental testing on the IRMA units destined for Antarctica. In the spring of 2005, the AIG research group acquired a large (292 l) environmental chamber[10] capable of reaching -86 C for low temperature testing of the IRMA MC and the Alt-Az mount..

Semiconductor electronics, such as the Rabbit mircocontroller modules, have been proved to operate normally under cold (-80 C) conditions in the freezer. Components containing electrolytic capacitors, such as the power supply and Maxon motor controllers have failed, as their capacitance drops with respect to temperature. The power supply for the cold temperature IRMA has been relocated to inside the AASTINO cabin (where it will

be within the operational temperature range of electrolytic capacitors). The electrolytic capacitors in the Maxon motor controllers have been replaced with tantalum capacitors, which can tolerate lower temperatures.

### 5.2.4 Remote Communications

The other challenge facing IRMA is its remote communication link. Tests have been performed using an Iridium satellite telephone to establish a serial PPP (point to point) connection between IRMA and a host computer at the University of Lethbridge. This link carries TCP/IP traffic, permitting a user to connect to IRMA as if it were another host on the Internet. Tests to dial into IRMA from a remote computer as well as from IRMA using the Iridium telephone have been successful. It is anticipated that IRMA operators will dial into IRMA to perform configuration or housekeeping tasks. For the majority of communication uplinks, IRMA will automatically dial out over the Iridium network and connect to a U of L based computer to transmit its science and housekeeping data. Network bandwidth is greater and less costly when dialing into Iridium's Internet service from an Iridium telephone, rather than directly dialing into a remote Iridium telephone. This is the method AASTINO uses to transmit its data to its operation center at the University of New South Wales (UNSW), in Sydney.

### 5.2.5 Migrating from 8-bit to 32-bit Embedded Computers

For true autonomous operation, IRMA will need to possess a greater degree of reliability and flexibility. This requires that operators have the option to log into IRMA regardless of IRMA's condition in order to manage the system, reconfigure (perhaps even re-

compile the IRMA source), and reset the IRMA software. The autonomous, remote version of IRMA is based around PC-104 small form factor computer hardware instead of Rabbit microcontrollers. Roughly 3.5 by 3.75 inches square, PC-104 computers are true IBM PC compatible computers capable of running desktop operating systems. The remote version of IRMA will run RedHat 9 (kernel version 2.4.20), permitting the IRMA master control software to be developed using conventional development tools and languages: ANSI C, using the GNU C/C++ compiler. All the PC-104 hardware selected for IRMA is all rated for extended temperature (-40 C to 85 C) range.

**Tri-M TMZ104 PC-104 Single Board Computer**


Figure 5.3: Tri-M TMZ104 PC/104 single board computer, powered by a 667 MHz Transmeta Crusoe 5500 CPU.

Based on a 667 MHz Transmeta Crusoe 5500 microprocessor, the Tri-M TMZ104

features 272 MB of SRAM, 1 USB 1.1 port, 2 RS-232 serial ports, 1 parallel port, 1 EIDE channel (supporting 1 master and 1 slave device) 1 keyboard port, 1 PS/2 mouse port, and a PC-104 16-bit expansion bus for connecting additional PC-104 modules. The TMZ104 is certified for operation at temperatures between -40 C to 85 C. Power consumption on the TMZ105 varies with the CPU workload. The CPU is configured to dynamically switch between 33 and 533 MHz, resulting in power consumption ranging between 1.8 and 1.93 W. The TMZ104 is manufactured by Tri-M Engineering[14].

#### Diamond Systems Emerald MM-DIO Serial/Digital IO Module

The Diamond Systems Emerald MM-DIO[9] is a 48 channel DIO card that also includes 4 RS-232 serial ports. The board uses the PC/104 form factor and interfaces to the TMZ104 via the 16-bit PC-104 bus. It is temperature rated for operation between -40 C to 85 C. Serial speeds up to 115 kbps are supported. All 48 DIO lines are bidirectional. Power consumption is set at 100 mA.

#### Aaeon PCM3660 10-BaseT Ethernet Module

The Aaeon PCM3660[18] is a 16-bit, 10 Mbit/s Ethernet module, based on the RealTek 8019 network interface chip (NIC). The 8019 is based on the Novel NE2000-compatible network interface chip. To use this network card under Linux, the system must be manually configured to load `ne.o` module, as this (as well as any other PC-104 card) is not plug-and-play, but rather, based on older-style ISA technology. The PCM3660 is not rated for extended temperature operation; it is designed to operate in temperatures between -15 C to 70 C. The PCM3660 consumes 400 mA.

### RTD CML16686GX333HR PC/104 CPU Module


Figure 5.4: RTD CML16686GX333HR PC/104 single board computer, featuring an on-board Ethernet controller. The computer is powered by a 333 MHz National Semiconductor Geode CPU.

Although the Tri-M TMZ104 CPU module is exceptional in its low power usage, it does not have on-board networking, which means that an additional PC/104 add-on network module must be added, adding to the bulk of the embedded computer. The RTD CML16686GX333HR[52] CPU module requires 6.3 W of power, but features a 10/100 Base-T Ethernet controller, as well as many of the features offered on the Tri-M module. The CPU is a National Semiconductor Geode GX-1 Intel-compatible processor clocked at 333 MHz; roughly half the speed as the Tri-M's Transmeta Crusoe processor. This CPU module is being considered as a replacement for the Tri-M board, because it along with the Diamond MM-DIO board can both fit in IRMA's electronics compartment.

#### M-Systems Disk-On-Chip 2000 Technology

A 576 MB M-Systems MD2203-D576[31] Disk-On-Chip 2000 (DOC) serves as the hard drive for the PC/104 version of IRMA. Having no moving parts makes the DOC highly desirable in a hostile (wet and cold) environments. The DOC is rated for operation at temperatures as low as -40 C. The main difficulty with using DOC storage is that it, being based on NAND-gate flash memory technology, can only tolerate between 100,000 and 1 million erase cycles. NOR-based flash memory, the most common type of non-volatile RAM used, can handle only a tenth of that – between 10,000 and 100,000 erase cycles[59] per memory cell. Fortunately, the DOC device driver supplied with the chip uses wear-leveling to spread read/write operations across the memory cell array. Despite this precaution, the number of erase cycles remains fixed, thus requiring judicious use of memory. Swap memory, which uses a section of the hard disk to store the state of suspended (swapped-out) processes, will be disabled, in order to reduce the amount of disk read/write activity. Consequently, IRMA's operating system will be configured to have a small memory footprint. This implies the use of a small, minimal kernel, using only necessary features. Linux is scalable, and should easily fit within the 272 MB memory space.

### 5.2.6 Porting Rabbit-based IRMA Software to the PC

The IRMA control software (excluding the Alt-Az software) is currently being ported over to the PC platform, which has required some significant structural changes. The CP software, along with the IRMA GUI (if required) will be hosted on the MC computer along with the MC software, eliminating the CP computer. Both programs will run in

separate processes (tasks), and communicate with each other over a local socket, which unlike the Ethernet link, is 100 percent reliable. The AAC will remain on the Alt-Az unit due to limited number of wires that can be contained in the flex cable. The Alt-Az controller is interfaced to 26 DIO lines (shown in figure 2.16) that handle motor control and position feedback. There is provision for a third computer, hosted on on a Rabbit RCM2200 microcontroller module, to monitor the solar power kit. The solar controller (SC) will communicate with the MC via a local Ethernet LAN using the IRMA network packet protocol and packet structure. No IRMA system planned for deployment requires a solar power unit or a Rabbit SC module.

Since the CP software is written in Perl under the Linux OS, it should require very little modification when ported over to the PC/104 platform, which will be also running Linux. Virtually no modifications are required for the CP software. It can communicate with the MC software using the PC/104 host's IP address, or by using the network loopback address. This is important as it will help ensure the Rabbit and PC IRMA software are compatible with each other. That is, the same CP software can be used with both platforms.

Porting the MC code, which was originally written in Dynamic C, will require more effort. The MicroC/OS-II tasks need to be translated over to UNIX-style processes. Hardware dependent system calls, such as DIO and serial communication, must be translated into the equivalent Linux system calls. Serial and digital I/O must be remapped from the Rabbit to the PC-104 hardware. The MC's data collection interrupt service routine (ISR) as defined on the Rabbit was anticipated to be rewritten as a Linux device driver. Fortunately, the Linux driver library for the Diamond MM-DIO board supports user-mode

interrupt functions, which are much simpler to implement (as they run in user memory space), and function in similar fashion to ISRs.

Finally, the Rabbit to PC-104 software port presents the opportunity to restructure the IRMA software to reduce (or hide) complexity and enhance readability. It may be advantageous to implement IRMA's modules and libraries as objects, implying that the IRMA software be rewritten in C++. One of the big problems identified in the current MC source code is the proliferation of globals, which is indicative of poor, or at least ad hoc design. Global variables and structures were used in the MC software as a means to pass data between tasks and hold system state. Under C or C++, a wide range of inter-process communication (IPC) mechanisms are available to the developer. Rethinking the design of the MC software may be a worthwhile exercise. Due to the limited time window available to deliver the Antarctic and Thirty Meter Telescope (TMT) IRMA units, the MC software is being rewritten in C in order to simplify and speed up the porting process.

## 5.3 Final Thoughts

The IRMA control system is by far the most complex instrument control system designed by the Astronomical Instrumentation group. Its code base adds up to roughly 25,000 lines spread across four autonomous software executables and three platforms: the PC-based command processor, the RCM2100-based master controller, the RCM2010-based Alt-Az controller, and the PC-based IRMA GUI front end written by Amy Smith. Use of a custom scripting language allows precise and flexible control of the IRMA instrument. The modules responsible for hardware control, namely the MC and AAC, are embedded

systems based on low-power, robust 8-bit microcontrollers containing no moving parts, making them well suited for use in hostile environments. Additionally, IRMA's hardware control modules deliver hard real-time performance by means a preemptive multitasking kernel. Future instrumentation designed by the University of Lethbridge's Astronomical Instrumentation Group will likely be influenced by IRMA for years to come.

# Appendix A

# IRMAscript

## A.1 Overview

When the operating specifications of IRMA were being established, it was decided early on that IRMA should be controlled not by a set of pre-defined operation sequences, as had been the case with the earlier incarnations of IRMA, but rather with a command language in order to provide the operator with the greatest amount of operational flexibility. This approach to device control is not uncommon with advanced systems. The Unidex[19] family of motion control units used with the AIG's Mach-Zehnder FTS (MZFTS) and Herschel/SPIRE Test FTS provide proprietary scripting languages to control their multi-axis motion controller.

Although time consuming at first, using scripts to control instrumentation allows the operator to define complex sequences in a file that can be executed at will. IRMA takes this approach. There is a GUI interface to do simple interactive tasks, and an interpreter to drive complex command sequences. Ultimately, everything in IRMA is based around scripts

and its native command language, IRMAscript. Each button and menu choice is mapped to a specific script, or generates a script dynamically, in order to define the behavior of the requested button click or menu selection. The IRMAscript interpreter, `irmaExec.pl`, is the primary interface between the operator and the IRMA instrument. All commands that IRMA responds to originate from this program.

## A.2 Language Structure and Features

IRMAscript is an interpreted language. That is, the language syntax is not converted into a primitive set of instruction codes before execution. Rather, each statement is extracted from its source file and tested for syntactic correctness and executed as they appear in the script. The process of interpretation results in programs executing slower than programs originally compiled into native machine code, due to the overhead of repeatedly converting human-readable computer language statements into machine-readable instructions (often re-interpreting the same statement over and over in the case of looped instructions). The IRMAScript language interpreter does not perform the actions defined in the IRMAscript statement, so speed of execution is not important – ease and flexibility of use, however, is.

An IRMAscript statement is structured simply. For commands that directly control IRMA, a command statement consists of a command type, followed by two modifiers, and zero to fifteen arguments. The command type and its two modifiers define a unique command. The arguments are provided in order to pass information pertinent to the command to IRMA. Most command statements, with the exception of the Alt-Az moveto/slewto

commands, have zero or one argument. In addition to IRMA commands, IRMAscript provides variables, data assignment, arithmetic, system commands (such as reading system time), looping mechanisms, lists, and flow control, and console I/O. They do not follow the same command structure described above.

Whitespace is used to delimit, or separate, each of the elements (command type, modifiers and arguments) that make up an IRMAscript statement. Whitespace can consist of spaces or tabs. Each statement must terminate with a carriage return. Only one statement can appear on one line, which precludes IRMAscript from being a free form language, such as C or C++. IRMAscript is case-less. It does not matter whether IRMAscript statements are written in upper or lower case letters. Within the interpreter, all statements are converted to uppercase.

Variables in IRMAscript are typeless since Perl, the language that IRMAscript is implemented in, is itself typeless. Type is determined by the context of the statement. For example, one would not want to perform arithmetic operations on textual data, such as a time/date string. Doing so will generate a runtime error and cause the currently running IRMAscript to break execution. Variables can have any name, including reserved words, but must be prefixed by a dollar sign '$'.

Numbers in IRMAscript, like in Perl, are real numbers. That is, they can be integer or floating point numbers, and be negative or positive. Literal numeric vales can be expressed as real numbers, just like in other languages. The only exception is that IRMAscript has no provision for scientific notation, nor can numbers be represented in different bases, such as hexadecimal or octal. Only base ten numbers are supported.

The range of numbers expressible in IRMAscript is based on the range of numbers expressible in Perl. In Perl, all numbers are represented internally as double precision floating point values. Thus, the range of numbers in IRMAscript is equal to the range of numbers expressible in double precision floating point numbers. The effective range of IEEE double precision floating point numbers is $\pm\ 10^{308.25}$.

Comments in IRMAscript are specified by placing a leading pound sign '#' at the beginning of the comment statement. For a block of text that needs to be commented, a pound sign must precede every line. There is no mechanism for multi-line comments such as /* ... */, as found in C, C++ or Java.

## A.3 IRMAscript Language Summary

The following table lists all IRMA system commands addressable within the IRMAscript language. Non-system commands, such as flow control commands, are not listed.

| **Command** | **Modifier 1** | **Modifier 2** | **Arguments** |
|---|---|---|---|
| STARTPROG | SOCKET | OPEN | |
| ENDPROG | SOCKET | CLOSE | |
| CRYO | STATE | ON | |
| CRYO | STATE | OFF | |
| CRYO | SET | MANUAL_MODE | |
| CRYO | SET | AUTO_MODE | |

| | | | |
|---|---|---|---|
| CRYO | SET | STOPPED_MODE | |
| CRYO | SET | SET_POINT | tempKelvin |
| CRYO | READ | COMP_AMP | |
| CRYO | READ | SET_POINT | |
| CRYO | READ | MODE | |
| CRYO | READ | CURR_TEMP | |
| CRYO | READ | OSC_FREQ | |
| CRYO | SERIAL | OPEN | |
| CRYO | SERIAL | CLOSE | |
| GPS | READ | DATE_TIME | |
| GPS | READ | EPOCH_TIME | |
| GPS | READ | LAT_LON | |
| GPS | SERIAL | OPEN | |
| GPS | SERIAL | CLOSE | |
| ADC | INIT | RESYNCH | |
| ADC | INIT | RESET | |
| ADC | INIT | RW_TEST | |
| ADC | SET | CSR | chan,gain,wordRate,polarity |
| ADC | SET | GAIN | channel, gainValue |
| ADC | SET | OFFSET | channel, offsetValue |
| ADC | SAMPLE | NO_INT | channel |

| | | | |
|---|---|---|---|
| ADC | SAMPLE | ON_INT | channel |
| ADC | READ | CSR | channel |
| ADC | READ | GAIN | channel |
| ADC | READ | OFFSET | channel |
| ADC | READ | CONFIG_REGISTER | |
| SHUTTER | STATE | OPEN | |
| SHUTTER | STATE | CLOSE | |
| SHUTTER | READ | LIMIT | |
| SHUTTER | READ | OVERCURRENT | |
| SHUTTER | SET | OC_RESET | |
| CHOP_MOTOR | STATE | ON | |
| CHOP_MOTOR | STATE | OFF | |
| CHOP_MOTOR | STATE | MEASURE_RPM_ON | |
| CHOP_MOTOR | STATE | MEASURE_RPM_OFF | |
| CHOP_MOTOR | READ | STATE | |
| CHOP_MOTOR | READ | RPM | |
| BB | STATE | ON | |
| BB | STATE | OFF | |
| BB | READ | STATE | |
| ALTAZ | MOVE_TO | DMS | elD,elM,elS,azD,azM,azS,spd |
| ALTAZ | STATE | POSLOG | poslog_enable/poslog_disable |

| | | | |
|---|---|---|---|
| ALTAZ | STATE | HALT | |
| ALTAZ | STATE | REBOOT | |
| ALTAZ | INIT | PING | |
| ALTAZ | INIT | ALTAZ | |
| ALTAZ | INIT | AXES | ELEVATION/AZIMUTH |
| ALTAZ | INIT | SERVO | |
| ALTAZ | INIT | MOTOR | |
| ALTAZ | SET | ALT_OFFSET | offset |
| ALTAZ | SET | AZ_OFFSET | offset |
| ALTAZ | READ | POSITION | |
| ALTAZ | READ | TASK_STATUS | |
| ALTAZ | READ | ALT_OFFSET | |
| ALTAZ | READ | AZ_OFFSET | |
| ALTAZ | READ | POSLOG_RANGE | |
| ALTAZ | READ | POSLOG_DATA | |
| ALTAZ | READ | POSLOG_STATE | |
| ALTAZ | SERIAL | OPEN | |
| ALTAZ | SERIAL | CLOSE | |
| ALTAZ | SLEW_TO | DMS | elD,elM,elS,azD,azM,azS,spd |
| RTC | SET | DATE_TIME | |
| RTC | READ | DATE_TIME | |

| | | | |
|---|---|---|---|
| RTC | SET | ARBITRARY_TIME | YYYY-MM-DDThh:mm:ss |
| SCAN | SIGNAL | ON_INT | |
| SCAN | SIGNAL | STOP | |
| SCAN | READ | STATE | |
| IRMA | STATE | OFF | |
| IRMA | READ | UPTIME | |
| SUN_SENSOR | READ | STATE | |
| SUN_SENSOR | READ | SHUTTER_STATE | |
| SUN_SENSOR | STATE | SHUTTER_OPEN | |
| SUN_SENSOR | STATE | SHUTTER_CLOSE | |
| NOTCH_FILTER | STATE | 60HZ_IN | |
| NOTCH_FILTER | STATE | 60HZ_OUT | |
| NOTCH_FILTER | STATE | 120HZ_IN | |
| NOTCH_FILTER | STATE | 120HZ_OUT | |
| NOTCH_FILTER | READ | 60HZ | |
| NOTCH_FILTER | READ | 120HZ | |
| BANDPASS_FILTER | STATE | IN | |
| BANDPASS_FILTER | STATE | OUT | |
| BANDPASS_FILTER | READ | STATE | |

## A.4 IRMAscript Language Definition

### A.4.1 List Manipulation

#### INITIALIZATION

Construct (initialize) a list with one or more elements.

**Example usage**

```
$angles = list 90 130.65 142.32 153.88 157.28 159.88 159.93
$fullname = list $firstName $middleName $lastName
```

#### LENGTH

Return the length of a list.

**Example usage**

```
$lstLen = listlength $someList
```

#### INDEX

Reference an element of a list, where the index ranges from 0 (the first element) to n.

**Example usage**

```
$x = substring $sourceString $index
$x = substring $sourceString 3
```

#### SUBSTRING

Retrieve a substring from a colon delimited data record. In IRMA commands that return multiple data items, such as `ALTAZ INIT PING`, data is returned as a colon delimited string. This command splits the data string into its constituent data items and returns the desired datum, based on an index value.

**Example usage**

```
$x = substring $sourceString $index
$x = substring $sourceString 3
```

### A.4.2 Utility Functions

#### DEG2DMS

Convert an Alt-Az coordinate expressed as floating point degrees into degree-minute-second (DMS) format. The Degrees, minutes and seconds must be variables because the deg2dms function places values in these variables. They are not input variables.

**Example usage**

```
deg2dms $angle $d $m $s
deg2dms 63.52 $d $m $s
```

#### STARTPROG SOCKET OPEN / ENDPROG SOCKET CLOSE

Open and close a TCP/IP stream socket connection to the IRMA master controller. If a script contains instructions to execute on the IRMA master controller, a network socket must be established to the IRMA MC, as low-level IRMA system commands and data flow over this connection. If a script does not contain IRMA hardware control commands, it is not necessary to wrap a script with these statements.

**Example usage**

```
startprog socket open
gps serial open
$currTime = rtc read date_time
gps serial close
endprog socket close
```

#### LOCALHOST

This command handles system functions performed by the host computer's operating system.

**localhost log open**
Open the log file. A log file name must be created using the `new log filename` command before logging can commence.

**localhost log close**
Close the log file.

**Example usage**

The example shown for the **new** command include examples of the **local log** commands.

**NEW**

The `new` family of functions creates new data items of various types, such as filenames and time stamps.

**new log filename**
Automatically generate and return a filename, and create a directory path for the new file. Filenames generated by this function follow the ISO time format:

**YYYY-MM-DDTHHmmSS.dat**

and end with the `.dat` extension. File paths follow the structure:

**/IRMAdata/IRMA_<boxNumber>/YYYY/YYYY-MM-DD**

where `/IRMAdata/` is a link (or filesystem shortcut) to some directory where IRMA data is stored, <**boxNumber**> is the IRMA unit's identifier number, **YYYY** is the year in which the data/log file was created, and **YYYY-MM-DD** is a year-month-day time stamp. This directory format organizes data files chronologically according to the particular unit.

**new iso timestamp**
Create a time stamp string conforming to the ISO date-time format:

**YYYY-MM-DDTHH:mm:SS.sss**

Where **YYYY** refers to year, **MM** to month (1-12), **DD** to day (1-31), **HH** to hour (0-23), **mm** to minute (0-59), **SS** to second (0-59), and **sss** to milliseconds (0-999). The symbols **-**, **T**, and **:** are delimitation symbols.

**Example usage**

```
$filename = new log filename
localhost log open $filename
repeat 500
   $timestamp = new iso timestamp
```

```
    $ch4 = ADC SAMPLE NO_INT 4
    print "1,\s,4,\s,$ch4,\s,$timestamp,\s,0,\s,0,\n"
    wait 60
endloop
localhost log close
```

### A.4.3 Variable Manipulation

#### ASSIGN

Assign a value to a variable. The source of the assignment can be literal or another variable. Literal values can be numeric or string. Strings can be defined with or without enclosing double quotes. When quotes are used, it is permitted to include whitespace in the string.

**Example usage**

```
assign $temp 4
assign $prevPos $currPos
assgin $date Jan-15-2005
assign $dateString "January 15, 2005 - 8:15 PM"
```

#### INCR / DECR

Increment or decrement a value contained in a variable. This operation does not work with literals, as literals cannot have values assigned to them.

**Example usage**

```
incr $cntr
decr $countDown
```

#### EVAL

Perform arithmetic operations and assign results to a variable. This command precedes a simple arithmetic statement involving two operands and one operator. The operations available are addition, subtraction, multiplication, division, modular division, and expo-

nentiation. The operands can be literals or variables, but the result must be assigned to a variable.

**Example usage**

```
eval $x = $y + 3
eval $diff = $prev - $curr
eval $weight = $mass * 9.81
eval $avg = $sum / $n
eval $z = $count % 256
eval $sqrt = 9 ^ 0.5
```

### A.4.4 Delays

#### WAIT

Delay execution of the script by $N$ seconds. $N$ can be a real value, ranging from 0 to some arbitrary value.

**Example usage**

```
wait $pauseValue
wait 30
```

### A.4.5 Flow Control

#### DO .. WHILE

While loops repeatedly execute a block of statements while some arbitrary condition is logically evaluated to be true. With **do .. while** statements, the condition is tested at the end of the block, as opposed to the beginning of the block, which occurs in **while** loops. A **do .. while** loop in IRMAscript opens with a **do** statement, and closes with a **while** ***condition*** statement. Any number of IRMAscript statements, including other **do .. while**

loops, can be included in this block. There is no limit to the number of **do .. while** loops that can be nested within one another.

The condition can take two forms: a simple comparison involving two operands, or a compound conditional statement that logically ands or ors two comparisons. For example, a simple conditional statement takes the form `$x < $y`, while a compound conditional is structured `$x < $y or $a = $b`.

Four kinds of comparison are available: less than `<`, greater than `>`, equality `=` and inequality `!=`. Logical anding and orring can be specified in a compound conditional using the symbols `and` and `or`. Do not use the symbols `&&` or || to perform logical evaluations.

**Example usage**

```
# simple conditional expression
do
    $x = cryo read curr_temp
    print "$x,\n"
    wait 1
while $x > 77

# compound conditional expression
assign $opened 1
assign $closed 2
do
    wait 2
    print "shutter_moving,\n"
    $x = shutter read limit
while $x != $opened and $x != $closed
```

### REPEAT .. ENDLOOP

Repeat execution of a block of statements. This structure is equivalent to a **for** loop that increments from 0 to $n$.

**Example usage**

```
assign $cnt 0
repeat 5
    print "$cnt,\n"
endloop
```

### GOTO

The most basic flow control mechanism is the goto statement. When the IRMAscript interpreter executes **goto *label*** statement, program control jumps to the IRMAscript statement immediately following the **label *labelName*** statement. Using gotos as a form of program flow control can lead to unstructured, unmanageable code. However, in the context of IRMAscript, whose scripts tend to be quite short (less than a printed page long), the issue of structured goto-less programming is not important. Given the relatively primitive flow control mechanisms available in IRMAscript, goto allows the programmer to develop sophisticated flow control within an IRMA script. With labels, the use of a colon after the label name is optional.

**Example usage**

```
assign $x 4
if $x < 12 and $x > 0
   if $x != 3
      if $x < 5
         goto DONE
      endif
      goto FAILURE
   endif
endif

label DONE:
   print "success!,\n"
   goto EXIT

label FAILURE:
   print "failure!,\n"
```

```
label EXIT:
```

### A.4.6 Input / Output Commands

#### PRINT

Feedback from an executing IRMAscript can be directed to the console (or shell) by means of the `print` command. The argument to the print command can be a literal or a variable. In its most simple form, `print` can accept bare literals, either text or numeric, which is inconsequential to IRMAscript, as it is a typeless language. If a string literal enclosed in double quotes is passed as the parameter, the user can format the output, mixing variables and literals together. The only stipulation is that each item in the string, whether literals or variables, must be separated by commas, and there must not be any whitespace between the quotes. The reason for the prohibition on whitespace is that the IRMAscript interpreter divides statements into their constituent parts (tokens) along whitespace divisions. Two special literals can be used within printf strings: The **\s** symbol defines a single whitespace, while the **\n** symbol defines a linefeed, and is often called a newline character.

Output can be directed to an open log file by including the `log` modifier immediately after the `print` command. The methods for defining the string format is identical to the standard `print` command. The `localhost` command has methods to open and close logfiles. Furthermore, the `assign` command can be used to define strings that can be assembled using the `print` command.

**Example usage**

```
print 345
print Word
```

```
print "Greetings!,\n"
print $timestamp
print "Time-Date:,\s,$hour,\s,$minute,\s,$second,\n"
print log "$ch,\s,$sample,\s,$timestamp,\s,$az,\s,$alt,\n"
```

### A.4.7 System Commands

The following group of commands are responsible for controlling and/or reading data from IRMA's hardware components, which includes the AAC.

#### NOTCH_FILTER

The **notch_filter state** [***filter***] commands enable or disable the 60 Hz notch filter. The filter is enabled with the `60hz_in` parameter, and disabled with the `60hz_out` parameter. Reading 60 Hz notch filter state can be done with the **notch_filter read 60hz** command. A return value of 0 (zero) indicates that the filter is not enabled, while a return value of 1 indicates that the filter is enabled.

**Example usage**

```
notch_filter state 60hz_in
notch_filter state 60hz_out
$x = notch_filter read 60hz
```

#### BANDPASS_FILTER

The **bandpass_filter state** [***in/out***] is used to enable or disable the 455 Hz bandpass filter, whose job is to filter out all frequencies above and below the 455 Hz chopper wheel frequency. The filter is enabled with the `in` parameter, and disabled with the `out` parameter. The state of the bandpass filter can be read with the **bandpass_filter read state** command. A return value of 0 (zero) indicates that the filter is not enabled, while a return value of 1 indicates

that the filter is enabled.

**Example usage**

```
bandpass_filter state in
bandpass_filter state out
$x = bandpass_filter read state
```

### SHUTTER

The command **shutter state** [***open/close***] signals the shutter control circuitry to respectively open or close the shutter. Once this command is issued, it cannot be aborted. The shutter will open or close until it has reached its destination position. Shutter condition during actuation can be read with the **shutter read limit** command. The following integer codes are returned: 3 - shutter is in the process of moving during shutter movement, 2 - shutter is closed (covering the optical aperture), and 1 - shutter is in the open position (optical aperture is exposed). Shutter jams can be detected by looking for an increase in the amount of current going to the shutter motor. The shutter overcurrent bit is set when this condition occurs. Calling the **shutter read overcurrent** statement returns the value of the overcurrent bit: 1 when the overcurrent condition exists, and 0 when it does not. When the overcurrent condition bit has been set, it must be reset to zero by calling the **shutter set oc_reset** command.

**Example usage**

```
assign $moving 3
assign $sClose 1
assign $sOpen  2

STARTPROG SOCKET OPEN

##############################
# set 40 second timeout period
##############################
```

```
$currTime = rtc read epoch_time
eval $timeout = $currTime + 40

SHUTTER STATE Open
do
   $x = shutter read limit
   print $x
   $currTime = rtc read epoch_time
   print "CURR_TIME:,\s,$currTime,\n"

   ################################################################
   # If shutter not opened within 40 seconds, assume that it
   # is either disconnected or jammed.  Reverse shutter direction
   # and exit...
   ################################################################
   if $currTime > $timeout
      shutter state close
      goto DONE
   endif

   wait 2
while $x != $sOpen

label DONE
ENDPROG SOCKET CLOSE
```

### BB

The **bb state** [***setting***] command enables or disables the blackbody shutter heater. The heater is turned on by calling this command with the argument **on**, while **off** turns the blackbody heater off. The state of the heater can be read by calling the command **bb read state**. The return value 1 indicates that the blackbody heater is on, while a return value of 0 (zero) indicates that it is off.

**Example usage**

```
bb state on
$x = bb read state
if $x = 1
    print "bb.heater.is.on,\n"
```

```
endif

bb state off
$x = bb read state
if $x = 0
    print "bb.heater.is.off,\n"
endif
```

### CHOP_MOTOR

The 450 Hz chop wheel is controlled and monitored by means of the **chop_motor** family of commands. The chop wheel is turned on or off by the **chop_motor state** ***setting*** command, where ***setting*** can be set to **on** or **off**. **chop_motor read state** reads the chop wheel status, returning the value 1 if the chop wheel motor is on, and 0 if it is off.

To read the chop wheel's angular speed in revolutions per minute (RPM), IRMA must first be set in angular speed measurement mode. In this mode, IRMA counts the number of interrupt pulses from the chop wheel over a selected period of time. Consequently, one cannot simultaneously perform a data collection scan and measure chop wheel angular speed. To perform a measurement, one turns the chop wheel on, then issues the command **chop_motor state measure_rpm_on** to put IRMA into angular speed measurement mode. The next step is to wait for a period of time, using the **wait** ***seconds*** command. The longer the time spent in angular speed measurement mode, the more accurate the average angular speed value will be. Wait periods ranging between 30 and 60 seconds provide adequate results. After the wait period has passed, one takes IRMA out of measurement mode with the command **chop_motor state measure_rpm_off**, then reads the resulting value with the command **chop_motor read rpm**.

**Example usage**

```
chop_motor state on
wait 15
chop_motor state measure_rpm_on
wait 45
chop_motor state measure_rpm_off
wait 2
$rpm = chop_motor read state
print "Motor_state:,$rpm,\n"
wait 3
$rpm = chop_motor read rpm
print "Motor_RPM:,$rpm,\n"
chop_motor state off
```

**RTC**

The IRMA MC's Rabbit 2100 microcontroller module contains a real-time clock (RTC) chip, from which the Rabbit obtains date-time information. The RTC chip uses 1980 epoch time, whereby time is calculated as the number of elapsed seconds since midnight, January 1, 1980. This is identical to how the Microsoft MS-DOS operating system calculates time, unlike UNIX or Linux, which uses the 1970 epoch as the basis for calculating time.

Current time on the MC's RTC is read with the command **rtc read date_time**. Returned is a colon-delimited string containing the current date time: ***year* : *month* : *day* : *hour* : *minute* : *second***. As an example, February 12, 2005, at 3:37:49 PM would be returned as **2005:2:12:15:37:49**. Months range from 1 to 12, days range from 1 to 31, hours range from 0 to 23, and minutes and seconds range from 0 to 59.

If the IRMA MC RTC is not set, date-time will default to the epoch time of 0, or January 1, 1980, 00:00:00. Date-time can be set in two ways: either by providing a user-defined date-time string, or by using the global positioning system (GPS) receiver's date-time, the former method being the most accurate.

Current time in 1980 epoch format can be read using the **rtc read epoch_time**

command. This command is convenient for timing events within an IRMA script, as it returns a 32-bit unsigned integer number representing the current time as the number of elapsed seconds since midnight of January 1, 1980.

User-defined date-time can be set using the the command **rtc set arbitrary_time *ISOtimeString***. An ISO formatted date time string has the following format: ***YYYY-MM-DD*T*hh*:*mm*:*ss***, where **YYYY** is a four-digit year, **MM** is month (1 - 12), **DD** is day-of-month (1-31), **hh** is hour (24-hour format), **mm** is minute, and **ss** is second. The punctuation contained in this format (the **T** and dashes **-**) must be left as shown.

The second method of setting date-time, using the GPS receiver, requires that the serial channel to the GPS board be opened. Not doing so will result in the call to set the RTC to timeout and fail. Once the serial channel has been opened, the command **rtc set date_time** will read the current date time from the GPS receiver, convert it to 1980 epoch format, and write it to the RTC. The GPS emits a time synchronization signal every second. Date-time is written to the RTC as soon as this time synch signal goes high. One concludes the RTC setting session by closing the serial channel to the GPS.

**Example usage**

```
# setting the RTC using GPS date-time
gps serial open
rtc set date_time
$x = rtc read date_time
print "$x,\n"
$y = rtc read epoch_time
print "$y,\n"
gps serial close

# setting the RTC with arbitrary time string
rtc set arbitrary_time 2005-01-20T15:37:45
wait 3
$x = rtc read date_time
```

```
print "$x,\n"
```

### GPS

The GPS family of commands involves the reading of time-date and location information from the IRMA MC's GPS receiver board. The GPS is interfaced to the MC by means of a 4800 bps serial channel. Consequently, all commands to the GPS must be preceded by issuing the command to open the GPS serial channel: **gps serial open**. After the transaction with the GPS has been completed, the GPS serial channel should be closed using **gps serial close**.

Date time is read from the GPS receiver using the command **gps read date_time**. The data returned is contained in a colon-delimited string: ***year*** **:** ***month*** **:** ***day*** **:** ***hour*** **:** ***minute*** **:** ***second***. Epoch time, returned in 1980 epoch format, is read by calling **gps read epoch_time**. Latitude-longitude data is read with the command **gps read lat_lon**. Data is returned as a colon-delimited string:

**Example usage**

```
gps serial open

# read date-time
$dateTime = gps read date_time
$year = substring $dateTime 0
$mon  = substring $dateTime 1
$day  = substring $dateTime 2
$hour = substring $dateTime 3
$min  = substring $dateTime 4
$sec  = substring $dateTime 5

# read epoch time
$epochTime = gps read epoch_time

# read IRMA's latitude & longitude
$latLon = gps read lat_lon
```

```
gps serial close
```

### IRMA

This family of commands is used to perform system-level activities on the IRMA system as a whole. The statement **irma state off** forces the IRMA MC software to reboot. The statement **irma read uptime** returns the number of elapsed seconds since the IRMA MC was powered up or last rebooted. The value returned by this command is represented in floaing-point seconds.

### Example usage

```
# read IRMA MC uptime
$uptime = irma read uptime
print "UPTIME,$uptime,\n"

# Reboot MC
irma state off

# Wait for reboot to finish
wait 10

# Read uptime again
$uptime = irma read uptime
print "UPTIME,$uptime,\n"
```

### SUN_SENSOR

The solenoid-controlled shutter protecting the filter and IR detector can be controlled in software using the **sun_sensor** commands. The state of the sun shutter is read using **sun_sensor read shutter_state**. A return value of 0 indicates that the shutter is closed (covering the filter and detector), while a value of 1 indicates the shutter is open. A photo cell coupled with discrete logic automatically closes the sun shutter when IRMA's line of sight comes within ± 15 degrees of the sun (or any bright light source), is read using the

command **sun_sensor read state**. A return value of 1 indicates that the sun sensor is detecting a bright light source in its line of sight. A value of zero indicates the opposite.

**Example usage**

```
sun_sensor state shutter_open

assign $shutterClosed "Sun shutter is closed"
assign $shutterOpen "Sun shutter is open"
assign $sunInView "Sun is within 10 degrees of view"
assign $sunNotInView "Sun is not in view"

$x = sun_sensor read shutter_state
if $x = 0
    print "$shutterClosed,\n"
endif
if $x = 1
    print "$shutterOpen,\n"
endif

$x = sun_sensor read state
if $x = 0
    print "$sunNotInView,\n"
endif
if $x = 1
    print "$sunInView,\n"
endif

sun_sensor state shutter_close
```

### CRYO

The Stirling engine (cryo cooler) that cools the IR detector is controlled by the **cryo** family of commands. Before attempting to send commands to the cyro cooler, the serial communication channel to the cooler must be opened with the command **cryo serial open**. Likewise, the channel is closed with the command **cryo serial close**.

**cryo read comp_amp**
Returns the compressor amplitude value as a floating point value.

**cryo read set_point**
Returns the cryo cooler's set point temperature in degrees Kelvin. The return value is a floating point number.

**cryo read mode**
Returns an integer code representing the operational mode of the cryo cooler controller.

**cryo read curr_temp**
Returns the current temperature in degrees Kelvin of the cryo cooler's cold finger. The return value is a floating point number.

**cryo read osc_freq**
Returns the cryo cooler's oscillation frequency, which is the frequency of the piston inside the cold finger. The oscillation frequency is expressed in cycles per second (Hz).

**cryo set manual_mode**
This command sets the cryo cooler into manual mode, which powers the cryo cooler down.

**cryo set set_point** ***temperature***
This command sets the desired temperature of the cryo cooler's cold finger. This command will successfully execute only when the cryo cooler is in **manual_mode**.

**cryo set auto_mode**
The cryo cooler begins to cool when this command is received. Cooling is a gradual process, taking roughly 30 minutes according to the cryo cooler controller's internal configuration settings. When target set point temperature is reached, the controller will maintain this temperature as long as it is in **auto mode**.

**Example usage**

```
# cryo_cooler_demo.irma
#######################

startprog socket open
cryo serial open

# define temperature set point
assign $setpoint 70
cryo set set_point $setpoint
$x = cryo read set_point
print "Set_point:$x,\n"
```

```
# start cooling cycle
cryo set auto_mode
$x = cryo read mode
print "mode:,$x,\n"

do
    $currTemp = cryo read curr_temp
    print "current_temperature:,$currTemp,\n"
    wait 10
while $currTemp > $setpoint

print "cryo_cooler_at__target_temperature,\n"

# keep cryo at target temperature for 10 hrs
$endTime = rtc read epoch_time
assign $tenHours 36000
eval $endTime = $endTime + $tenHours
do
    wait 60
    $compAmp = cryo read comp_amp
    $currTemp = cryo read curr_temp
    $oscFreq = cryo read osc_freq
    $currTime = rtc read epoch_time
while $currTime < $endTime

# power down cryo cooler
cooler set  manual_mode
$x = cooler read mode
print "mode:,$x,\n"

cryo serial close
endprog socket close
```

### ADC

Control of the Cirrus CS5534 Delta-Sigma ADC is handled by the **adc** family of commands. Before A/D conversions can be performed, the ADC must be first initialized using the **resynch** command, **adc init resynch**, then reset using the **reset** command, **adc init resynch**. The last step involves configuring each of the ADC's four channels with the

command: **adc set csr *arguments***. The following list describes each of the CS5534 IRMAscript functions in depth.

**adc init resynch**
Calling this command puts the ADC's serial port into a known state. When using the ADC for the first time, it is recommended that this command be called in order to ensure that the ADC will successfully accept serial commands. At low level, this command serially writes 15 bytes of the value 0xFF, followed by 1 single byte valued 0xFE.

**adc init reset**
This command resets the ADC and sets its fundamental parameters. At low level, the reset command sets the **RS** bit in the CS5534's configuration register, which has the effect of forcing a system reset.

**adc set offset *channel value***
**Set offset** command allows the user to configure each of the CS5534's four input channels' offset registers. Channels 1 through 4 can be specified, while the value field can accept offset values ranging between $-2^{23}$ and $2^{24}$. The offset value represents the fraction of the input span that must be applied to the output value of the ADC to shift it up or down. Offset values must be defined in ADC units. For example, an offset of 255 refers to a positive offset of $255/2^{24}$ of the ADC 's input span. ADC channels configured for taking unipolar samples have an input span of $2^{24}$, while channels configured for bipolar mode have an input span of $2^{23}$[8]. The CS5534's default offset setting is 0.

**adc set gain *channel value***
Similar to the **set offset** command, set gain allows the user to manually set a gain value, ranging from 64 to $2^{-24}$, to channels 1 through 4. When a channel's gain register is set, the offset is subtracted from the A/D sample value, after which this result is multiplied with the the gain value. IRMA currently does not use custom gain settings. Instead, gain and offset are applied to the the data in post processing. The CS5534's default gain value is 1.

**adc read gain *channel***
**adc read offset *channel***
These two commands read the current gain and offset values from the CS5534 ADC.

**adc set csr *channel gain word-rate polarity***
Each of the CS5534's four input channels can be configured in terms of signal gain, accuracy (word rate) and input span (polarity). Channel settings are stored in the CS5534's four channel setup registers (CSR). Gain as defined in the CSR is separate from the gain contained in the channel gain registers described

earlier. Gain values can be defined with the IRMAscript constants or their respective numeric values, as shown in table A.2.

| **Gain** | **Value** |
|---|---|
| CS5534_GAIN_1 | 1 |
| CS5534_GAIN_2 | 2 |
| CS5534_GAIN_4 | 4 |
| CS5534_GAIN_8 | 8 |
| CS5534_GAIN_16 | 16 |
| CS5534_GAIN_32 | 32 |
| CS5534_GAIN_64 | 64 |

Table A.2: CS5534 ADC gain settings in IRMAscript.

Resolution refers to the number of noise free bits contained in the A/D sample value. The longer the ADC integrates the analog signal, the greater the accuracy (or resolution) of the digitized sample. Table A.3 lists the different sample resolutions in terms of noise-free resolution bits, integration time (in milliseconds), and word-rate. Input span of digitization can be either unipolar, where A/D values contain values ranging from 0 to $2^{24}$ - 1, or bipolar, which allow signed values ranging from $-2^{23}$ to $2^{23}$ -1. Table A.4 lists constants and their respective numeric values can be applied to the polarity field.

| **Resolution** | **Bits** | **Integration** | **ms** | **Word rate** |
|---|---|---|---|---|
| CS5534_RES_23 | 23 | CS5534_INTEG_538 | 538 | 7 |
| CS5534_RES_22_SLOW | 22 | CS5534_INTEG_269 | 269 | 15 |
| CS5534_RES_22_FAST | 22 | CS5534_INTEG_136 | 136 | 30 |
| CS5534_RES_21_SLOW | 21 | CS5534_INTEG_69 | 69 | 60 |
| CS5534_RES_21_FAST | 21 | CS5534_INTEG_35 | 35 | 120 |
| CS5534_RES_18 | 18 | CS5534_INTEG_19 | 18.2 | 240 |
| CS5534_RES_17_SLOW | 17 | CS5534_INTEG_10 | 9.9 | 480 |
| CS5534_RES_17_FAST | 17 | CS5534_INTEG_6 | 5.7 | 960 |
| CS5534_RES_16 | 16 | CS5534_INTEG_4 | 3.6 | 1920 |
| CS5534_RES_13 | 13 | CS5534_INTEG_2 | 1.5 | 3840 |

Table A.3: CS5534 ADC sample resolution settings in IRMAscript.

.

| Polarity | Value |
|---|---|
| CS5534_UNIPOLAR | 1 |
| CS5534_BIPOLAR | 2 |

Table A.4: CS5534 ADC polarity settings in IRMAscript.

.

**adc read csr *channel***
The contents of each of the CSR channels can be read using this command. A colon-delimited string having the following format is returned:

***channel* : *gain* : *word-rate* : *polarity***

**adc init rw_test**
Primarily used for troubleshooting and verification, the read-write test command tests the ADC to ensure that the IRMA software can communicate with it. An arbitrary value is written to one of the CS5534's offset registers, then that value is read back from the offset register. If the two values are identical, the test is deemed a success, and a value of 1 is returned. A failed read-write test returns a 0 (zero). This command should be followed with a ADC system reset in order to clear the dummy value in the offset register.

**adc sample *sample_type channel***
This command initiates an A/D sample on a given ADC channel. The returned value is given in ADC units, thus it must be interpreted according to the channel's polarity setting: bipolar or unipolar. Two types of samples can be taken: those synchronized to the 450 Hz chop wheel, specified with the **on_int** parameter, or samples not synchronized to the chop wheel, specified with the **no_int** parameter. When the **on_int** parameter is specified, the A/D sample commences when the chop wheel signal (mapped through the Rabbit interrupt channel) reports a logic level of 1. The **channel** parameter is mapped to 11 separate channels, 8 of which are multiplexed into ADC channel 4.

**Example usage**

```
###############
# adc_demo.irma
###############
startprog socket open
```

| Channel | Description |
|---|---|
| 1 | IR Signal |
| 2 | Humidity |
| 3 | Atmospheric Pressure |
| 4 | Temp1: Blackbody Shutter |
| 5 | Temp2: Blackbody Shutter |
| 6 | Temp3: Mirror Base |
| 7 | Temp4: ADC |
| 8 | Temp5: Base Compartment |
| 9 | Temp6: Pump |
| 10 | Temp7: Shutter Motor |
| 11 | Temp8: Pre-amp |

Table A.5: ADC channel usage on the IRMA MC.

```
# initialize ADC
adc init resynch
adc init reset

# perform read-write test
$x = adc init rw_test

if $x == 0
    goto ADCFAILURE
endif

# configure channel setup registers
adc init reset
adc set csr 1 1  CS5534_INTEG_538 CS5534_UNIPOLAR
adc set csr 2 8  CS5534_INTEG_4   CS5534_UNIPOLAR
adc set csr 3 16 CS5534_INTEG_4   CS5534_UNIPOLAR
adc set csr 4 1  CS5534_INTEG_4   CS5534_BIPOLAR

# configure offset and gain on channel 1
adc set offset 1 5000
adc set gain 1 12
$ch1offset = adc read offset 1
$ch1gain = adc read gain 1
print "$ch1offset:,$ch1gain,\n"

# read back channel setup registers
assign $ch 1
```

```
do
    $csr = adc read csr $ch
    $chan = substring $ch 0
    $gain = substring $ch 1
    $wordrate = substring $ch 2
    $polar = substring $ch 3
    print "CHAN:,\s,$chan,\n"
    print "GAIN:,\s,$gain,\n"
    print "WORDRATE:,\s,$wordrate,\n"
    print "POLARITY:,\s,$polar,\n"
    incr $ch
while $ch < 5

# read all the ADC channels
assign $ch 1
do
    $sample = adc sample no_int $ch
    print "$ch,\s,$sample,\n"
    incr $ch
while $ch < 12

label ADCFAILURE:
endprog socket close
```

**SCAN**

The scanning process involves repeatedly sampling the IR signal and temperature / pressure / humidity channels at some interval. Scanning differs from reading an ADC channel directly in that the A/D sampling process is contained separate real-time task, and uploads the data to a separate network port on the IRMA CP. This allows the MC to service other commands while the data collection process is executing, such as moving the Alt-Az mount, or querying the status of the cryo cooler. The Alt-Az serial communications channel must be opened before executing the scan command. Additionally, it is vital that the Alt-Az channel be left open for the duration of the scan. Closing the channel during a scan will lead to scan failure, which results in the scan terminating itself.

**scan read status**

Returns the value 1 if a scan is currently executing on the MC, otherwise the value 0 is returned.

**scan signal on_int**
Forks the data collection process task, where the IR signal is sampled on the positive edge of the notch notch interrupt signal. Temperature, pressure and humidity channels are each sampled following one IR signal sample in a round-robin fashion.

**scan signal no_int**
IR signal, temperature, pressure and humidity are sampled, but the IR signal A/D conversion is not synchronized to the notch interrupt.

**Example usage**

```
################
# scan_demo.irma
################
startprog socket open

# check if a scan is already running.
# if scan isn't running, scan read state
# will return 0
$x = scan read state
if $x == 1
    goto ENDOFSCRIPT
endif

# configure ADC
adc init resynch
adc init reset
adc set csr 1 1 7 1
adc set csr 2 1 1920 1
adc set csr 3 1 1920 1
adc set csr 4 1 1920 1

# open serial port and *leave* it open!
altaz serial open

# fork the scan task
scan signal on_int

label ENDOFSCRIPT:
endprog socket close
```

### ALTAZ

Movement and control of the altitude and azimuth axes is handled by the **altaz** family of commands. Given that the AAC is connected to the MC over a serial communications link, the AAC-MC serial connection must be opened before any altaz command can be sent. Failing to open the serial port when sending AAC commands produces subtle errors that are hard to track down. In executing a sequence of commands within a single file, the error is more direct: the command string destined for the AAC will not get transmitted. However, the current implementation of IRMA is typically controlled via multiple single-command scripts that are generated on-the-fly by the IRMA GUI, whereby a single command is contained in its own script. This results in the incomplete transmission of command strings, leaving the remaining characters in the serial buffer. The operator would observe that the IRMA MC sends the entire command string, yet the IRMA AAC only reads a portion of the string, hangs, then times out once the five second timeout period has expired. If the operator sends another command, the AAC receives a corrupted string, because it contains the new command plus the fragment characters left over from the last command. There are five subgroups of commands within the altaz command family: Alt-Az mount initialization commands, parameter setting commands, parameter/status reading commands, movement commands, and operation mode commands. Each of the altaz command groups will be examined in detail.

Commands destined for the AAC are sent over the MC/AAC serial link using the serial packet communications protocol. This protocol is discussed in depth in section 3.6.2, and includes a discussion of error codes that result due to serial transmission errors.

**altaz serial open**

**altaz serial close**
These commands respectively open and close the serial channel from the MC to the AAC.

**altaz init altaz**
Once the Alt-Az serial channel has been opened, the first command that should be sent to the AAC is the **init altaz** command. This command has the effect of initializing the AAC's two-channel optical encoder chip that is responsible for digitizing axis encoder positions. Upon initializing the optical encoder chip, altitude and azimuth axis positions are set to 90,000 encoder units. There are 8192 encoder units in one revolution.

**altaz init axes *axis***
Upon using the AAC for the first time, or where re-initialization is required, the axes should be sent to their default positions. The **init axes** command performs a homing operation, whereby it determines the clockwise and counter-clockwise optical limits on both axes. When axis homing has completed, altitude and azimuth positions are set to position 0 (in encoder units). The axis parameter can be defined in three ways: ***altitude***, ***elevation***, or ***azimuth***.

**altaz init motor**
The gearboxes and motor controllers used in the IRMA AAC differ from unit to unit. In order to deal with these variations, gear ratios, motor RPM values, and other configuration information unique to the given IRMA unit is contained in a configuration file. By issuing this command, the CP uploads motor configuration information, stored in the particular IRMA unit's configuration file, into the AAC. Without this information, the AAC cannot calculate motor speeds or slewing times. Therefore, it is vital that **init motor** be called before any axis movement is attempted.

**altaz init servo**
The AAC uses a servo loop, based on proportional-integration-derivative (PID) motion control algorithm, to control axis movement. PID servo control algorithm has three constants, **P**, **I** and **D**, which are unique to each Alt-Az mount. The **init servo** command loads the PID constants for the altitude and azimuth axes into the AAC. If the servo-controlled movement **move_to** command is going to be used, servo parameters must be loaded into the AAC beforehand. The **slew_to** non-servo movement command does not require servo parameters to be set. The command **init servo** may be called while the AAC is idle (not moving) as many times as required, which is particularly helpful if the user is "tuning" the servo algorithm.

**altaz init ping**
The Alt-Az **ping** command is used to check if the AAC is on-line, ready to receive commands. If the AAC is alive and on-line, it returns a three-field, colon delimited string of the form:

***987654321*** : ***123456789*** : ***uptime***

If the AAC is not on-line or unresponsive, the three fields will contain the code **999999999**. The **Uptime** field indicates the number of elapsed CPU ticks since the IRMA MC was booted. Each CPU tick is 1/64 seconds, therefore to convert this value to elapsed seconds, divide it by 64. Uptime can also be read using the command **altaz read uptime**.

**altaz set alt_offset** ***offset_value***
**altaz set az_offset** ***offset_value***
The **set** ***offset*** commands are provided in order to allow the user to define virtual fiducial points, thus avoid the necessity of physically orienting IRMA's fiducial (the axis limits) to external physical references, such as zenith for elevation, or north for azimuth. By providing an offset value defined in optical encoder units, IRMA's AAC calculates all axis moves relative to the offset position instead of the default physical limit. Axis offsets is the angle between the physical limit and the position where the physical reference is determined to be. The default offset value for both axes is zero.

**altaz state poslog** ***action***
AAC position logging is controlled using this command. Three separate activities can be performed: log initialization, log enabling and log disabling. Upon AAC start-up, the position log is allocated, zero-filled, and its index pointer is pointed to the first element in the position log array. This action should be explicitly called before using the position log by using the **log_clear** constant in the action parameter. One begins logging an axis movement by calling this command using the **log_enable** constant. Logging is stopped by using the **log_disable** constant.

**altaz state halt**
Stop movement immediately in both axis.

**altaz state reboot**
Perform a soft reset (or reboot) of the AAC software running on the Alt-Az controller. The master controller software is not affected.

**altaz move_to** ***axis alt_d alt_m alt_s az_d az_m az_s speed***
Servo-controlled movements, which track a theoretical velocity versus position profile, are performed using the **move_to** command. Three parameters must be provided: the axis to be moved, the destination angle, and the axis rotation speed, specified in degrees per second. Options available for **axis** include: **altitude**, **azimuth**, and **dualaxis**.

The destination angle is defined in degree-minute-second format, where altitude degrees, minutes and seconds occupy fields 4, 5 and 6 respectively (assuming field 1 refers to the "**altaz**" symbol). For single-axis movement, altitude or

azimuth destinations should be written to fields 4, 5 and 6, while fields 7, 8 and 9 should be zero-filled. For dual-axis movements, altitude should occupy fields 4, 5 and 6, and azimuth should occupy fields 7, 8 and 9. Field 10 is populated with the desired axis speed. In the case of dual-axis movement, the speed refers to the diagonal speed between the two moving axis, or rather, the speed required for both axes to meet at the final altitude/azimuth coordinate.

Since this is a servo-controlled move command, movements are continuous, and are consequently limited to the speed options provided by the given Alt-Az mount's gearing. The slowest speed possible with this command occurs when the axis motor is driven at 0 volts, which corresponds to 500 motor RPM. The axis will rotate considerably slower than the minimum motor RPM, due to the motor's gear box and drive belt. Be aware that it is impossible to perform movements slower than the minimum motor RPM. For performing movements slower than the minimum achievable speed, there is the **slew_to** command.

**altaz slew_to dms *axis alt_d alt_m alt_s az_h az_m az_s speed***
Usage of the **slew_to** command is identical to **move_to**. What differs is the range of speeds available, and the fact that movement is not servo controlled. When a speed less than the minimum achievable speed is selected, **slew_to** goes into stepping mode, where the given slew path is broken up into sub-degree, one encoder unit steps. The axis (or axes) rotate for the duration calculated from the slew path length and the requested speed.

Mention should be made about the relationship between offset angles and axis moves. Offset angles for each axis are measured from the counterclockwise limit switch in the clockwise direction. The AAC rotates to the requested angle, to which is added the currently defined offset angle. The offset angle should be considered as zero degrees. Altitude angles less than the offset angle are reported as negative angles, while azimuth angles less than the offset wrap at 360 degrees, because the azimuth axis has the ability to rotate a full 360 degrees.

Full rotational movement allows for the possibility of destination angles that lie beyond the far (clockwise) limit. In such cases, the AAC rotates the azimuth axis in the opposite direction to the target angle lying beyond the far limit. In the case of low-speed, small-distance movements that result in destinations crossing the rotational limit, the AAC drives the axis to the destination angle in the opposite direction at high speed in order to eliminate the annoyance of slewing nearly 360 degrees as low speed.

**altaz read position**
This command returns a three-value colon-delimited string containing altitude and azimuth value respectively. The third field contains scan status: 1 when a scan is executing, and 0 when no scan is running.

***altitude*** : ***azimuth*** : ***scan_status***

**read position** is the most common query request to the AAC because during scans, the MC requests axis positions for each data point collected.

**altaz read task_status**
In order to remain responsive to incoming commands, the AAC executes axis movements separate from the main dispatcher task. **read task_status** allows external processes, such as an executing IRMA script, to check up on an ongoing AAC movement, and determine when the operation has completed. Task status is returned as one of three codes: code 0 indicates there is no axis movement task operating, while code 2 indicates a task is executing. Code 1 is returned when the AAC is dispatching a long-duration job to one of its available tasks. It is rare that this code would be encountered, and should be considered simply as a running task.

**altaz read alt_offset**
**altaz read az_offset**
These two commands respectively return the currently defined altitude and azimuth offset values in optical encoder units. There are 8192 units per revolution.

**altaz read poslog_state**
The **poslog** commands are used primarily for Alt-Az servo tuning. They allow the user to collect axis motion data necessary for tuning the AAC's PID servo control loop. The **read poslog_state** command returns the current operation mode of the position log, the table in the AAC that is used to store servo and position data. Three states can be reported: code 1 indicates the position log is enabled. Code 0 indicates the position log is disabled. Code 2 is returned if the position log was not initialized during AAC start-up. This can happen if an extended memory allocation failure occurred on board the AAC Rabbit processor.

**altaz read poslog_range**
Calling this command returns the dimensions of the position log, a memory array aboard the AAC containing position and servo data. A four field colon-delimited string is returned:

$$\textit{min array index} : \textit{max array index} : \textit{curr array index} : \textit{NULL}$$

The range of data readable from the AAC's position log is found between the minimum array index and the current array index inclusive. Reading values beyond the maximum array index will result in a memory read error on the AAC.

**altaz read poslog_data *index***
Given some index value, this command returns the position log entry at that index. A four field, colon-delimited array is returned:

***DAC val*** : ***rel pos*** : ***theor pos*** : ***error val***

**DAC val** contains the 8-bit unsigned integer that is written to the AAC's DAC, which in turn controls axis speed. **Rel pos** refers to the actual position of the axis relative to its start position, and is given in optical encoder units. **Theor pos** is the calculated theoretical axis position, also given in optical encoder units. It is this theoretical displacement path that the PID servo must track. The last field, **error val**, contains the PID algorithm error value. All four data are necessary in the servo tuning process.

**Example usage**

```
#################
# altaz_demo.irma
#################
startprog open socket

altaz serial open

assign $running 2

# ping the AAC
$status = altaz init ping
print "ping.status:,$status,\n"
$a = substring 0 $status
$b = substring 1 $status
$c = substring 2 $status
if $a != 987654321
    if $b != 123456789
        if $c != 181818181
            goto DEAD_AAC
        endif
    endif
endif


# initialize motor, servo, then optical encoder chip
altaz init motor
altaz init servo
altaz init altaz

# find optical limits on elevation axis
altaz init axes altitude
wait 4
do
```

```
    $x = altaz read task_status
    print "$x,\n"
    wait 2
while $x == $running

wait 2

# then on azimuth axis
altaz init axes azimuth
wait 4
do
    $x = altaz read task_status
    print "$x,\n"
    wait 2
while $x == $running

# set elevation and azimuth offsets
altaz set alt_offset 183
altaz set az_offset 5234
$a = altaz read alt_offset
$b = altaz read az_offset
print "offsets:,$a,$b,\n"

# move elevation and azimuth axes to 5.5 degrees
# at 3 degrees per second
altaz move_to dms dualaxis 5 30 0 5 30 0 3
do
    wait 2
    $status = altaz read task_status
    print "status:,$status,\n"
    $pos = altaz read position
    $altPos = substring 0 $pos
    $azPos = substring 1 $pos
    print "position:,$altPos,$azPos,\n"
while $status == $running

# move elevation axis to zenith @ 2 degrees per second
altaz move_to dms altitude 90 0 0 0 0 0 2
do
    wait 2
    $status = altaz read task_status
    $pos = altaz read position
    $altPos = substring 0 $pos
    print "elevation:,$altPos,\n"
```

```
while $status == $running

# step azimuth 50 degrees @ 0.05 degrees/sec
altaz slew_to azimuth 50 0 0 0 0 0 0.05
do
    wait 2
    $status = altaz read task_status
    $pos = altaz read position
    $azPos = substring 1 $pos
    print "azimuth:,$azPos,\n"
while $status == $running

label DEAD_ALTAZ:
# reboot the AAC
altaz state reboot
altaz serial close

endprog close socket
```

The following example demonstrates turning the servo by using the Alt-Az position log. This example program assumes that the elevation axis has been initialized, and that the motor and servo parameters have been loaded into the AAC.

```
#######################################
# servo_tuning_demo.irma
# demonstrate tuning the elevation axis
#######################################
startprog socket open

altaz serial open

assign $running 2

########################
# Start position capture
########################

altaz state poslog poslog_clear
altaz state poslog poslog log_enable

altaz moveto dms altitude 90 0 0 0 0 0 4
do
    wait 1
    $status = altaz read task_status
```

```
altaz poslog disable

#########################################################
# playback position log values.  These values could be
# captured by redirecting terminal output to file
#########################################################
$tuneValues = altaz read poslog_range
print $tuneValues \n

$minIndex =  substring $tuneValues 0
$maxIndex = substring $tuneValues 1
$currIndex = substring $tuneValues 2
print "min_index:,$minIndex,\n"
print "max_index:,$maxIndex,\n"
print "curr_index:,$currIndex,\n"

assign $cnt 0
do
    $data = altaz read poslog_data $cnt
    print "$data,\n"
    incr $cnt
while $cnt < $currIndex

altaz serial close

endprog socket close
```

# Bibliography

[1] European Space Agency. Packet Telecommand Standard. Technical Report PSS-04-017, Packet Telecommand Standard, Issue 2, April 1992.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, Reading, MA, 1986.

[3] Eric Aristidi. Photo of Concordia Station, Dome C, Antarctica. Universitè de Nice.

[4] Bradley W. Carroll and Dale A. Ostlie. *An Introduction to Modern Astrophysics*. Addison Westley, Reading, MA, 1996.

[5] I. M. Chapman, D. A. Naylor, and R. R. Phillips. Correlation of Atmospheric Opacity Measurements by SCUBA and an Infrared Radiometer. *Monthly Notices of the Royal Astronomical Society*, 354(2):621–628, 2004.

[6] Ian M. Chapman. The Atmosphere Above Mauna Kea at Mid-Infrared Wavelengths. Master's thesis, Univeristy of Lethbridge, 2002.

[7] Cirrus Logic, Inc., Austin, TX. *Cirrus Logic CS5531/32/33/34 Data Sheet Errata, Single Conversion Mode Timing*, March 2000.

[8] Cirrus Logic, Inc., Austin, Texas. *CS5531/32/33/34 Product Data Sheet*, September 2004. www.cirrus.com/en/pubs/proDatasheet/CS5531-32-33-34_F1.pdf.

[9] Diamond Systems Corporation. *Emerald-MM-DIO Quad RS-232 + 48 Digital I/O PC/104 Users Manual*. Newark, CA, 2002. www.tri-m.com/products/diamond/files/manual/emmdio_man.pdf.

[10] Thermo Electron Corporation. Revco Web Page. World Wide Web, 2005. www.revco-sci.com/catalog/ult/value/86_chest_specs.html.

[11] H. M. Deitel and P. J. Deitel. *Java: How to Program*. Prentice Hall, Upper Saddle River, NJ, 1997.

[12] A. James Diefenderfer and Brian E. Holton. *Principles of Electronic Instrumentation*. Saunders College Publishing, Philadelphia, PA, third edition, 1994.

[13] Peter Duffett-Smith. *Practical Astronomy With Your Calculator*. Cambridge University Press, Cambridge, second edition, 1981.

[14] Tri-M Engineering. *TMZ104 User Manual*. Port Coquitlam, BC, 2003. engineering.tri-m.com/products/engineering/files/manual/tmz/tmz104

[15] Globalsat Technology Corporation, Taipei, Taiwan. *GPS Receiver Engine Board Software Command*, January 2001. www.usglobalsat.com/downloads/General Downloads/GPS/NEMA_commands.pdf.

[16] Per Brinch Hansen. *Brinch Hansen on Pascal Compilers*. Prentice Hall, Englewood Cliffs, NJ, 1985.

[17] Theresa Hitchens. Space-Based Missile Defense: Not So Heavenly. World Wide Web, July 2003. www.cdi.org/friendlyversion/printversion.cfm?documentID=1487.

[18] Aaeon Systems Inc. *PCM-3660/3661 PC/104 Ethernet module*. Hazlet, NJ, May 1995. ftp://data.aaeonsystems.com/DOWNLOAD/MANUAL/PCM-3660Manual.pdf.

[19] Aerotech Inc. Unidex 500 PC-Card Multi-Axis Motion Controller. World Wide Web. www.aerotech.com/products/pdf/u500.pdf.

[20] Rabbit Semiconductor, Inc. Rabbit 2000 Microprocessor Interrupt Problem. Technical Note TN301, Rabbit Semiconductor, Davis, CA. www.rabbitsemiconductor.com/documentation/docs/refs/TN301/TN301.pdf.

[21] Rabbit Semiconductor, Inc. Rabbit Memory Management In a Nutshell. Technical Note TN202, Davis, California. www.zworld.com/documentation/docs/refs/TN202/TN202.pdf.

[22] Rabbit Semiconductor, Inc. Company Info: About Rabbit Semiconductor. World Wide Web, 2005. www.rabbitsemiconductor.com/company/aboutUs.shtml.

[23] Z-World, Inc. Rabbit Semiconductor Technical Bulletin Board. World Wide Web. www.zworld.com/support/bb/index.html.

[24] Texas Instruments. Applying Oversampling Data Converters: 2002 Signal Acquisition and Conditioning for Industrial Applications Seminar. focus.ti.com/docs/training/catalog/events/event.jhtml?sku=SEM402009, 2002.

[25] Liviu Ivanescu. Arctic Astronomy. www.eso.org/gen-fac/pubs/astclim/espas/Arctic, October 2004.

[26] Rutherford Appleton Laboratory. SPIRE Homepage. World Wide Web, 2002. www.ssd.rl.ac.uk/SPIRE/.

[27] Jean J. Labrosse. *MicroC/OS-II: The Real-Time Kernel*. CMP Books, Lawrence, KS, second edition, 2002.

[28] J.S. Lawrence, Ashley, Burton M.C.B, M.G., and et al. The AASTINO: Automated Astrophysical Site Testing INvincible Observatory. *Memorie della Societa Astronomica Italiana Supplementi*, 2:217–220, 2003.

[29] O. P Lay. MMA Memo 209: 183 GHz Radiometric Phase Correction for the Millimeter Array. Web, 1998. www.alma.nrao.edu/memos/html-memos/alma209/mma209.ps.gz.

[30] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[31] M-Systems, Inc., Newark, CA. *DiskOnChip 2000 DIP Data Sheet*, September 2004. www.m-systems.com.

[32] Bruce J. MacLennan. *Principles of Programming Languages: Design, Evaluation and Implementation*. Holt, Rinehart and Winston, New York, second edition, 1987.

[33] Howard V. Malmstadt and Christie G. Enke. *Electronics and Instrumentation for Scientists*. The Benjamin/Cummings Publishing Company, Inc., Reading, MA, 1981.

[34] Maxim Integrated Products, Sunnyvale, California. *MAXIM MAX5223 Low-Power, Dual 8-Bit, Voltage Output Serial DAC in 8-Pin SOT23*, 2001. pdfserv.maximic.com/en/ds/MAX5223.pdf.

[35] Maxim Integrated Products, Sunnyvale, CA. *Maxim MAX6126 Ultra-High Precision, Ultra-Low-Noise Series Voltage Reference*, July 2004. pdfserv.maximic.com/en/ds/MAX6126.pdf.

[36] Maxon Motor AG, Sachseln, Switzerland. *Maxon Motor Control 1-Q-EC Amplifier DEC50/5*, January 2003.

[37] David A. Naylor, Ian M. Chapman, and Bradley G. Gom. Measurements of Atmospheric Water Vapor Above Mauna Kea Using an Infrared Radiometer. In Joseph A. Shaw, editor, *Proceedings of SPIE Volume 4815: Atmospheric Radiation Measurements and Applications in Climate*, pages 36–45, Bellingham, WA, September 2002. SPIE - The International Society for Optical Engineering.

[38] David A. Naylor, Bradley G. Gom, Ian S. Schofield, and et al. Remotely Operated Infrared Radiometer for the Measurement of Atmospheric Water Vapor. In Marija Strojnik Bjorn F. Andresen, Gabor F. Fulop, editor, *Proceedings of SPIE: Infrared Technology and Applications XXVIII*, pages 208–928, Bellingham, WA, 2003. SPIE - The International Society for Optical Engineering.

[39] H. T. Nguyen, Bernard J. Rauscher, Scott A. Severson, and et al. The South Pole Near-Infrared Sky Brightness. *Publications of the Astronomical Society of the Pacific*, 108:718–720, August 1996.

[40] European Southern Observatory. ESO Search of Potential Astronomical Sites Working Groups Homepage. World Wide Web, 2000. www.eso.org/gen-fac/pubs/astclim /espas/.

[41] National Radio Astronomy Observatory. Atacama Large Millimeter Array. World Wide Web, April 2005. www.alma.nrao.edu/ALMAHandout/Apr05.pdf.

[42] Charles Petzold. *Programming Windows*. Microsoft Press, fifth edition, 1999.

[43] Robin R. Phillips, David A. Naylor, James diFrancesco, and et. al. Initial Results of Field Testing an Infrared Water Vapor Monitor for Millimeter Astronomy (IRMA III) on Mauna Kea. In Jr. Jacobus M. Oschmann, editor, *Proceedings of SPIE Volume: 5489*, pages 146–153, Bellingham, WA, September 2004. SPIE - The International Society for Optical Engineering.

[44] Maxim Integrated Products. Demystifying Sigma-Delta ADCs. Application Note 1870, Maxim Integrated Products, January 2003. www.maximic.com/appnotes.cfm/ appnote_number/1870.

[45] Rabbit Semiconductor, Inc., Davis, CA. *Introduction to TCP/IP*, 2001. www.rabbitsemiconductor.com/documentation/docs/manuals/TCPIP/Introduction/ tcpintro.pdf.

[46] Rabbit Semiconductor, Inc., Davis, CA. *Roadmap to TCP/IP Sample Programs*, 2003. www.rabbitsemiconductor.com/documentation/SamplesRoadmap/tcpip-roadmap.pdf.

[47] Rabbit Semiconductor, Inc., Davis, CA. *Dynamic C TCP/IP User's Manual Vol. 1*, 2004. www.rabbitsemiconductor.com/documentation/docs/manuals/TCPIP/UsersManualV1/tcpV1.pdf.

[48] Rabbit Semiconductor, Inc., Davis, CA. *Dynamic C TCP/IP User's Manual Vol. 2*, 2004. www.rabbitsemiconductor.com/documentation/docs/manuals/TCPIP/UsersManualV2/tcpV2.pdf.

[49] Rabbit Semiconductor, Inc., Davis, CA. *RCM2000 RabbitCore Data Sheet*, 2005. www.rabbitsemiconductor.com/products/rcm2000/rcm2000.pdf.

[50] Rabbit Semiconductor, Inc., Davis, CA. *RCM2100 RabbitCore Data Sheet*, January 2005. www.rabbitsemiconductor.com/products/rcm2100/rcm2100.pdf.

[51] Rayming Corporation, Industry, CA. *GPS Development Kit DK-ER102*, September 2003. www.usglobalsat.com.

[52] RTD Embedded Technologies, Inc., State College, PA. *CML16686GX cpuModule User's Manual*, 2003. www.rtd.com/NEW_manuals/hardware/cpumodules/CML16686GX.pdf.

[53] Ian Schofield. Test Facility FTS Data ICD. Technical Report SPIRE-UoL-PRJ-001452, Herschel / Spire Project, January 2004.

[54] Ian S. Schofield and David S. Naylor. Instrumentation Control Using the Rabbit 2000 Embedded Microcontroller. In Gianni Raffi Hilton Lewis, editor, *Proceedings of SPIE Volume: 5496*, pages 392–401, Bellingham, WA, September 2004. SPIE - The International Society for Optical Engineering.

[55] Rabbit Semiconductor. *Rabbit 2000 Microprocessor Designer's Handbook*. Davis, California, 2000. www.rabbitsemiconductor.com/documentation/docs/manuals/Rabbit2000/DesignersHandbook/R2000DH.pdf.

[56] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts*. Addison Wesley Longman, Inc., Reading, MA, 1997.

[57] Graeme J. Smith. An Infrared Radiometer for Millimeter Astronomy. Master's thesis, University of Lethbridge, 2001.

[58] J. W. V. Storey, M. C. B Ashley, J. S. Lawrence, and et al. Dome C - The Best Astronomical Site in the World? *Memorie della Societa Astronomica Italiana Supplementi*, 2(13):13 – 18, 2003.

[59] Arie Tal. Two Technologies Compared: NOR vs. NAND White Paper. Technical Report 91-SR-012-04-8L, Rev 1.1, M-Systems, July 2003. www.m-systems.com/files/documentation/doc/nor_vs_nand.pdf.

[60] OPTEK Technologies. Product Bulletin OPB930L. World Wide Web, July 1996. www.optekinc.com/pdf/OPB930L.pdf.

[61] US Digital Corporation, Vancouver, WA. *LS7266R1 Encoder to Microprocessor Interface Chip*, March 2004. www.usdigital.com/products/ls7266/.

[62] US Digital Corporation, Vancouver, WA. *E6 Optical Kit Encoder*, July 2005. www.usdigital.com/data-sheets/E6 Data Sheet.pdf.

[63] Frank Vahid and Tony Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley and Sons, Inc., Hoboken, NJ, 2002.

[64] Tim Wescott. PID Without a PhD. *Embedded Systems Programming*, October 2000. www.embedded.com/2000/0010/0010feat3.htm.

[65] Z World, Inc., Davis, California. *Dynamic C User's Manual*, September 2004. www.zworld.com/documentation/docs/manuals/DC/DCUserManual/DCPUM.pdf.